Received February 1, 2022, accepted March 2, 2022, date of publication March 3, 2022, date of current version March 11, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3156692

Learning Future Reference Patterns for Efficient Cache Replacement Decisions

HYEJEONG CHOI AND SEJIN PARK
Department of Computer Science, Keimyung University, Daegu 1095, South Korea

Corresponding author: Sejin Park (baksejin@kmu.ac.kr)

This work was supported by the Institute for Information & Communications Technology Promotion (IITP) funded by the Korean Government [Ministry of Science and ICT (MSIT)] through the Core Technologies for Hybrid P2P Network-Based Blockchain Services under Grant 2021-0-00484.

ABSTRACT This study proposes a cache replacement policy technique to increase the cache hit rate. This policy can improve the efficiency of cache management and performance. Heuristic cache replacement policies are mechanisms that are designed empirically in advance to determine what needs to be replaced. This study explains why the heuristic policy does not achieve a high accuracy for certain patterns of data. A machine learning method is proposed to predict the blocks that need to be requested in the future to prevent erroneous decisions. The core operation of the proposed method is that when a cache miss occurs, the machine learning model predicts a future block reference sequence that is based on the block reference sequence of the input sequence. The predicted block is added to the prediction buffer and the predicted block is removed from the non-access buffer if it exists in the non-access buffer. After filling the prediction buffer, the conventional replacement policy can be replaced with a time complexity of O(1) by replacing the block with a non-access buffer. The proposed method improves the least recently used (LRU) algorithm by 77%, the least frequently used (LFU) algorithm by 65%, and the adaptive replacement cache (ARC) by 77% and shows a hit rate similar to that of state-of-the-art research. The proposed method reinforces the existing heuristic policy and enables a consistent performance for LRU- and LFU-friendly workloads.

INDEX TERMS Machine learning, cache memory, performance evaluation.

I. INTRODUCTION

Cache is a concept that is used to reduce the performance difference between storage layers; it is applied in a variety of fields such as operating systems, databases, and network systems [1]– [3]. For example, solid-state drives (SSDs) provide faster speeds than hard disk drives (HDDs), but a system performance bottleneck remains because the central processing unit (CPU) and dynamic random access memory (DRAM) provide three times lower access latency [4]. To address this bottleneck, a cache is placed between the storage tiers to store frequently used items. This allows the requested item to be placed in a cache that has faster access than the SSD without direct access to the SSD. However, although access through the cache is faster than through the lower-tier storage, it is much smaller; thus, the items in the cache must be managed efficiently. Cache replacement is a technique that increases the efficiency of the cache. It selects an item to be removed from the cache when the requested item is not in the cache, and a new space is required to add an item to the cache. When selecting an item to be replaced, an item that is determined to be the most unnecessary in the future should be selected. In this case, the optimal cache replacement algorithm is Belady's MIN algorithm [5]. However, this algorithm cannot be practically realized because the reference to all the future items must be known. Various algorithms have been proposed as alternatives to the optimal algorithm, and the most well-known heuristic algorithms are the least recently used (LRU) and the least frequently used (LFU). The LRU replaces the block that was requested the oldest, and the LFU replaces the block with the least pollution-requested frequency. Each of the LRU and LFU policies contains advantages and disadvantages. The LRU adapts well to changes in the working set, but it has the worst performance when a set of identical items that is larger than the cache size has a looping pattern that has to be accessed at regular intervals. The LFU adapts well to the looping pattern, but it cannot adapt to current changes because it only remembers the previously requested number of times when the working set changes frequently. To solve this problem, policies that

The associate editor coordinating the review of this manuscript and approving it for publication was Hao Luo.

combine these two algorithms have been proposed [6], [7]. These policies utilize the past and currently accessed items to determine the replacement items. The combined policies show superior results in comparison to the LRU and LFU, but there are problems because they still cannot predict the future. For example, adaptive replacement cache (ARC) has the same problem as the LRU, wherein the performance is degraded when the working set is larger than the cache size [8]. In addition, there is a possibility of cache pollution. This is because long-term utility is not guaranteed when the correlated references, in which the same page is quickly referenced more than once at the upper level of the memory hierarchy, are frequently observed [9].

These heuristic-based policies such as the LRU, LFU, and ARC show a good performance in the pattern that is targeted by the heuristic. However, numerous patterns can exist, and the performance of these policies is poor when there are a variety of patterns. Therefore, this study presents an idea to help predict the future that is based on past information so that the conventional heuristic-based replacement policy can make correct decisions based on these predictions. Predicting the future based on past information is possible through machine learning; machine learning can learn a complex distribution of data that is difficult to interpret. Considerable research has been conducted on the application of machine learning to cache replacement [1], [3], [10]–[12]. This study confirmed the applicability of machine learning.

In this study, to predict future information from past information, a model that uses the information of the block-level I/O traces were collected from a variety of applications as the training data; the model can predict the block number sequence. To learn the pattern of consecutive block numbers in the past, the model was trained using the sequence-to-sequence (Seq2Seq) [13] model that is based on long short-term memory (LSTM) [14] networks. During the simulation, when a new block is accessed and cache replacement is required, the trained model predicts the next block to be accessed and it stores the block in order in the prediction buffer. In addition, we propose a non-access buffer that stores blocks that exist in the cache but are not referenced in the future. If the predicted block exists in the non-access buffer, the block is removed from the non-access buffer. The conventional cache replacement policy determines whether to remove the block by checking the prediction buffer that stores the block to be accessed in the future. In addition, to increase the accuracy of the prediction buffer, the actual request block and predicted block are checked. If the number of failures is greater than or equal to a set threshold, the model fills the prediction buffer again to increase the accuracy of the prediction buffer. Based on the traces used for the previous cache replacement policy evaluation, we evaluate the proposed method. Experimental results show that while the proposed method outperforms heuristic cache replacement policy with a significant margin by identifying future sequence that policies such as LRU and LFU do not recognize, and show a slight performance improvement or similar performance when compared with the state-of-the-art cache replacement policy [10].

- We designed a Seq2Seq-based model that uses the Block I/O as the input; this model predicts the block sequence to be accessed. This predicted block sequence it is applied to the conventional heuristic replacement policy.
- By combining heuristic cache replacement with the future sequence predicted by the model, unnecessary replacements can be prevented in advance by identifying reference patterns that are not explored by existing cache replacement policies.
- In the LRU-friendly workload and LFU-friendly workload, the hit rate of the proposed model outperforms the LRU by 77% and the LFU by 65%. This shows that the model works in both workloads well.

II. RELATED WORK

A. RECURRENT NEURAL NETWORK

A recurrent neural network (RNN) is a model for handling sequence data. Sequence data are data in which the order of the data is preserved, and the data are related instead of being independent entities. The existing neural network ignores the temporal aspect of the data and recognizes them as independent entities. The proposed model that needs to be considered for this aspect is an RNN. Unlike the existing neural networks, RNNs can remember hidden states. When a new input is introduced, the RNN changes the hidden state slightly and it becomes the hidden state of the next input; thus, preserving the previous memory. There are various models using the RNN, the most famous of which is the long short-term memory (LSTM). The LSTM is designed to solve the vanishing gradient problem, which is a weakness of the RNN. The main characteristic of LSTM is that a cell state that can preserve the long-term memory is added to the hidden state, and the cell state is controlled by adding three gates. First, the forget gate determines what cell state information is forgotten. Second, the input gate determines which value to store in the cell state. Finally, the output gate is responsible for deciding what should be sent to the output. There is a Seq2Seq model that is composed of these two RNN models, which has two characteristics: the long-term memory capability of the RNN model and the output of the sequences of the different domains from the input sequence. The two RNN models are divided into the encoder and decoder. The encoder receives the sequence called the source as the input; it compresses the meaning of the source and transmits it to the hidden state of the decoder. The output value of the compressed encoder is called the context vector. The decoder uses the context vector as the initial value of the hidden state and outputs the sequence values one by one.

B. CACHE MANAGEMENT

The least recently used (LRU) algorithm is a technique that replaces the block that was requested the oldest in the cache. This technique can adapt well to the request patterns that show temporal and spatial localities. However, as the blocks that are requested frequently or are requested less often cannot be identified, the sequential reference that is not reused in the future may unnecessarily occupy cache space.

