TE Model: A Transformer Encoder Only Based Model for Prefetch and Replacement in Solid-state Drivers

Jianming Ge
School of Software
Beihang University
Beijing, China
gejianming@buaa.edu.cn

Abstract—Memory technology offers access latency that is three orders of magnitude lower than that of solid-state drives (SSDs); prefetching and replacement, which rely on predicting future access sequences, are commonly used methods to bridge this speed gap, but previous research typically treats prefetching and replacement as independent tasks: the Stride prefetcher is only suitable for memory with strong locality, the LSTM prefetcher supports only short input sequences, and replacement strategies mostly rely on heuristic rules; in this work, we formulate prefetching as a time-series prediction classification problem, utilizing long historical sequences as features to predict future sequences for guiding prefetching and replacement decisions, propose the TE model, a Transformer encoder-only framework dedicated to learning and predicting complex access patterns in SSDs, which can forecast multi-step future request sequences and derive optimal prefetching and replacement strategies by integrating current cache blocks with predicted future access blocks; to the best of our knowledge, this is the first caching management framework for SSDs that unifies prefetching and replacement under a holistic design, and experimental results show that in most scenarios, the TE-base framework—which only predicts future sequences to guide prefetching—outperforms three baseline prefetching models, while the TE-advance model, which uses future sequence predictions to guide both prefetching and replacement, achieves even better performance, demonstrating that the proposed model effectively learns complex access patterns and that an integrated perspective of prefetching and replacement holds great promise for future research.

Keywords—deep learning, Transformer, data prefetching, replacement, LBA(logical block address), SSDs

I. INTRODUCTION

In the era of big data, the demand for storage space continues to grow. The SSDs have become the main storage medium due to their decreasing cost and increasing access speed. However, there remains an access latency gap between SSDs and memory, reaching up to three orders of magnitude [1]. As shown in Figure 1, the response time of SSDs is composed of address translation latency and data movement latency [2]. To prevent SSDs from becoming a performance bottleneck, the cache area (DRAM or SRAM) inside SSDs acts as data cache and hides the access delay of NAND flash. Prefetching and replacement technology are the most common methods to improve cache utilization [3][4][5].

Accurately predicting future I/O access at the storage level faces several main challenges: First, SSD capacities reach 16TB, resulting in an enormous address space. Second, access patterns are highly complex due to the interleaved access of multiple users and applications. Third, Storage-level access may retrieve data of different sizes each time; prefetching excessive data wastes system resources, while prefetching insufficient data fails to meet system requirements [4]. These challenges hinder the development of prefetcher. The stride prefetcher [7] performs well only in the storage with strong locality. The LSTM-based prefetcher [8][9][10] is not suitable for long-term dependence. G&L[11], a Transformer Decoder-only frame with a mapping table of LBA and I/O size, requires cyclic execution for multi-step predictions, and its mapping table lacks generalization.

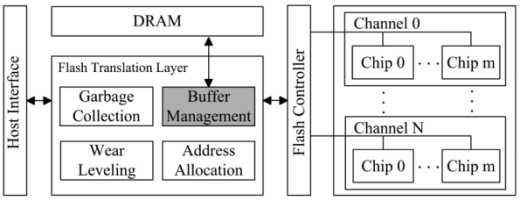

Figure 1 The internal architectural overview in SSDs

Beyond accurate prefetching, it is critical to determine which cache block to replace. Maximizing cache utilization requires an understanding the reuse distance of prefetched data, which is challenging to quantify. This work proposes a Transformer Encoder-Only framework (TE) equipped with a self-attention mechanism, which can capture contextual information within sequences, learn SSD access patterns, and predict future multi-step access sequences—including the LBA, I/O size, and reuse distance of cache blocks. By leveraging the aligned prefetch sequence to guide replacement decisions, the TE framework approximates the OPT algorithm [12] and improves cache utilization.

The key contributions of this work are as follows: 1) A Transformer-based Encoder module designed specifically for predicting SSD access patterns; 2) A comprehensive algorithm for integrated cache prefetching and replacement; 3) A custom simulator developed to evaluate cache utilization.

We conducted extensive experiments on six real-world datasets. Empirical results demonstrate that the TE framework consistently outperforms three baseline models. Specifically, its prediction accuracy improved by an average of 74.34%, 12.80%, and 12.80%, while coverage increased by 179.83%, 20.92%, and 21.86%, respectively. In terms of I/O size prediction, the TE-base framework outperformed LSTM and G&L by 11.05% and 8.54%, respectively. The advanced version, TE-advance, achieves even better performance by incorporating the replacement factor into its strategy.

The remainder of this paper is structured as follows: Section II reviews related work; Section III discusses observations and motivation; Section IV details the design and implementation of the model; Section V presents

experiments and evaluations; and finally, Section VI concludes the paper.

II. RELATED WORK

A. Prefetcher

The stride prefetcher [7] is the most traditional prefetcher, which employs a reference prediction table to store the most recently accessed LBAs and their associated strides for computing subsequent LBAs. While it can capture constant strides in sequential access patterns, it struggles to detect variable strides in irregular access patterns and performs poorly in storage with weak locality.

