Learning I/O Access Patterns to Improve Prefetching in SSDs

Chandranil Chakraborttii[1] (✉) and Heiner Litz[1]

Department of Computer Science, University of California Santa Cruz
11156 High St, Santa Cruz, CA 95064, USA
cchakrab@ucsc.edu, hlitz@ucsc.edu

Abstract. Flash based solid state drives (SSDs) have established themselves as a higher-performance alternative to hard disk drives in cloud and mobile environments. Nevertheless, SSDs remain a performance bottleneck of computer systems due to their high I/O access latency. A common approach for improving the access latency is prefetching. Prefetching predicts future block accesses and preloads them into main memory ahead of time. In this paper, we discuss the challenges of prefetching in SSDs, explain why prior approaches fail to achieve high accuracy, and present a neural network based prefetching approach that significantly outperforms the state-of the-art. To achieve high performance, we address the challenges of prefetching in very large sparse address spaces, as well as prefetching in a timely manner by predicting ahead of time. We collect I/O trace files from several real-world applications running on cloud servers and show that our proposed approach consistently outperforms the existing stride prefetchers by up to 800× and prior prefetching approaches based on Markov chains by up to 8×. Furthermore, we propose an address mapping learning technique to demonstrate the applicability of our approach to previously unseen SSD workloads and perform a hyperparameter sensitivity study.

Keywords: Prefetching · neural network · flash

1 Introduction

Solid state drives (SSDs) have become the primary storage device technology for mobile devices and high-performance servers. SSDs have replaced the spinning disks (HDDs) for many applications in cloud services due to their higher I/O performance [47], lower failure rate [37], and better endurance [34]. Nevertheless, although SSDs deliver significantly higher speeds than HDDs, SSDs still remain a performance bottleneck of computing systems [24], as processors and DRAM technologies support three orders of magnitude lower access latency. Two common approaches to hide the high access latency of storage devices are caching and prefetching. Caching utilizes less dense but faster types of memory to store frequently used data items, filtering out many accesses of the slow SSDs. Examples include Linux's page cache [12] and filesystem caches [43]. Prefetching [9]

approaches read data from SSDs in advance, in order to serve the later demand accesses from the cache with low latency. Prefetching can be implemented either in software, e.g., within operating system [29, 35] or within the SSD itself [48].

Existing prefetching mechanisms [39, 38] are limited by the computational complexity and difficulty of correctly predicting future I/O accesses. For instance, the read-ahead prefetcher [15, 26] is limited to prefetching the next data item within a file to accelerate sequential accesses. More advanced prefetchers [17, 5] that can learn complex I/O access patterns have been dismissed because of their computational cost. Recently, storage vendors, including Samsung, have proposed SmartSSDs [33, 14], adding computational capabilities to SSDs. These devices offer new opportunities as they enable offloading of prefetching to hardware, removing the burden from the host CPU. While this approach addresses the compute overhead of prefetching, predicting future I/O accesses accurately remains a challenge. Real-world applications not only perform sequential accesses, but also exhibit complex workload patterns [7]. Applications are frequently used by multiple users simultaneously, performing independent tasks, resulting in a mix of sequential and random I/O requests which are difficult to model and challenging to predict. Furthermore, in existing systems, I/O accesses need to traverse a deeply layered software stack, transforming the easy to predict accesses on the application side into seemingly random accesses on the SSD level. Predicting future memory accesses from multiple interleaved I/O access streams on the SSD device layer hence represents a challenging problem.