The least frequently used (LFU) technique replaces the block with the smallest number of requests when considering the blocks in the cache. It works well when it has a looping pattern that does not change the workload. However, as the workload changes over time, cache pollution occurs because the old blocks remain in the cache.

The adaptive replacement cache (ARC) [6] maintains two LRU lists. The two LRU lists are called L1 and L2, where L1 keeps the pages that are accessed only once, and L2 keeps the pages that are accessed at least twice. L1 is classified as T1 and B1, and L2 is classified as T2 or B2. T1 and T2 are lists in the cache, and B1 and B2 are ghost buffers that contain the pages that are removed from T1 and T2. The main characteristic of the ARC is that the sizes of T1 and T2 can be dynamically set by adjusting the parameter p, rather than classifying T1 and T2 into a fixed size. The parameter p is adjusted based on B1 and B2; it can quickly adapt to the workload based on the past history.

Recently, several attempts have been made to use machine learning for efficient cache management. DLIRS [15] was inspired by ARC that dynamically allocates cache space, and proposed a method of dynamically allocating LIR and HIR space in LIRS policy. Learning cache replacement (LeCaR) [10] is an algorithm that is based on the RL; it minimizes the regret to select the LRU and LFU according to the current request pattern. DeepMEC [2] can determine the content that needs to be stored in the cache by predicting the popularity of the content based on the RNN, the convolution neural network (CNN), and the CRNN in order to maintain the popular content in the cache when the edge node uses the cache for content management. PARROT [3] approximates Belady's MIN using imitation learning (IL) [16] to improve the performance of the CPU cache replacement. Glider [13] predicts whether each current program counter (PC) is cache-friendly or cache-averse based on an integer support vector machine (ISVM). Pseudo-OPT [12] can predict the future reuse distance based on the past history using the LSTM as a model. Learning Relaxed Belady (LRB) [1] uses a Gradient Boosting Machine (GBM) to select candidates by randomly sampling objects in the cache to obtain cache replacement candidates. Among the selected candidates, the candidate with the longest request distance is removed from the cache. CACHEUS [17] removes the static hyperparameters present in LeCaR and can resist scan and churn types, thus eliminating the disadvantages of LRU and LFU workloads. Therefore, design a model that can identify more types of workloads to show better performance. This paper predicts future reference sequences differently from some studies that utilize machine learning models for the selection of candidates in the cache. Therefore, this study differentiates it from some state-of-the-art studies by bringing the advantages of the existing cache replacement policy as it is and supplementing the shortcomings of the existing cache replacement policy.

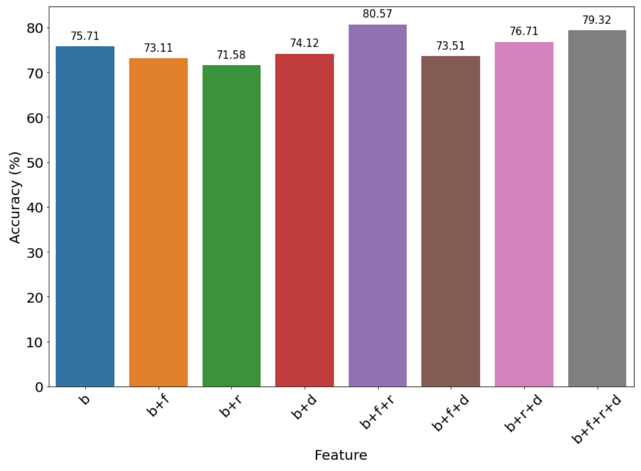


FIGURE 1. The validation accuracy for input feature.

III. PROBLEM FORMULATION

To design our model based on ML solution, we formulate cache replacement as a probabilistic prediction problem and view its output as a probability distribution. The goal of efficient cache replacement is to exploit relationship between access blocks and next access blocks. Therefore, we use the method of assigning probabilities to word sequence in a language model to predict future block sequence. The input sequence is called X, and one input existing in X is expressed as a lowercase letter x. When expressing the probability of appearance of input data, it can be expressed as $P(X) = P(x_n, \ldots, x_1)$. When the probability of data appearing after x_n is $P(y_1)$ based on data probability, it can be expressed as $P(y_1|x_n, \ldots, x_1)$ if expressed as a conditional probability. The next occurrence probability y_1 can be used to predict the occurrence probability of y_2 following y_1, which is $P(y_2|y_1, x_n, \ldots, x_1)$, $P(y_3|y_2, y_1, x_n, \ldots, x_1)$, $P(y_m|y_{(m-1)}, \ldots, y_1, x_n, \ldots, x_1)$. In conclusion, we define this study as a classification problem for predicting the next block, and design a model to understand the contextual flow as a transition block leads to the next block.

IV. PROPOSED METHOD

This section describes the design method of the model proposed in this study based on the above formulation in the following order. First, selection of input and output data related to future blocks, second, selection of a machine learning model that can accurately predict the probability of appearance, and finally, design as a method used for cache replacement based on the value output by the designed model.

A. CONFIGURATIONS

1) TRAINING DATA

This subsection explores the data to use as the input and the data to use as the output. Exploration is important, because each component influences the our model. Our model

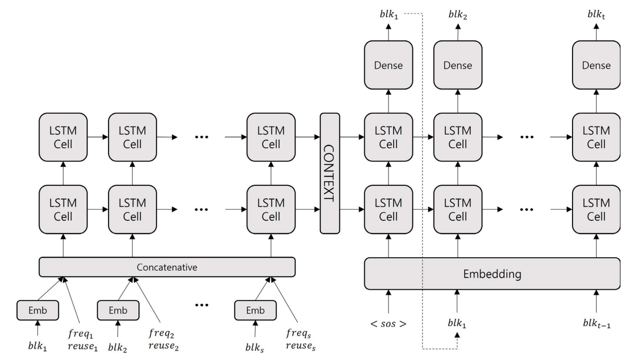


FIGURE 2. Seq2Seq: Block I/O stream will be encoded and decoded in a sequence that can be for Seq2Seq learning.

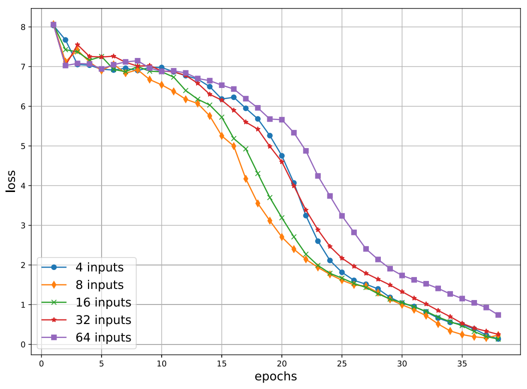


FIGURE 3. The validation loss to find a suitable number of source sequences data.

considers the block number, frequency, reuse distance and delta as inputs. Frequency is the number of requests for which the block number is requested, reuse distance is the number of blocks requested between two consecutive identical request blocks, and delta is the difference between consecutive blocks. Figure 1 shows the validation accuracy when four input features are combined: block number (b), frequency (f), reuse distance (r), and delta (d). In our experiments, the block number is converted into an embedding vector and frequency, reuse distance, and delta are normalized. In conclusion, the block number, frequency, and reuse distance with the highest validation accuracy are selected as input features. Our model considers the block number and delta as output. We analyze the coverage of block numbers and deltas to reduce the number of classes that the model needs to predict. Based on traces used in the evaluation section (Section 5), we obtain the frequency of the unique block number and delta found in all requests, sort them in descending order of frequency, and calculate the coverage of the top 1000 frequencies for all frequencies. As shown in Table 1, the block number is used as the output because the block number shows higher coverage in traces excluding cscope, glimpse, and postgres.