To learn more complex access patterns, deep learning-based approaches have been proposed, which leverage LBA deltas as input [8][9][10][11]. For example, Delta-LSTM [9] addresses the challenges of large and sparse LBA spaces by jointly learning top-K LBA deltas and I/O size features. In the top-K classes (e.g., K = 1000), the search space is constrained, thereby alleviating the class explosion problem. However, these models fail to capture the relationships between LBA deltas, which limits their ability to model more complex patterns, such as continuous LBA accesses that span multiple pages. Deep prefetchers [9] transform LBA sequences into LBA increments and then use a word2vec model and LSTM architecture to capture hidden features in the input sequences. Nevertheless, they are inefficient on large-scale trace datasets. G&L [12], a generative pre-trained (GPT)-based prefetcher, integrates a Logical Block Address (LBA)-I/O size table. It can only perform one-step prefetching at a time, and while multi-step prefetching is implemented iteratively, it faces scalability challenges due to its reliance on table lookups.

B. Replacement

Hawkeye [13] is a binary classification model based on reuse distance, which categorizes access sequences into cache-friendly and cache-averse groups. Glider [14] uses LSTM for offline training and an SVM for online decision-making, enabling dynamic cache management. Mockingjay [15] extends Hawkeye by introducing multi-label reuse distances, but its strategies rely on fixed assumptions and static designs, which may limit its adaptability. Care [16] incorporates both data locality and concurrent access patterns to enhance replacement decisions, aiming to balance cache efficiency and real-time access demands.

C. Summary

The limitations of the aforementioned related works can be summarized as follows:

1) In the field of SSD prefetching, the multi-step prediction of long-term dependencies in time series remains underexplored;

2) There is a lack of comprehensiveness, as these works examine cache prefetching and replacement in isolation, ignoring the potential benefits of a joint optimization strategy.

III. OBSERVATION AND MOTICATION

A. SSDs prefetching challenges

As mentioned in Section I, data prefetching at the storage level presents several critical challenges: First, applications are frequently accessed by multiple users simultaneously, each executing independent tasks, which results in a complex mix of sequential and random I/O requests. In existing operating systems, I/O access involves traversing a deeply layered software stack, which often transforms predictable application-level accesses into seemingly random access patterns at the SSD level. Second, most file-related locality is absorbed by the higher layers of the memory hierarchy, including CPU caches and main memory. Additionally, sequential access is not the primary pattern for disk access, such as in convolutional visualization and video playback. In these scenarios, strategies that explore spatial locality are not very effective at predicting access patterns. Finally, unlike CPU-level caches, the I/O size of SSDs is not fixed and must also be considered during prefetching. As a result, most requests received by storage devices exhibit poor spatial locality. Although some learning-based methods have been designed to capture non-sequential (or random) access patterns, these methods struggle to predict unseen access patterns. This is because the patterns of accessed LBAs are rare with few repetitions, and the potential values of LBAs are vast, which increases the complexity of learning algorithms.

To validate these challenges, we conducted a series of analyses on the MSR Cambridge Trace [17]. Figure 2(a) presents a snapshot of the trace, where the x-axis represents the access order parsed from read requests, and the y-axis depicts the logical block number (LBA) accessed by each request. Our analysis focuses on read requests because when the requested data is not present in the data buffer, read requests directly impact the device's service time. In contrast, write requests typically store data in the device buffer, minimizing their immediate impact on service time (or response time). The observed trend of LBA accesses spans up to 14GB, with access patterns frequently transitioning between sequential and random behavior. This variability highlights the difficulty of directly predicting LBA access patterns. Figure 2(b) illustrates the corresponding sizes of read requests, which also exhibit continuous fluctuations. Accurate prediction of read request sizes is critical for improving prefetching efficiency and overall system performance.

Further analysis of the MSR Cambridge Trace I/O data reveals that over 40% of LBA read requests are accessed only once during the trace collection period. The I/O sizes vary widely, ranging from 512 bytes to several megabytes. This combination of a vast number of possible LBAs, nearly random access sequences, and dynamically changing I/O sizes poses significant challenges for developing effective models for learning-based prefetchers.

One of the challenges in predicting the LBAs for subsequent requests is the vast address space. In this work, we adopt the same approach as previous studies [9][10] for handling LBAs, applying the first-order difference of two consecutive LBAs, referred to as Delta

$$\mathrm{Delta}_t = \mathrm{LBA}_{t+1} - \mathrm{LBA}_t \quad (1)$$

By monitoring the changes between consecutive LBA requests via Equation (1), new features are extracted to identify similar access patterns. LBA and LBA denote the LBAs accessed at times t+1 and t, respectively. Utilizing Delta to learn the dependencies within a sequence is more effective than learning the dependencies of the sequence itself. For instance, LBA Sequence i [1, 3, 5, 7] and LBA Sequence ii [2, 4, 6, 8] share the same Delta sequence [2, 2, 2]. If the LBA sequence itself is used for learning, the access patterns

learned from Sequence i cannot be applied to Sequence ii, as the two sequences are completely distinct from the model's perspective. This new Delta feature not only helps improve learning accuracy but also reduces the complexity of the learning process.