Modern SSDs and operating systems offer a wide range of telemetry data for analysis. Utilizing I/O access tracing in hardware and software enables the collection of large, clean, and automatically labeled datasets that can fuel powerful machine learning models. In this work, we leverage Long Short-Term Memory (LSTM) [19] based sequence-to-sequence neural networks to learn spatial I/O access patterns of applications from block level I/O traces collected from a diverse set of data center applications. LSTMs are capable of capturing long-term dependencies in data and can address sequences of different lengths. LSTMs integrate model training and representation learning together, without requiring additional domain knowledge, enabling the discovery of unseen patterns in the data to improve generalization capability of a model. In this work, we leverage LSTMs to deliver the following contributions. First, our model provides high accuracy even in the presence of complex interleaved I/O streams. Second, it addresses the challenge of timeliness by predicting multiple I/O accesses ahead of time. Third, to cope with the dynamic behavior of applications and to improve the reusability of our model, we propose an address mapping learning (AML) technique enabling our model to predict different types of workloads. To demonstrate the practicality of our approach, we build a simulator enabling us to measure timeliness in addition to prediction accuracy. We utilize I/O traces to train the neural network models offline and predict future logical block addresses (LBAs) at runtime using the simulator. To reduce address space, we take the l_1 norm between a pair of consecutive memory accesses as input to the model in addition to the requested I/O size. This enables the model to also predict the

size of the incoming I/O request, representing the amount of data blocks to be prefetched ahead of time. We show that our approach enables predicting LBAs sufficiently far ahead to compensate for the read latency of accessing flash as well as for the inference latency of our model. We present an analysis of the impact of predicting N steps ahead into the future and evaluate the impact of cache size on the performance of our prefetcher. We compare our work with three baselines, a naive approach that only prefetches the most frequently accessed LBAs, a stride prefetcher [23], and the Markov chain based prefetcher [11, 26, 48], showing an improvement of up to 800× over the stride prefetcher and up to 8× over the Markov chain prefetcher.

2 Background

2.1 Flash Device Architecture

NAND flash drives or SSDs are non-volatile memory devices storing individual bits on floating gate transistors. Floating gate transistors are arranged in large bit cell arrays increasing not only the storage capacity, but also the access latency. Furthermore, flash cells suffer from limited endurance and frequent bit errors, which are exacerbated by transistor scaling and the introduction of techniques such as multi-level cells [31]. To ensure data integrity, multiple reads using different reference voltages need to be performed, and the controller needs to perform error detection and correction as part of each read, further increasing the read latency. As a result, the I/O access latency of SSDs ($\sim$100μs) is three orders of magnitude higher than the latency of reading DRAM ($\sim$100ns). Hence, a mechanism that prefetches the data into DRAM provides significant performance gains.

2.2 Prefetching

Prefetching in storage systems is the process of preloading data from a slow storage device into faster memory, generally DRAM, to decrease the overall read latency. Accurate and timely prefetching can effectively reduce the performance gap between different levels of memory [30]. There are three important metrics used to compare prefetchers including coverage, accuracy, and timeliness of prefetchers [23]. Coverage is the ratio of the number of SSD reads that can be prefetched to the total number of SSD reads. Accuracy is the ratio of number of data blocks being prefetched to the number of prefetched data blocks that were actually requested by the application. Timeliness requires data blocks to be prefetched sufficiently ahead of time so that the data is present in DRAM whenever the read request is performed by the application. If the prefetched data blocks are not available when they are needed, the application is required to stall, rendering prefetching ineffective. Furthermore, if the data is prefetched too early, it may not be available anymore when it is actually needed, due to the eviction from the capacity-limited cache. Inaccurate prefetches that read in unneeded