2) MODEL ARCHITECTURE

The architecture of the proposed model uses a LSTM-based Seq2Seq network that can predict the next sequence of the I/O stream [18]. The architecture of the proposed model is illustrated in Figure 2. In Figure 2, *blk* is the block number, *freq* is the frequency, and *reuse* is the reuse distance. To preserve the meaning of the block number sequence, there is an embedding layer that consists of 256 neurons in the input layer. The model has two hidden LSTM layers, where each LSTM layer has 256 hidden nodes. To determine the sequence length of the encoder, we evaluate the validation loss at 4, 8, 16, 32, 64 as shown in Figure 3. We choose 8 as the input sequence length because training converges quickly when it is 8. The first input of the decoder starts with "<sos>" and outputs the block number to be accessed. Because the length of the output sequence is fixed at 3, a symbol indicating the end of the sequence is not added. The final output layer of the decoder is a dense layer with softmax [19]. The number of output neurons in the density layer includes the number of unique blocks that are found in the training dataset and "<unknown>", which is unpredictable.

B. BEHAVIOR

The operation process of the proposed method for the role is explained through figures and algorithms. Figure 4 shows the overall architecture of the proposed method. The architecture has six components: the request queue, history buffer, prediction buffer, non-access buffer, Seq2Seq, and the cache. The request queue stores the actual requested blocks (RBs) in a temporal order; it keeps the RBs in the form of RB1, RB2, …, which is the request time order from the past to the future. In addition, t_{now} is the time of the currently requested block, t_{now+1} and t_{now+2} are expressed as the time of the block that may be requested in the future, and t_{now-1} and t_{now-2} are expressed as the times of the block that were requested in the past. The history buffer has the same length as the input sequence of the model, and the information of blocks are recorded in the most recently used (MRU) location in the requested order. When the input sequence length set in Figure 3 is four, a total of four information that exist in the current referenced time t_{now} to t_{now-3} are stored in the request order in the history buffer. The prediction buffer has the same size as the cache, and the blocks that are predicted to be requested in the future by the Seq2Seq model are recorded in the MRU location along with the order of the requests. When the cache size set in Figure 3 is six, a total of six information exist at the next time t_{now+1} to t_{now+6} after the currently referenced time t_{now} is stored in the order of the request in the prediction buffer.

The block at the LRU position in the prediction buffer is the block that is predicted to be requested next. Because the predicted block (PB1) is removed every time a new block request arrives, the LRU position always maintains the block that is predicted to be requested next. For example, when a block is requested at the next time t_{now+1}, the LRU position (PB1) of the prediction buffer stores the block (RB5) to be requested in t_{now+1}. Therefore, because the block at the LRU position in the prediction buffer is the same as the block at the current time t_{now+1}, the prediction buffer maintains the block to be requested at the next time t_{now+2} by deleting the

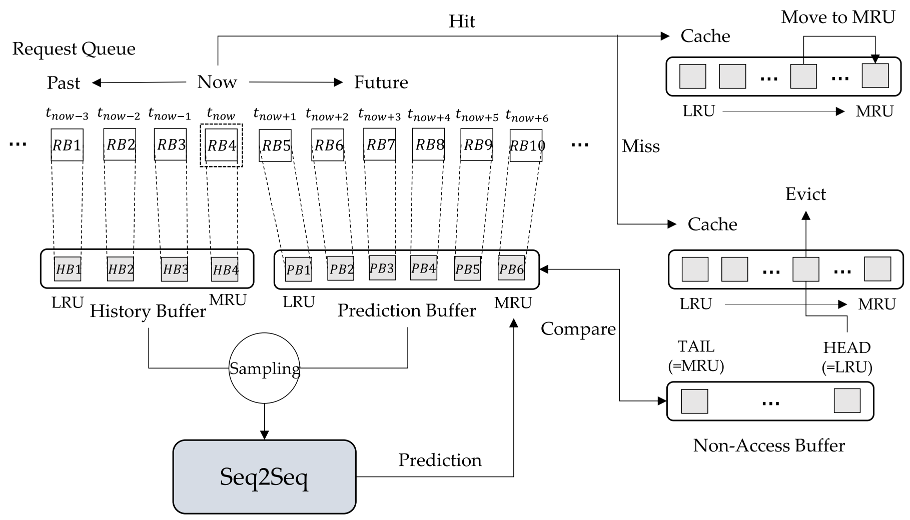
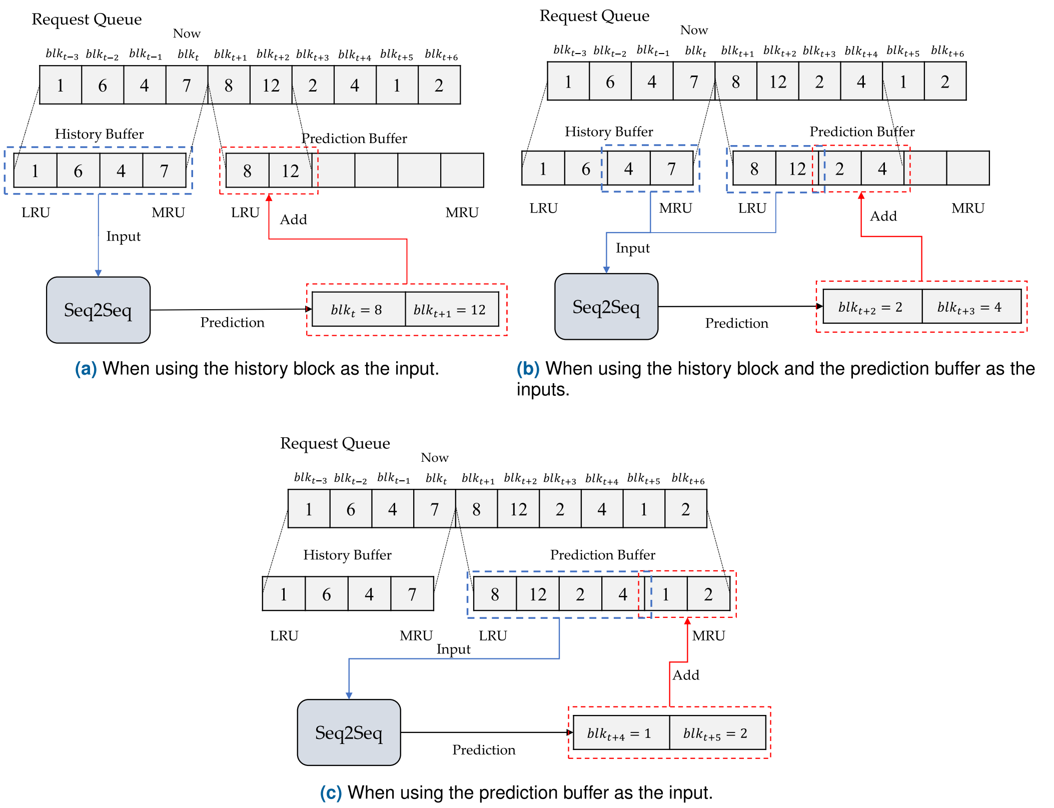


FIGURE 4. Proposed architecture overview.



(a) When using the history block as the input.

(b) When using the history block and the prediction buffer as the inputs.

(c) When using the prediction buffer as the input.

FIGURE 5. Prediction buffer update.

block (t_{now+1}) at the LRU position. Conversely, when moving to the next request, the predicted block at the LRU position of the prediction buffer is deleted from the prediction buffer, and the deleted block and requested block are compared for equality. If the two values are different, the number of prediction failures is increased; if the number of prediction failures is the same as the set value (threshold), there is an empty prediction buffer and it fills the prediction buffer again. The non-access buffer is a buffer that stores cache internal blocks that are judged not to be requested in the future. If the predicted block exists in the buffer, it is removed from the buffer because the buffer assumes that the block currently in the cache will be requested in the future. The reason that non-access buffer exists is to obtain information from the buffer because the cache replacement policy removes blocks that will not be requested in the future. Seq2Seq predicts the blocks to be requested in the future based on the input. In Seq2Seq, the input data are sampled from the history buffer and prediction buffer, and there are three cases, which are described as follows.