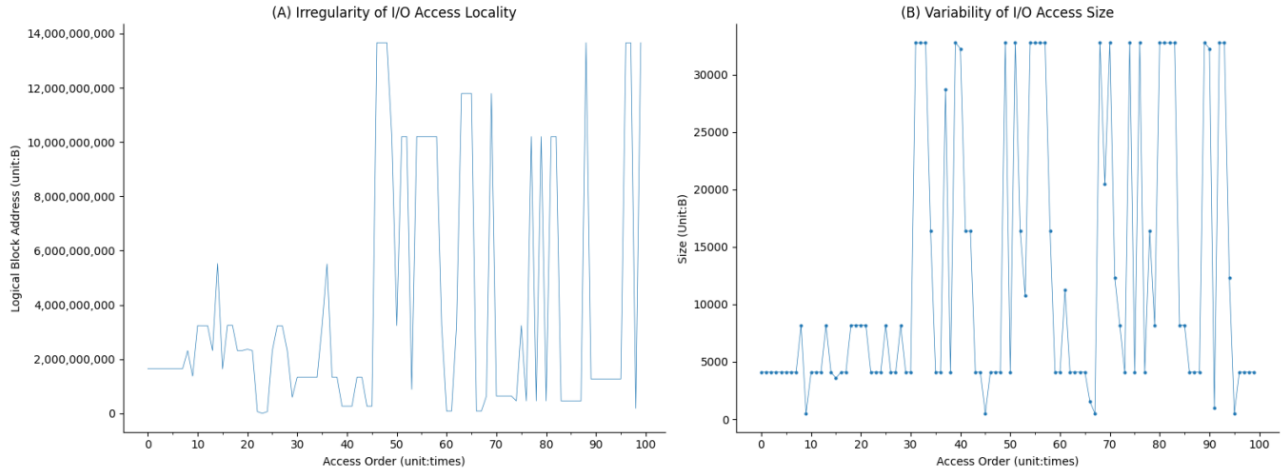

Figure 2 (a) Irregularity of I/O Access Locality and (b) Variability of I/O Access Size. For instance, the 5th request will access LBA at round 2,000,000,000 with 4096B

B. *An Integrated Perspective on Prefetching and Replacement*

Consider a program with an access pattern of 'ABCA'. Assume the system has two cache blocks, and transferring data from the SSD flash to the DRAM cache takes T time units, while accessing the cache takes only 1 time unit. Prefetching starts one access unit in advance. As illustrated in Figure 3, without prefetching and using the OPT replacement algorithm, the total time consumed is T + 3 time units. In contrast, Figure 4 depicts the result of aggressive prefetching (prefetching at all times) combined with the OPT replacement algorithm, where the total time spent increases to 2T + 2 time units.

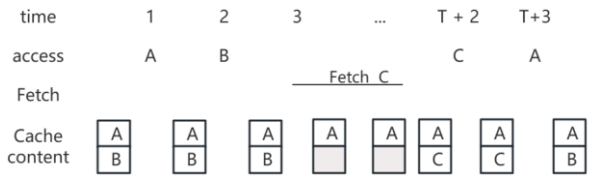

Figure 3 Cache model without prefetching, using the optimal replacement strategy. Accesses to A and B are hits; access to C causes a stall of T time units, and the final access to A is a hit, resulting in a total consumption of T + 3 time units.

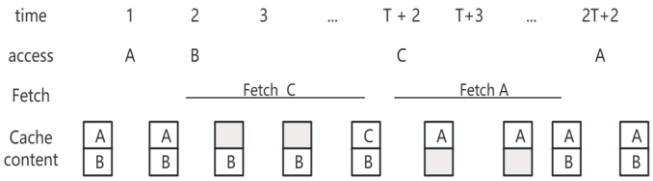

Figure 4 Cache model with continuous prefetching. Transferring data from flash storage to the cache takes T time units, and accessing the cache takes 1 time unit, resulting in a total access time of 2T + 2 time units. First, access to A is a hit; then access to B is a hit, while C is prefetched and A is replaced. The prefetching of C causes a stall of T-1 time units. When access to C occurs, prefetching of A is initiated, and B is replaced. The final access is to A, during which a stall of T-1 time units occurs.

This example demonstrates that aggressive prefetching is not always beneficial. In the no-prefetching scenario, only one block is fetched, whereas the aggressive prefetching strategy fetches two blocks. The cost of the additional fetch outweighs the benefits of hiding access latency. However, if the access stream changes from 'ABCA' to 'ABCB', aggressive prefetching outperforms the no-prefetch strategy, as it successfully reduces fetch delays. This example highlights the double-edged nature of aggressive prefetching: while it can effectively hide fetch delays, it may also lead to an increased number of fetches, potentially offsetting its benefits.

C. *Summary*

The above analyses highlight that the unique characteristics of SSDs inherently make accurate prefetching challenging. However, addressing this challenge becomes even more difficult when prefetching and replacement are treated as separate, independent tasks—a common practice in most existing approaches. Therefore, a more effective solution requires a model that not only learns and adapts to SSD access patterns but also performs multi-step predictions to guide both prefetching and replacement decisions within a unified framework. By leveraging predicted access sequences and reuse distances, such a model can achieve holistic optimization and mitigate the limitations of current strategies.

IV. Design and Implementation

A. *Data Preprocess*

We utilize trace files from six datasets, which are derived from two publicly available sources: Microsoft Research Cambridge [17] and Microsoft SNIA [18], as summarized in Table 1. For data preprocessing, we compute the first-order difference between consecutive access moments as described in Equation (1). This transformation converts the problem of predicting the next accessed LBA into the task of predicting Delta. As described in equation (2), the prediction function f, learned using a neural network, takes a sequence of Delta values with length l as input, where θ denotes the set of parameters for f.

To reduce the address space, we focus on the 1000 most frequent Delta values. All other increments are grouped into

a single class (class 1001), which represents the "no prefetch" operation. In this scenario, if the model predicts a Delta value outside the top 1000, no data is prefetched. This approach effectively reduces the address space to 1001 classes, simplifying the prediction task.

As shown in Table 1, the coverage of the top 1000 Delta values consistently exceeds that of direct memory addresses (offsets). This higher coverage makes Delta values a more suitable feature for model training, enabling the prefetcher to achieve higher accuracy and efficiency.

Table 1 LBA Dataset Description

Trace Source	Dataset Name	Num obs	Coverage Offset (%)	Coverage LBA Delta (%)	Unique Number I/O size	Coverage page count（%）
VDI	2016021614-LUN4	1198701	1.78	38.69	257	85.55
VDI	2016021618-LUN1	316288	5.84	60.18	257	92.05
VDI	2016021620-LUN4	1080637	2.01	59.09	257	93.97
MSR	CAMRESISAA02-lvm1	52451732	16.85	59.40	128	98.92
MSR	CAM-01-SRV-lvm1	41426266	1.46	87.00	128	98.73
MSR	CAM-02-SRV-lvm3	2128303	0.88	94.33	128	99.54

$$\text{Deltat} = f(\text{Deltat-l}, \text{Deltat-l}+1, \ldots, \text{Deltat-1}, \theta) \quad (2)$$

The size of read requests ranges from 512B to several hundred KB, covering more than 200 distinct I/O size values. To reduce the complexity of the search space for I/O sizes, we adopt page alignment techniques, rounding each size up to the nearest page boundary (4KB), which is described in equation (3). We focus on the 8 most frequently occurring page counts, which collectively account for 85% of the observed page sizes. As a result, I/O sizes are categorized into 9 groups, where the 9th category represents the default prefetch of a single page. This mapping significantly reduces the number of unique I/O size values, and in the worst-case scenario, only one additional page of data may be prefetched.

$$\text{PageCount} = \text{ceil}(\text{I/O size}/4\text{KB}) \quad (3)$$

Additionally, reuse distance and usage frequency are incorporated as features in the model. Frequency refers to the number of requests made to a specific page, quantifies the access heat of the page, and helps the model identify high-value pages (pages with high access frequency are more valuable for prefetching);while reuse distance is defined as the number of pages requested between two consecutive accesses to the same page, quantifies the access correlation and cache friendliness of the page (pages with near reuse distance are more likely to be accessed again. To simplify the representation, reuse distance is categorized into three classes: near, medium, and far. The "near, medium, and far" classification simplifies feature representation and helps the model accurately judge the cache priority of pages. For example, assuming a cache size of 10, a reuse distance in the range [0, 5) is classified as near, [5, 10) as medium, and [10, +∞) as far.

Algorithm 1 presents the **LBA feature preprocessing procedure**, which transforms raw LBA sequences into training samples with structured input features and corresponding prediction targets. Given the raw LBA trace sequence T, the preprocessing takes three key parameters: **H** refers to the length of the history sequence (input to the model); **F** refers to the length of the future sequence (to be predicted); **L=H+F** refers to the total length of each sliding window. The algorithm iterates over T using a sliding window of size L. At each step i, it extracts a window T[i : I + L], splits it into a history segment T[i: I + H] and a future segment T[i + H : i + H + F], and then maps these segments to the input feature matrix X and target matrix y, respectively.

During data preprocessing, raw LBA sequences are transformed into feature sequences with an arbitrary history length m and a feature length n. The extracted features include LBA, Delta, Size, Frequency, and Reuse Distance, providing a comprehensive representation of the access patterns for model training.