1) PREDICTION BUFFER UPDATE POLICY

Figure 5 illustrates the three cases of sampling the input data. The request queue stores the blocks that are requested from the past to the future and are now requested. The cache represents the current state of the cache, and the cache size and the size of the prediction buffer are set to six, the length of the input sequence is set to four, and the length of the output sequence is set to two. In addition, *blk* represents the request blocks that exist in the request queue. When the current request time is t, blk_{t+1} and blk_{t+2} are the blocks that will be requested in the future, and blk_{t-1} and blk_{t-2} are the blocks that were requested in the past.

The first sampling method is when the prediction buffer is empty; here the history buffer is taken and used as the input to the model. In Figure 5 (a), as the history buffer stores as much as four, which is the input sequence length of Seq2Seq, blk_t to blk_{t-3} exists in the history buffer. Therefore, the information of blocks that are present in the history buffer are used as the input of Seq2Seq, and Seq2Seq predicts two blocks. The two predicted blocks store in the prediction buffer in the requested order.

The second sampling method is when the prediction buffer is filled, but the size of the prediction buffer is not sufficient for the length of the input sequence. In this case, a part of the history buffer and the information of blocks that are stored in the prediction buffer are required. In Figure 5 (b), the blocks that are currently stored in the prediction buffer are blk_{t+1} and blk_{t+2}; thus, it is necessary to predict it from the next blk_{t+2}. To predict the next block, four input values from blk_{t-1} to blk_{t+2} are required. Therefore, blk_{t-1} and blk_t are stored in the history buffer, and blk_{t+1} and blk_{t+2}, which are stored in the prediction buffer, are concatenated and used as the input sequence for Seq2Seq. The predicted blk_{t+3} and blk_{t+4} are stored in the prediction buffer for the order of the request.

The last sampling method is when the blocks that exist in the prediction buffer exist as much as the length of the input sequence. In Figure 5 (c), because the last block to be requested in the current prediction buffer is blk_{t+4}, it is necessary for the model to predict the blocks from the next requested blk_{t+5}. To make a prediction from blk_{t+5}, the input sequence of Seq2Seq is required from blk_{t+1} to blk_{t+4}. Therefore, because the required input sequence exists in the prediction buffer, the model predicts the block numbers using blk_{t+1} to blk_{t+4} as the input sequence. The predicted blk_{t+5} and blk_{t+6} are stored in the prediction buffer for the order of the request.

2) CACHE UPDATE POLICY

The cache is the actual cache memory that stores the requested blocks. When the cache hits, the requested block is moved to MRU. This is because when all the blocks in the cache do not exist in the prediction buffer or all the blocks in the prediction buffer exist in the cache, the cache is replaced based on the LRU. When the cache is missing, the blocks in the head position of the non-access buffer are removed from the cache. The reason is that the existing replacement policy follows the LRU, so blocks in the non-access buffer are stored in the order of the request from head to tail. Therefore, in our replacement policy, the oldest requested block among blocks that are not referenced in the future is selected for replacement.

Algorithm 1 Proposed Method(PB,HB,NAB)

```
Require: requested block $b$
 1: $FaultCount \leftarrow 0$
 2: if $HB$.Length is InputLength then
 3:     $HB$.DELETEFRONT();
 4: end if
 5: $HB$.ADD($b$);
 6: if $PB$ is not empty then
 7:     if $PB$.FRONT is not $b$ then
 8:         $FaultCount \leftarrow FaultCount + 1$
 9:         if $FaultCount$ is Threshold then
10:             $PB$.CLEAR()
11:             $FaultCount \leftarrow 0$
12:         end if$PB$.DELETEFRONT();
13:     end if
14: end if
15: if $b$ is in $C$ then
16:     $C$.UPDATE($b$);
17: else
18:     if $C$ is full then
19:         while $B$.Length $<$ $C$.Capacity do
20:             $inputs \leftarrow$ [];
21:             if $PB$ is empty then
22:                 $start \leftarrow HB$.Length;
23:                 $start-$ =InputLength;
24:                 $inputs$.ADD($HB[start,]$);
25:             else
26:                 if $B.Length < InputLength$ then
27:                     $start \leftarrow HB.Length$;
28:                     $start- = InputLength - PB.Length$;
29:                     $inputs$.Add($HB[start,]$);
30:                     $inputs$.Add($PB[,]$);
31:                 else
32:                     $start = PB.Length - InputLength$;
33:                     $inputs$.ADD($PB[start,]$);
34:                 end if
35:             end if
36:             $blocks = model$.PREDICT($inputs$);
37:             $PB$.ADD($blocks$);
38:             $NAB$.DELETE($blocks$);
39:         end while
40:         $victim = NAB$.HEAD().$block$;
41:         $NAB$.DELETE($victim$);
42:         $C$.DELETE($victim$);
43:     end if
44:     $C$.ADD($b$);
45: end if
46: $NAB$.ADD($b$);
```

TABLE 1. Dataset description.

Dataset Name	Summary	Reference Count	Coverage BlockNo. (%)	Coverage Delta (%)
glimpse	Text information retrieval utility	6015	69.78%	100%
postgres	Joins queries among four relations in a relational database system	10448	76.67%	100%
cscope	An interactive C source program examination tool	6781	73.74%	100%
sprite	The Sprite network file system	133996	81.59%	64.77%
2-pools	Multi-user database	100000	58.45%	10.18%
multi1	cscope, cpp	15858	75.43%	70.84%
multi2	cscope, cpp, postgres	26311	62.83%	56.08%
multi3	cpp, gnuplot, glimpse, postgres	30241	56.36%	51.19%
web07	Product detail page HTTP requests	76118	51.86%	32.78%
web12	Product detail page HTTP requests	95607	66.59%	29.76%

Algorithm 1 shows the process that was previously described when a new block is requested. Because the history buffer (HB) is requested as much as the input sequence length of the model, if it exceeds the input sequence length, the oldest block is removed, and the currently requested block b is added (lines 2–4). A comparison is performed between the block that is located in the LRU of the prediction buffer and the current block b. If the two blocks are different, the number of model failures are increased (FaultCount) by one, and the current prediction buffer is emptied if the threshold and FaultCount are equal. If the prediction buffer is not empty, the block at the LRU position of the prediction buffer is removed (lines 6–14). If the currently requested block b exists in the cache, it is moved to the MRU location according to the LRU policy (lines 15–16). When the block does not exist in the cache and the cache is full, the model makes predictions that are based on the input sequence to fill the prediction buffer (PB). The inputs consist of the input sequence of the model, and there are three cases of input generation (see Fig. 5 (a), (b), (c), lines 20–38). When a prediction block exists in a non-access buffer (NAB), it is excluded from cache replacement (line 39). Finally, if the prediction buffer is full, the block at the head of the non-access buffer is removed from the cache (lines 41-43). In the proposed algorithm, the time complexity for cache replacement is O(1). The reason is that the cache, prediction buffer, and non-access buffer are implemented as a deque dictionary, so the query time has a constant time complexity.

V. EXPERIMENTS AND EVALUATION

A. EXPERIMENTAL SETTINGS

For these experiments, a total of 10 traces [20] were used; web07 and web12 are the traces that were used in [21] to compare the efficiency of the replacement policy. In addition, cscope, glimpse, postgres, and sprites are the traces that are extracted from the various applications and they were used in [7], [22], [23]. Two 2-pools, multi1, multi2, and multi3 are the synthetic traces that are obtained by running applications concurrently and they were used in [7], [22]. Table 1 provides a detailed description of the traces that are used in the experiments. The reference count refers to the number in the trace, and the coverage block numbers/deltas see that the coverage of top 1000 block number/deltas for all accesses.

Some of the aforementioned traces are classified as four types of file cache access patterns according to Choi *et al.* [23].

- Sequential Pattern: All the blocks are requested one after another and are never re-accessed.
- Looping Pattern: All the blocks are requested repeatedly after a regular interval (period).
- Temporally clustered patterns: The blocks that are requested more recently are the ones that are more likely to be requested in the near future.
- Probabilistic Pattern: Each block has a stationary reference probability, and all the blocks are requested independently according to the associated probabilities.

Assuming that a sequential pattern is a special looping pattern case, cscope, glimpse, and postgres belong to the looping pattern. In addition, the two 2-pools belong to the probabilistic pattern, sprite belongs to the temporally clustered pattern, and multi1, multi2, and multi3 belong to the mixed pattern [7]. This study trained the proposed model using the Pytorch library with an Intel (R) Xeon (R) CPU E5-2609 v4 that operates on 1.70 GHz; the system contains eight CPU cores and two GeForce GTX 1080Ti GPUs. For all the experiments, the model was trained using the Adam optimizer with a cross-entropy loss function and a learning rate of 10^{-4} for a minimum of 100 epochs and a maximum of 500 epochs. This investigation set the threshold of the model for the re-prediction as the output sequence length. The prediction buffer was added to the cache based on the LRU replacement policy and it was displayed as a model in the result graph (Figs 7–10). To evaluate the performance of the proposed model, this study used the LRU, LFU, ARC, Belady's MIN algorithm, and state-of-the-art, LeCaR as comparison targets.

B. RESULTS

This section measures the performance of the proposed method. First, after training the model based on the training data separated from the trace, the accuracy of the actual blocks and the predicted blocks in the traces are compared. Second, the cache hit rate when the 10 traces mentioned above are simulated.

1) PREDICTION ACCURACY

The goal of this study is to predict in advance the correct blocks in the order of future requests. Therefore,

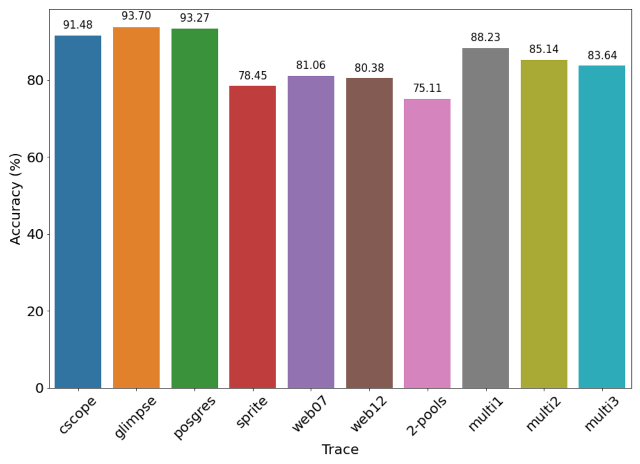


FIGURE 6. **Accuracy of prediction blocks and real blocks.**

Figure 6 means the ratio of the number of blocks correctly predicted by the model to the total blocks requested. The evaluation measured whether the block predicted in advance at that point in time was the same as the actual requested value when the cache simulation was run on a model trained by a specific trace in advance. Overall, the evaluation shows high accuracy in all traces, and 2-pools with the lowest accuracy shows 75% accuracy despite the difficulty of accurately predicting future blocks due to frequent call sequence changes.

2) LOOPING PATTERNS

The cscope (Fig. 7. (a)), glimpse (Fig. 7. (b)), and postgres (Fig. 7. (c)) have various looping patterns at different intervals. By contrast, because the LRU is based only on recency, it cannot distinguish between the blocks that are frequently requested and the blocks that are not frequently requested. Therefore, if the size of the loop set is larger than the cache size, the LRU will have the worst performance. Because the LRU policy does not use the loop period information of the looping reference, when the loop blocks with the short loop periods and the long loop period blocks come in, the short loop references can be removed from the cache before they are re-referenced [22]. In this experiment, the hit rate of the proposed model outperforms the LRU by 77%, the LFU by 2%, and the ARC by 77% in the cscope trace. When using the glimpse trace, the proposed model outperforms the LRU by 31%, the LFU by 4%, and the ARC by 21%. When considering the postgres trace, it outperforms the LRU by 29%, and the ARC by 21%. Because the proposed method predicts the block that is to be requested in the future and it knows the block that needs to be kept in the cache in advance, it can solve the weak characteristics of the LRU replacement policy that occurs when the cache size is small in the LFU-friendly workload. In glimpse trace, LeCaR shows similar behavior to LFU when the same iteration set is requested at regular intervals, it does not quickly recognize when various iteration sets exist, such as glimpse and postgres, so the proposed model shows high performance at a specific cache size.

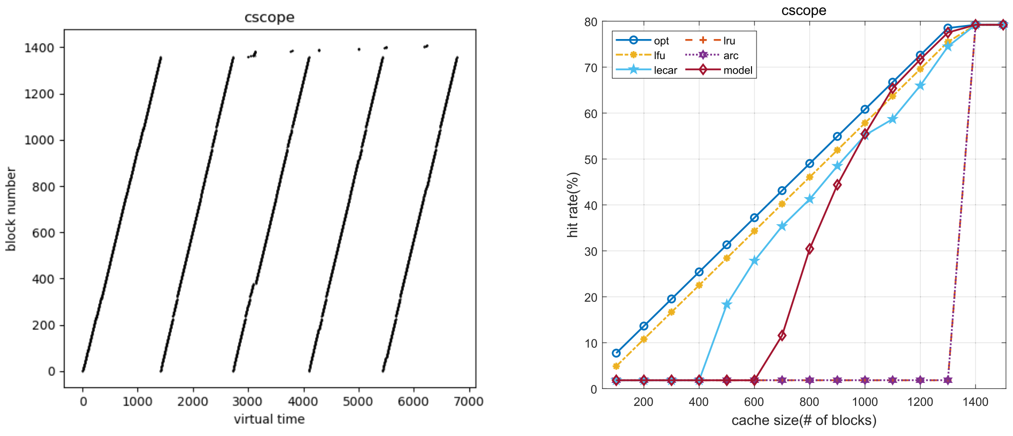
3) TEMPORALLY CLUSTERED PATTERNS

The sprite (Fig. 8. (a)) is a trace with temporal locality; web07 (Fig. 8. (b)) and web12 (Fig. 8. (c)) are traces with a long-tail distribution [21]. The sprite is LRU-friendly; therefore, if the proposed model makes a wrong decision, the hit rate may be worse than the LRU. However, as shown in Figure 6(a), the basic operation of the proposed model is based on LRU, so it brings the advantages of LRU as it is, and additionally checks future references to show a few performance improvement over LRU. In this experiment, the hit rate of the proposed model outperforms the LRU by 2%, the LFU by 65%, and the ARC by 3% when using the sprite trace. This is because the proposed method can accurately predict the block to be requested and prevent the risk that the block to be requested will be selected as a replacement target within a short time. In addition, because the working sets that frequently change in the LRU-friendly workloads can be detected in advance, the previously requested working set is prevented from being evicted from the cache.

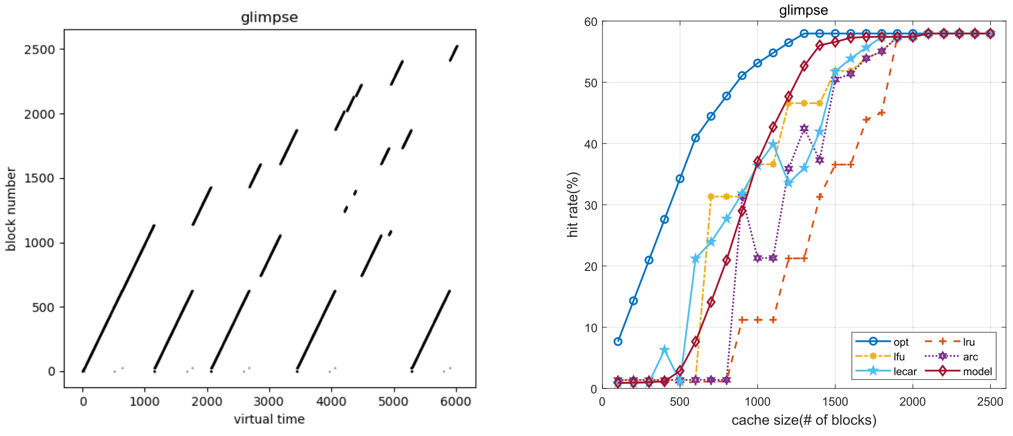
In addition, web07 and web12 show the temporal locality, sequential reference, and the other references. The LFU can distinguish the most popular data from a long-tail distribution but it cannot take advantage of the temporal and spatial locality of the user access [24]. The LRU has a higher hit rate than the LFU because the period of blocks that are frequently referred to in web07 and web12 is short and it can use temporal and spatial locality. Because the ARC manages the LRU list and the LFU list separately, locality data and popular data can be maintained even when sequential or other references come in. However, because the LRU relies only on recency, the locality data can be replaced by the LRU. In this experiment, the hit rate of the proposed model outperforms the LRU by 3%, the LFU by 24%, and it shows a hit rate equal to or slightly higher than the ARC when using the web07 and web12 traces. This is because when the data that do not have temporal locality such as the sequential reference and the other references that come in, the locality data and popular data that are highly likely to be re-requested can remain in the cache. When compared with LeCaR, the proposed model shows similar performance to slightly higher than LeCaR.