Algorithm 1: LBA Feature Preprocessor

Input: T: LBA Sequence; L: Window size; H: History step; F: Future step

Output: X, y

```
/* X is a two-dimensional list, where each item contains
   multiple elements of data (Delta-LBA, SIZE, LBA,
   Reuse-Distance etc.); y is a two-dimensional list, where
   each item contains multiple elements of data (Delta-LBA,
   SIZE, etc.); L is the sliding window size, where L = H +
   F.                                                    */
begin
    i ← 0 ;
    while i + L < T.length() do
        temp ← T[i : i + L];
        i ← i + k;
        X[i] ← T[i : i + H];
        y[i] ← T[i + H : i + H + L];
    end while
    return X, y
end
```

B. TE Model

We propose the TE prefetcher, a unified prefetching and replacement framework built upon a Transformer Encoder prediction module. As illustrated in Figure 5, its workflow consists of three sequential steps: data preprocessing, model prediction, and cache prefetching with replacement.

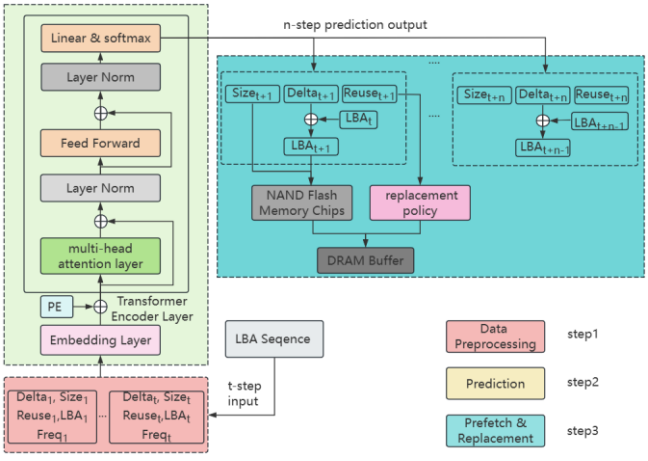

Figure 5 Architecture of our TE-Prefetcher & Replacement

In the model prediction step, the pipeline begins by applying word embeddings to represent discrete features (Delta, Size, and Reuse Distance), while continuous features (LBA and Frequency) are standardized. Treating feature

values as discrete IDs, word embeddings enable these inputs to be fed into the deep model, effectively addressing data sparsity. Each input is mapped to a new distributed representation, encoded as a t-dimensional vector that captures spatial locality information (i.e., relationships among neighboring inputs). A compact embedding matrix of size k is used for this encoding, where k determines the vector length training. To balance this trade-off, embedding vectors are typically set to several hundred dimensions. (or compression factor). Larger k values capture richer semantic relationships, while smaller values accelerate model.

In the TE model, the embedding dimension k is set to 256 for Delta. For page-aligned I/O sizes (9 categories), a 46-dimensional embedding is used, and for Reuse Distance (3 categories), a 16-dimensional embedding is applied. These embeddings—Emb-Delta, Emb-Size, and Emb-Reuse Distance—are concatenated with the normalized LBA and Frequency features, forming a 320-dimensional feature vector, which is subsequently augmented with positional encoding.

This distributed representation offers two key benefits to the TE prefetcher: (1) it reduces the dimensionality of raw inputs into compact k-sized vectors, and (2) it provides a more expressive representation by capturing contextual similarity and semantic relationships within the sequence data. This design effectively compresses the search space and lowers the overall learning complexity.

The TE model is designed to predict future sequences of length n from historical feature sequences of length t. The self-attention mechanism is particularly effective for sequential data processing, as it enables each element in the input sequence to interact with and contextualize against all other elements. This allows the model to build a comprehensive representation of the entire sequence, focusing on relationships between elements at different positions and effectively capturing complex dependencies. Unlike recurrent neural networks (RNNs), self-attention processes all sequence elements in parallel, significantly improving computational efficiency. Moreover, it excels at modeling long-range dependencies, making it highly suitable for learning patterns in sequential traces.

Following feature learning, the next stage involves classifying key attributes such as Delta, LBA, Size, and Reuse Distance. The output from the Transformer Encoder is passed through affine transformations in a fully connected (FC) layer, which maps and aggregates the learned representations to generate predictions, as shown in Equation (4):

$$\theta = W^T h_t \tag{4}$$

where W denotes the FC layer weights and h_t is the Transformer Encoder output. Notably, the model employs three independent classification heads, with the number of neurons in each FC layer corresponding to the total number of unique labels for Delta, Size, and Reuse Distance, respectively. The output of Equation (4) is then fed into a softmax function, which converts the raw scores into probability distributions over all possible prediction targets. For example, the probability of predicting class j given the model output θ_i is computed as Equation (5):

$$\text{loss}(y, \hat{y}) = -\sum_{i=0}^{M}\sum_{j=0}^{N}(y_{i,j}\, log(\hat{y}_{i,j})) + L2 \tag{5}$$

The output represents the normalized probability P(y=j|θ) where θ denotes the previous hidden state values. The candidate with the highest probability is ultimately selected as the output. Since the predicted Delta values are not the final target of the prefetcher, the corresponding prefetch LBA is derived using Equation (3). Notably, the TE model supports flexible multi-step LBA sequence prediction without modifying the model architecture, which distinguishes it from prior works [10,11] that are restricted to single-step outputs. To generate multi-step results, these prior methods iteratively feed their own single-step predictions back as input, replacing the oldest entry in the sequence until the desired number of future values is obtained.

The training process involves computing the loss function and updating model parameters over multiple epochs. TE employs categorical cross-entropy loss, which is widely used for multi-class classification tasks. Given the ground-truth

The training of the neural network involves calculating the loss function and updating parameters through multiple passes over the network. Among the available alternatives, TE utilizes the categorical cross-entropy loss, which is commonly used for multi-class classification problems. Given the true value (y) of a sequence and the predicted value ($\hat{y}$) , the categorical cross-entropy loss function calculates the prediction error generated by the model after each batch of training as formula(6):

$$\text{loss}(y, \hat{y}) = -\sum_{i=0}^{M}\sum_{j=0}^{N}(y_{i,j}\, log(\hat{y}_{i,j})) + L2 \tag{6}$$

Where $\hat{y}_{i,j}$ is the normalized probability of the model output, M is the number of unique Delta values in the training set, N is the batch size, and L2 denotes the regularization term that penalizes large weights to reduce model complexity. Here, $y_{i,j}$ is a binary indicator that equals 1 if class i is the true label for sample j, and 0 otherwise. The loss value reflects the discrepancy between predictions and ground truth: a larger value indicates greater prediction error. Training proceeds by backpropagating this error through the network using gradient-based optimization, iteratively adjusting parameters to minimize the loss function. Specifically, we use the Adam optimizer, which enables adaptive learning rates for each parameter and supports efficient training of deep models. To determine the number of pages to prefetch for each I/O request, the I/O size is aligned to the 4KB page boundary by the ceiling function

C. *Cache hit & replacement*

When a cache miss occurs, the prediction workflow is triggered. The TE model employs three classification heads: Delta LBA, I/O Size, and Reuse Distance. After resolving the target LBA, the corresponding data is prefetched from flash storage, with the number of pages determined by the I/O size, as defined in Equation (3). Before inserting new data into the cache, the system executes two core algorithms to manage buffer operations and replacement decisions.

Algorithm 2 calculates the hit size of the current request by checking how many pages are already resident in the buffer. Given the prefetching address, I/O size, and buffer state, it converts the LBA into page offsets and iterates over the requested range to count hits, returning the number of valid pages already cached.

Algorithm 2 Get_Cache

Input: *LBA*: Prefetching address; *I/O_size*: Prefetching Data; *Size*: The maximum buffer capacity; *Buf*: Buffer;
Output: *Buf*: Hit_size;

```
initialization;
Count ← 0 ;
Hit_size ← 0 ;
PageStart ← ⌈LBA/4KB⌉ ;
PageEnd ← PageStart + I/O_size ;
for CurPage ← PageStart to PageEnd do
    if CurPage ∈ Buf then
        Count ← Count + 1 ;
    end if
end for
Hit_size ← min(I/O_size, Count) ;
return Hit_size
```

Subsequently, Algorithm 3 manages the replacement policy to maintain the buffer. It first identifies the block with the longest reuse distance, then replaces it with the new data. The replacement order prioritizes blocks with the furthest reuse distance first; when multiple blocks fall into the same category, and given that most cache blocks are accessed only once, the FIFO (First-In, First-Out) policy is applied. For non-prefetched requests without reuse distance labels, a default category of mid-distance is assigned.

Algorithm 3 Set_Cache

Data: current_block, buffer
Result: updated buffer
Input : current_block: The block to be inserted into the buffer
buffer: The cache buffer containing existing blocks
Output: The updated buffer with the least recently used block replaced by current_block

```
FindLongestReuseDistanceBlock(buffer);
lrdb ← FindLongestReuseDistanceBlock(buffer);
// Find the block with the Longest reuse distance in the buffer;
ReplaceBlock(lrdb, current_block, buffer);
// Replace the Longest reuse distance block with the current block;
```

V. PERFORMANCE EVALUATION

A. Evaluation Metrics

The prefetching algorithm is evaluated using two primary metrics: **accuracy** and **coverage**, as defined in Equations (7) and (8)

$$\text{Accuracy} = \frac{prefetch\_hits}{total\_prefetch} \tag{7}$$

$$Coverage = \frac{prefetch\_hits}{misses\_without\_prefetching} \tag{8}$$

Following the methodology in [19], we developed an SSD simulator with a dedicated replacement policy to emulate a real-world storage system equipped with a 32 MB DRAM buffer. The buffer operates with a 4 KB block size, matching the native page granularity of NAND flash memory. Prefetched data is loaded into the DRAM buffer, and the prefetching algorithm considers not only which LBAs to fetch but also the timing of prefetch operations and the buffer replacement strategy. For a fair comparison, all experiments adopt a miss-triggered prefetch policy. When prefetching is disabled, the LRU policy is used to evict blocks when the buffer is full. When prefetching is enabled, replacement prioritizes blocks with the longest reuse distance and the oldest access timestamp, while prefetched data is placed at the head of the buffer.