4) PROBABILISTIC PATTERNS

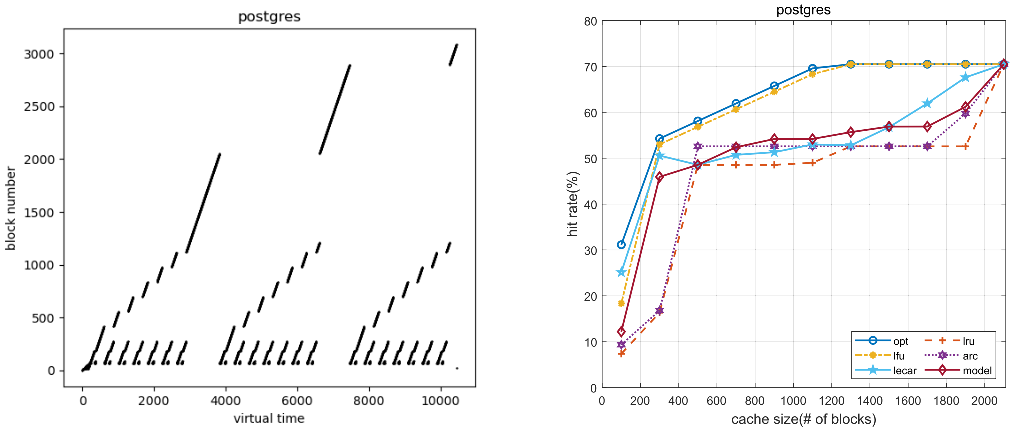
The 2-pools (Fig. 9) trace is a synthetic trace that is obtained by simulating a multi-user database application to evaluate whether the cache replacement policy can recognize request patterns over a long period of time. When considering the two 2-pools, a request with a low frequency and a request with a high frequency are requested alternately [8]. Because the LRU and LFU only consider recency and frequency, it is difficult to distinguish when the cache size is small. However, because the ARC remembers the previous request pattern by adding a ghost buffer, the LRU and LFU list can be flexibly adjusted in consideration of the current request. In this experiment, the hit rate of the proposed model outperforms the LRU



(a) cscope



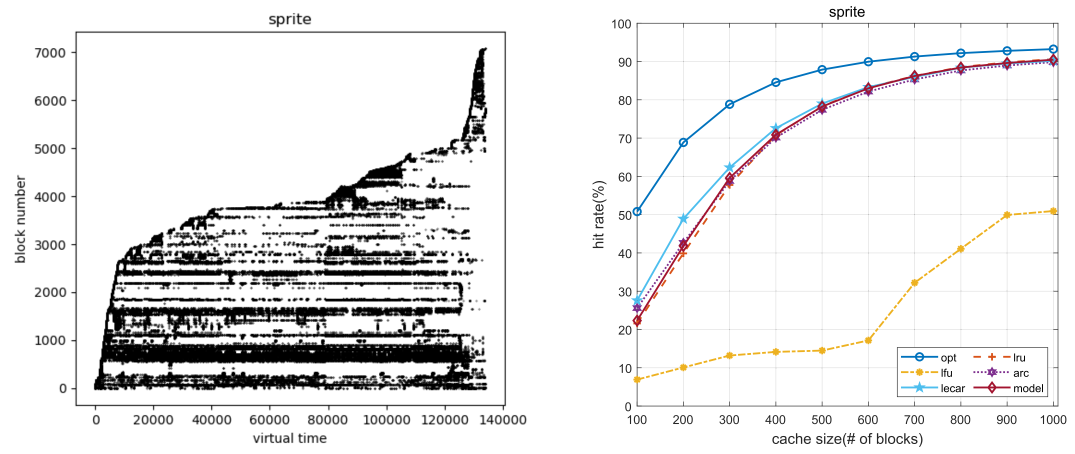
(b) glimpse



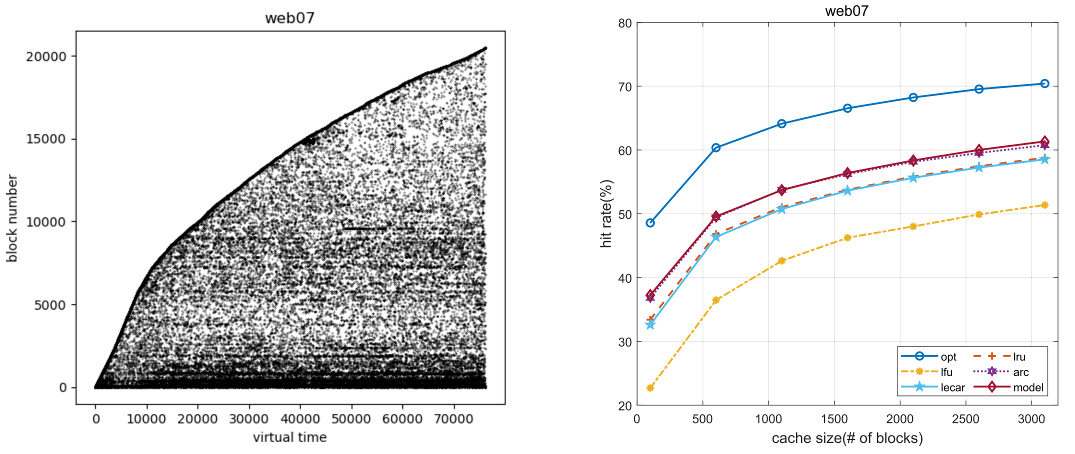
(c) postgres

FIGURE 7. Block reference and hit rate of the looping pattern workload.

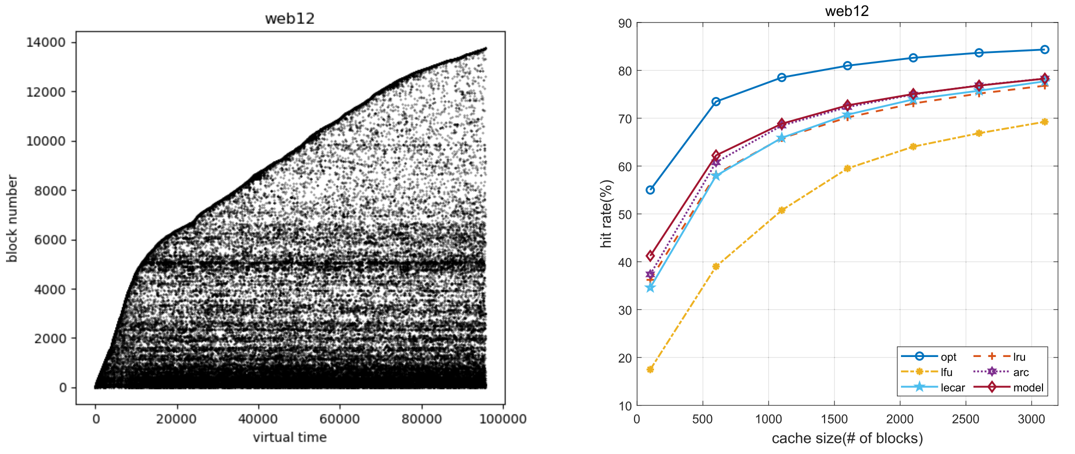
by 10%, the LFU by 13%, and the ARC by 2% when using the two 2-pools. This is because the proposed model does not consider the blocks that have been replaced in the past, but only the blocks that exist in the cache. In addition, it predicts the sequence of the future requests for a certain period of time; thus, it can adapt to future request patterns faster than the LRU that is based on past request patterns. LeCaR overall shows a higher hit rate than the proposed model. The reason is that the LRU itself cannot distinguish between low and high frequencies, so it relies on future references. However,