We compare the proposed TE prefetcher against three baselines:

Stride Prefetcher: This assumes a regular access pattern with constant stride. Given consecutive addresses LBAt−1, LBA_t, LBA_{t+1}, it assumes $Delta_{t-1}$ = $Delta_t$, and initiates prefetching LBA_t + $Delta_t$, followed by LBA_{t+1} + 2 * $Delta_t$, with a prefetch length of 2.

LSTM-based Prefetcher: This model processes Delta values as input features. The I/O size is rounded to the nearest class of 2^n, and both Delta and the quantized I/O size are fed into an LSTM network for prediction.

G&L Prefetcher: This approach leverages a Transformer decoder network combined with an LBA-Size lookup table for prefetch decisions.

B. Experiment Parameter Settings

We employ a 4-layer, 8-head self-attention module in the TE model. Dropout is disabled due to the high complexity of the input features. The loss weights for the three classification heads are set to 1: 0.8: 0.5. The prototype of the proposed model is implemented using Python and the PyTorch 2.2 deep learning framework.

In this work, the model is trained on an Ubuntu 22.04 system equipped with an Intel(R) Core™ i9-14900K CPU (32 cores) and four NVIDIA GeForce RTX 4090 Ti GPUs.

C. Experimental Results Evaluation

Note that for each trace, the Stride prefetcher monitors the last three I/O accesses. If the stride (i.e., the difference between consecutive addresses) remains consistent, the prefetcher identifies the pattern and initiates prefetching for the subsequent accesses.

In the experiments, all baseline algorithms and **TE-base** variant employed the LRU replacement policy. In contrast, the **TE-advanced** framework, which leverages reuse-distance labels, implements a FIFO-based replacement strategy that prioritizes evicting blocks with the farthest reuse distance first. The experimental results for I/O size accuracy, as well as LBA accuracy and coverage, are illustrated in Figures 6, 7, and 8, respectively.

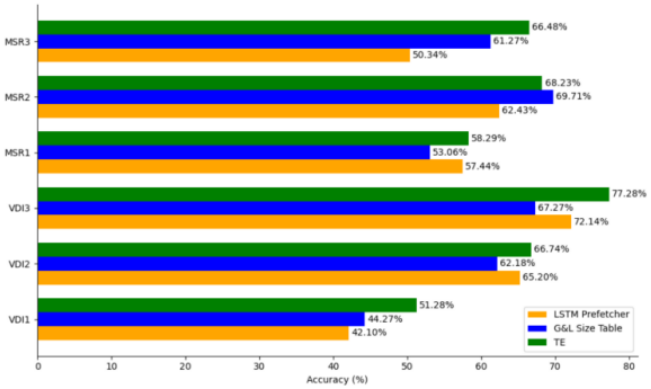

Figure 6 I/O size accuracy

The results clearly demonstrate that **TE-base** outperforms the baseline methods across the majority of workloads. Specifically, the TE framework achieves an average improvement of **74.34%**, **12.80%**, and **12.80%** in prediction accuracy compared to the three baselines, while coverage is boosted by an average of **179.83%**, **20.92%**, and **21.86%**. In terms of I/O size prediction, TE surpasses the LSTM and G&L prefetchers by **11.05%** and **8.54%**, respectively. The advanced iteration, **TE-advanced**, further elevates

performance by integrating the reuse-distance-based replacement strategy into its prefetching logic.

VI. Conclusion

To improve SSD prefetching efficiency, this work proposes the TE prefetcher, a unified framework predicting future access addresses and I/O sizes. Using a Transformer encoder with self-attention and LBA-I/O size alignment, it captures LBA patterns and classifies request lengths. TE-advanced further integrates predicted reuse distances to guide cache eviction, rather than separating prefetching and replacement. Experiments show TE outperforms baselines in both predictions, and TE-advanced achieves better performance, verifying the effectiveness of joint optimization.