(a) sprite



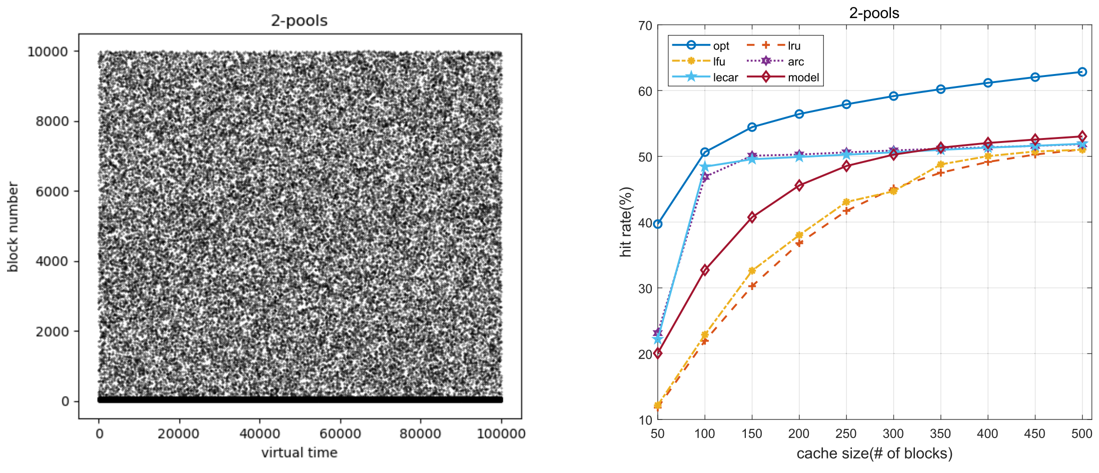
(b) web07



(c) web12

FIGURE 8. **Block reference and hit rate of the temporally pattern workload.**

if the cache size is small, since the future reference that the proposed model can memorize is limited, it is more likely to be operated based on LRU, which may have a negative effect. In contrast, as the cache size increases, at the Figure 9 can be seen that the hit ratio of the proposed model gradually increases.



(a) 2-pools

FIGURE 9. Block reference and hit rate of the probabilistic pattern workload.

5) MIXED PATTERNS

Multi1 (Fig. 10. (a)), multi2 (Fig. 10. (b)), and multi3 (Fig. 10. (c)) are traces that are obtained by executing various applications simultaneously; they have various request patterns, such as looping and temporal locality. Because a looping pattern appears for each request, it can be observed that the hit rate is improved when the cache size of the LRU can accommodate all of the looping sets. As ARC is also vulnerable in the looping patterns, the hit rate is lower than the LFU when the loop is frequent. In this experiment, the hit rate of the proposed model outperforms the LRU by 31%, the LFU by 11%, and the ARC by 22% when using the multi1 trace. When considering the multi2trace, it also outperforms the LRU by 23%, the LFU by 7%, and the ARC by 8%. Finally, when using the multi3 trace, the performance is better than the LRU by 16%, the LFU by 8%, and the ARC by 10%. This is because the proposed model can predict the loop set in advance; thus, it can prevent the removal of blocks that are close to the LRU that may be re-requested. Compared to LeCaR, the proposed model shows overall lower performance. Therefore, the LRU policy, which is the basis of the proposed model, has the characteristic of being weak in the corresponding traces when the future sequence is short enough to not be able to distinguish various features.

VI. LIMITATIONS AND FUTURE WORK

This section describes the three limitations that can be found by linking the machine learning model to predict the future block sequence with the cache replacement policy. In addition, this section describes the future directions of research.

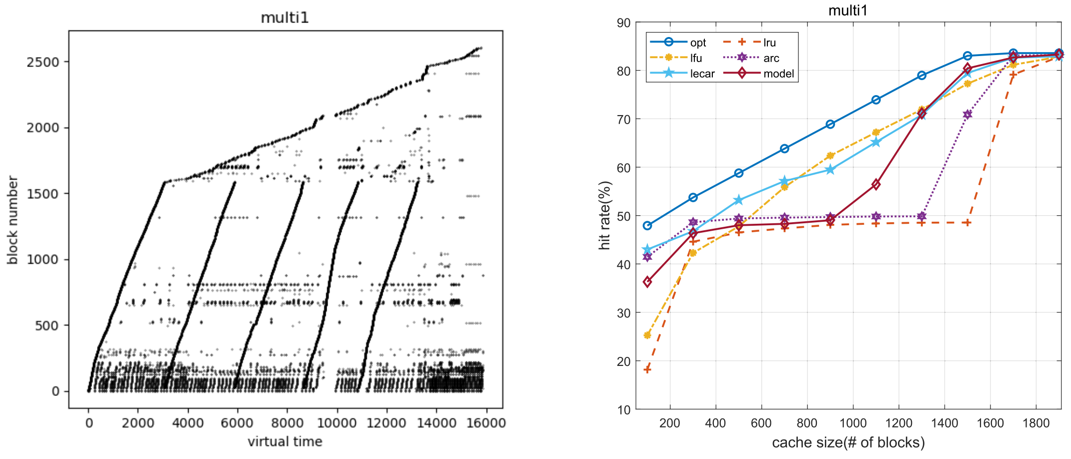
First, the model uses past requests as the inputs to predict the next request. The blocks that are predicted by the model can also be used as the input to fill the prediction buffer. If the predicted value by the model is ‘‘<unknown>’’, the previous block cannot be known even if the next block is predicted; therefore, the prediction buffer cannot be filled anymore. To solve this problem, this study proposes a new method of filling the prediction buffer when the prediction of the model is incorrect by the set value. However, because this method has to fill the prediction buffer, there is a temporal overhead. If the prediction is frequently wrong, the performance will be the same as the performance of the existing replacement policy.

Second, the model was trained for each workload so that similar I/O access patterns of the different workloads were not utilized. To demonstrate the reusability and practicality of the model, it is necessary to be able to learn common patterns of the workloads that have not been seen before and the workload that the model has learned. In addition, for high-accuracy predictions in the workloads that have never been observed, as well as common patterns, the model must be trained online. However, because the LSTM model is trained offline, it only works well on the trained data.

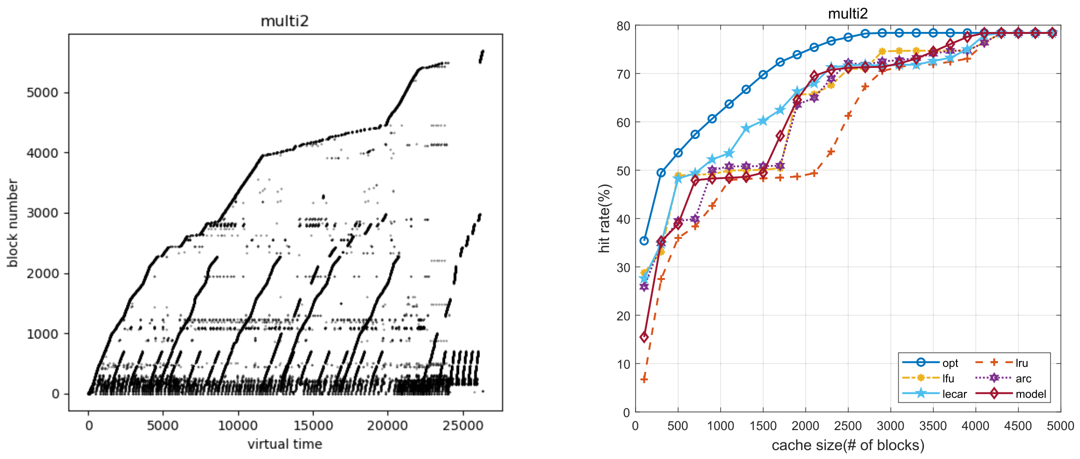
Third, the buffer size is fixed to the cache size; hence, the next request time for the block that is being currently held in the cache, such as Belady’s MIN, cannot be determined. Therefore, there is still a possibility of an inefficient replacement.

Therefore, when the prediction fails, a secondary method is required to allow the model to continue making predictions, and an online model is needed so that the LSTM that is trained in the individual workloads can learn the various workload patterns. In addition, future research is needed to improve the predictions.

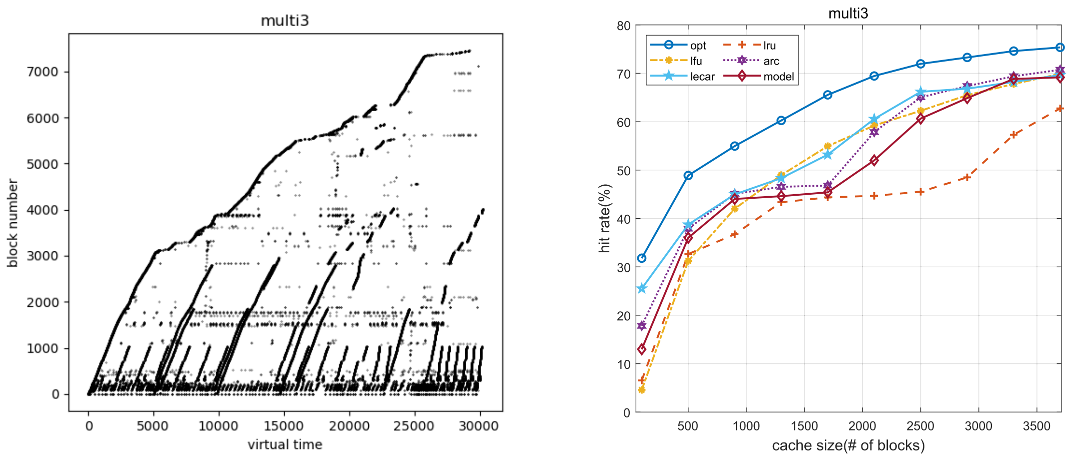
This work focuses on the ML challenges of predicting the future block reference patterns; it does not explore the practicality. To address these concerns, future studies can focus on investigating techniques such as a method of reducing the inference time of a trained model by caching the hidden layer output [25] and a method of solving the LSTM memory overhead by compressing the model [26]. Finally, when compared with the latest study, certain patterns showed higher performance, but other specific patterns showed similar or lower performance. This is a disadvantage derived from the



(a) multi1



(b) multi2



(c) multi3

FIGURE 10. Block reference and hit rate of the mixed pattern workload.

heuristic replacement policy, which is the default operation, and it is necessary to select a default cache replacement policy that can be well combined with the future sequence and replacement policy. By considering this trend, it is possible to confirm the feasibility and the direction of the deep learning-based system performance improvement.

VII. CONCLUSION

This study proposes a machine learning model called Seq2Seq to predict the future block sequences, a prediction buffer that stores the predicted value of the trained model, a re-prediction process that increases the accuracy of the prediction buffer, and a method to determine the target to be evicted based on the future block sequence in O(1) time complexity by adding the non-access buffer. When the machine learning model that interprets the input sequence pattern and predicts the future sequence is applied to the existing heuristic cache replacement policy, the proposed method outperforms the LRU by 77%, the LFU by 65%, and the ARC by 77%. Therefore, by combining the proposed method with a heuristic cache replacement policy, such as the LRU, adaptation is possible in the LRU-friendly or LFU-friendly workloads. In addition, we explore the improvement direction of the proposed method by showing slightly higher, lower, and similar performance with the state-of-the-art cache replacement policy. In conclusion, we propose an improved cache replacement policy that allows the heuristic cache replacement policy to utilize both past and future information by using future reference information. We think that the ML-based cache replacement policy for efficient caching has the potential to improve efficiency when handling various workload patterns.

REFERENCES

[1] Z. Song, D. S. Berger, K. Li, A. Shaikh, and W. Lloyd, "Learning relaxed Belady for content distribution network caching," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2020, pp. 529–544.

[2] K. Thar, N. H. Tran, T. Z. Oo, and C. S. Hong, "DeepMEC: Mobile edge caching using deep learning," *IEEE Access*, vol. 6, pp. 78260–78275, 2018.

[3] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 6237–6247.

[4] C. Chakraborttii and H. Litz, "Learning I/O access patterns to improve prefetching in SSDs," in *Proc. ICML-PKDD*, 2020, pp. 427–443.

[5] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[6] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. FAST*, vol. 3, 2003, pp. 115–130.

[7] S. Jiang and X. Zhang, "Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 31–42, Jun. 2002.

[8] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proc. FAST*, vol. 4, 2004, pp. 187–200.

[9] R. Santana, S. Lyons, R. Koller, R. Rangaswami, and J. Liu, "To ARC or Not to ARC," in *Proc. 7th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2015, pp. 1–5.

[10] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, and J. Liu, "Driving cache replacement with ML-based LeCaR," in *Proc. 10th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2018, pp. 928–936.

[11] P. Li and Y. Gu, "Learning forward reuse distance," 2020, *arXiv:2007.15859*.

[12] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 413–425.

[13] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*.

[14] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, 2000.

[15] C. Li, "DLIRS: Improving low inter-reference recency set cache replacement policy with dynamics," in *Proc. 11th ACM Int. Syst. Storage Conf.*, 2018, pp. 59–64.

[16] S. Ross and J. Andrew Bagnell, "Reinforcement and imitation learning via interactive no-regret learning," 2014, *arXiv:1406.5979*.

[17] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, and R. Rangaswami, "Learning cache replacement with CACHEUS," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, 2021, pp. 341–354.

[18] A. Farhangi, J. Bian, J. Wang, and Z. Guo, "Work-in-progress: A deep learning strategy for I/O scheduling in storage systems," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2019, pp. 568–571.

[19] W. Liu, Y. Wen, Z. Yu, and M. Yang, "Large-margin softmax loss for convolutional neural networks," in *Proc. ICML*, 2016, vol. 2, no. 3, pp. 507–516.

[20] B. Manes. (2016). *Caffeine: A High Performance Caching Library for Java 8*. Accessed: Sep. 21, 2021. [Online]. Available: https://github.com/ben-manes/caffeine

[21] *cache2k*. Accessed: Sep. 21, 2021. [Online]. Available: https://cache2k.org/benchmarks.html

[22] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A low-overhead,high-performance unified buffer management scheme that exploits sequential and looping references," in *Proc. 4th Conf. Symp. Operating Syst. Design Implement.*, vol. 4, 2000, pp. 119–134.

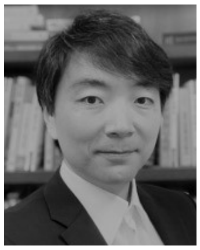
[23] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "An implementation study of a detection-based adaptive block replacement scheme," in *Proc. USENIX Annu. Tech. Conf., Gen. Track*, 1999, pp. 239–252.

[24] G. Karakostas and D. N. Serpanos, "Exploitation of different types of locality for web caches," in *Proc. 7th Int. Symp. Comput. Commun. (ISCC)*, 2002, pp. 207–212.

[25] A. Balasubramanian, A. Kumar, Y. Liu, H. Cao, S. Venkataraman, and A. Akella, "Accelerating deep learning inference via learned caches," 2021, *arXiv:2101.07344*.

[26] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2018, pp. 11–20.

HYEJEONG CHOI received the B.S. degree in computer engineering from Keimyung University, in 2020, where she is currently pursuing the master's degree in computer science. Her research interests include AI-based operating system to improve high-performance and distributed learning for user convenience.



SEJIN PARK received the B.S. degree in software engineering from the Kumoh National University of Technology, in 2007, and the Ph.D. degree in computer science from POSTECH, in 2016. He was a Software Engineer with the SK Telecom Research Center, from 2016 to 2018. He has been an Assistant Professor with the Department of Computer Engineering, Keimyung University, since 2018. His research interests include AI-assisted high-performance operating systems, AI-assisted music composition technologies, and blockchain technologies.