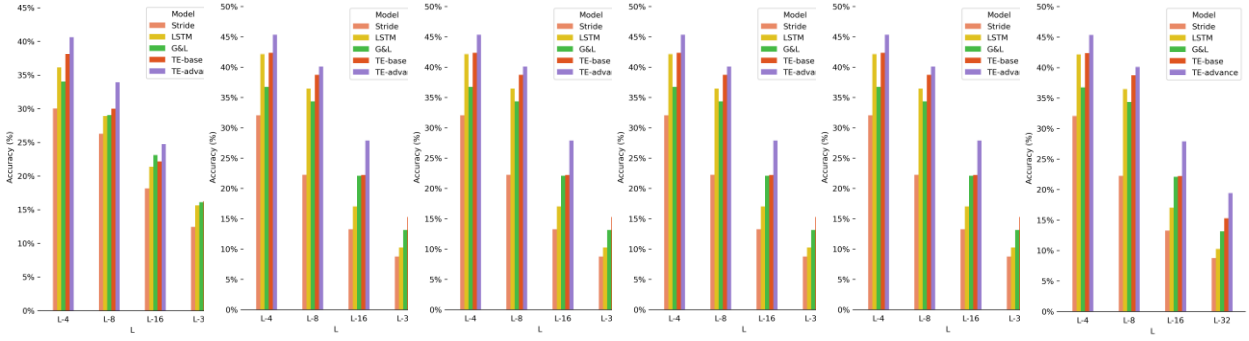

Figure 7 Accuracy

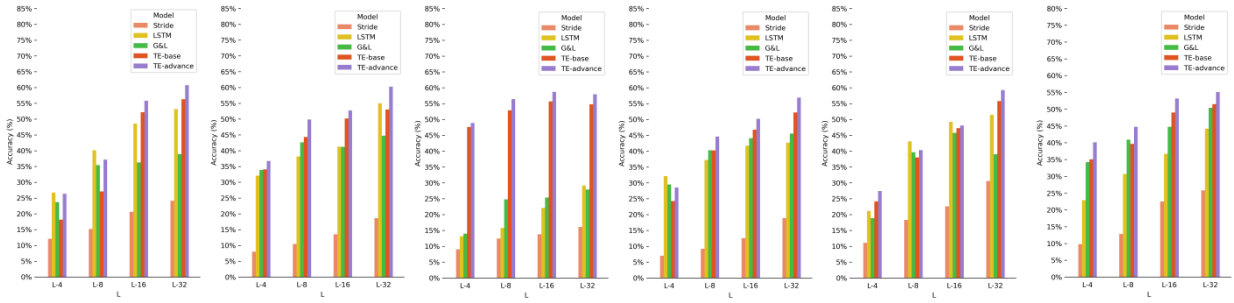

Figure 8 Coverage

[1] Kim, H., Ramachandran, U.: Flashfire: Overcoming the performance bottleneck of flash storage technology. Tech. rep., Georgia Institute of Technology (2010)

[2] Callahan, D., Kennedy, K., Porterfield, A.: Software prefetching. ACM SIGARCH Computer Architecture News 19(2), 40–52 (1991)

[3] H. Choi and S. Park, "Learning Future Reference Patterns for Efficient Cache Replacement Decisions," in IEEE Access, vol. 10, pp. 25922-25934, 2022, doi: 10.1109/ACCESS.2022.3156692.

[4] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. 1995. A study of integrated prefetching and caching strategies. SIGMETRICS Perform. Eval. Rev. 23, 1 (May 1995), 188–197. https://doi.org/10.1145/223586.223608

[5] X. Lu, H. Najafi, J. Liu and X. -H. Sun, "CHROME: Concurrency-Aware Holistic Cache Management Framework with Online Reinforcement Learning," 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Edinburgh, United Kingdom, 2024, pp. 1154-1167, doi: 10.1109/HPCA57654.2024.00090. keywords:

[6] J. Lee, H. Kim, and R. W. Vuduc, "When prefetching works, when it doesn't, and why," ACM Trans. Archit. Code Optim., vol. 9, no. 1, pp. 2:1–2:29, 2012.

[7] Ki, A., Knowles, A.E.: Stride prefetching for the secondary data cache. Journal of systems architecture 46(12), 1093–1102 (2000)

[8] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in Proceedings of the 35th International Conference on Machine Learning, ICML 2018, 2018, pp. 1924–1933.

[9] Chakrabortti and H. Litz, "Learning I/O access patterns to improve prefetching in ssds," in Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2020, Proceedings, Part IV, 2020, pp. 427–443.

[10] G.O. Ganfure, C. -F. Wu, Y. -H. Chang and W. -K. Shih, "DeepPrefetcher: A Deep Learning Framework for Data Prefetching in Flash Storage Devices," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3311-3322, Nov. 2020, doi: 10.1109/TCAD.2020.3012173.

[11] C.Yang, X. Man and J. Shao, "G&L: An Attention-based Model for Improving Prefetching in Solid-state Drives," 2023 International Joint Conference on Neural Networks (IJCNN), Gold Coast, Australia, 2023, pp. 1-8, doi: 10.1109/IJCNN54540.2023.10191741.

[12] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," in IBM Systems Journal, vol. 5, no. 2, pp. 78-101, 1966, doi: 10.1147/sj.52.0078.

[13] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea (South), 2016, pp. 78-89, doi: 10.1109/ISCA.2016.17.

[14] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's min policy," in 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2022, pp. 558–572.

[15] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 413–425.

[16] X. Lu, R. Wang, and X.-H. Sun, "Care: A concurrency-aware enhanced lightweight cache management framework," in 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2023, pp. 1208–1220.

[17] Msr cambridge traces.http://iotta.snia.org/traces/4928

[18] Msr cambridge traces. http://iotta.snia.org/traces/388

[19] J. Liao and S. Chen, "Optimization of reading data via classified block access patterns in file systems," IEEE Access, vol. 4, pp. 9421–9427, 201