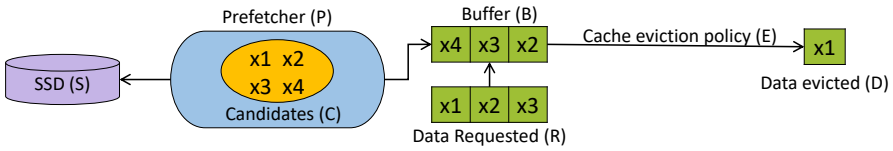

Fig. 1. Demonstration of prefetching at time, t

data are harmful as they waste I/O bandwidth and DRAM capacity. If prefetching is performed too conservatively, coverage is low and the overall performance gains are limited. Hence, the ideal prefetcher has high coverage, high accuracy, and executes prefetching timely so that the data is fetched exactly when it is needed. A basic prefetching mechanism is shown in Fig. 1. The SSD prefetcher (P) is responsible for predicting candidate data blocks (C) to prefetch from the SSD (S) into a fast cache (DRAM) buffer (B) of size s. The cache eviction policy (E) is responsible for evicting the data blocks from B in order to make space for new incoming data. In this example, at time t, P determines candidates x1, x2, x3, and x4 for prefetching, but the actual data requested at time t is x1, x2, and x3. Here, x1 was prefetched too early while x4 was inaccurately prefetched, resulting in cache miss in both the cases. Candidates x2 and x3, however, were present in the cache when requested, and hence, result in cache hit.

2.3 Neural Network based Prefetching

While most work on I/O prefetching has focused on conventional techniques, some prior works have explored using machine learning techniques. Hashemi [18] used neural network based sequence models for prefetching DRAM accesses. The models proposed in this work, however, cannot be applied to our problem as prefetching I/O accesses differs significantly from prefetching DRAM accesses. First, I/O accesses do not contain instruction information to enable stream disambiguation, second, I/O accesses do not have a fixed size like DRAM accesses, third, I/O accesses and DRAM accesses interact differently with the OS, and fourth, I/O prefetching models need to account for timeliness. A second line of work utilized Markov chains [11] for prefetching data from SSDs [26, 48]. We compare our approach with these prior works in Section 6, confirming prior observations that Markov chain based prefetchers perform poorly on real world applications where the I/O streams are more complex [44].

3 Problem Statement

We assume a digital system that consists of the following components. A flash based digital storage device (SSD) that provides high capacity but low performance, and a high access latency. A central processing unit (CPU) that can process data at orders of magnitude faster than the SSD. In addition, the system is comprised of a cache (usually DRAM) that is placed in between the CPU

and the SSD. The CPU can access data with low latency from the cache, however, the cache capacity is orders of magnitude smaller than the SSD capacity. Reads access a specific logical block address (LBA) and are generally more performance critical than writes, as future operations depend on the data supplied by the reads, which is why this work focuses on reads. The goal we aim to achieve is to accurately predict future LBAs so that they can be prefetched into the cache, enabling low latency accesses by the CPU. In addition to the LBA, we also need to predict the size of the I/O, as prefetching only parts of an I/O access is useless. Thus, an efficient prefetching mechanism requires optimizing three metrics, particularly, the *coverage*, *accuracy*, and *timeliness*.

Coverage or recall refers to the ratio of future memory accesses that are attempted to be prefetched. Prefetching of an LBA is accurate if the same LBA is subsequently accessed by a demand read. Accuracy is hence defined as the ratio of accurate prefetches to executed prefetches. A prefetch is timely if it is executed sufficiently ahead of time of the demand read. In particular, $T_{cand} + T_{read} < PA * T_{arrival}$ must hold, where T_{cand} represents the time to compute a prefetch candidate, T_{read} represents the time to perform a read from the SSD, $T_{arrival}$ represents the inter arrival time between demand reads, and PA represents prefetch-ahead, which is the number of accesses we need to predict into the future. Executing prefetches too early is generally of a lesser concern as prefetches can be stored for a finite time in the cache. As a result, the time that a prefetch can be executed too early is bounded only by the cache capacity.

Storage accesses to an LBA are generally handled by the operating system. User applications, however, generally communicate with the storage devices by reading and writing files. Consequently, the filesystem layer within the OS needs to map file accesses to LBA accesses before they can be submitted to the storage device. Furthermore, to improve performance, the OS maintains several caching layers in the filesystem and logical block layer, aiming to filter out a significant fraction of all application accesses. The result of this architecture is that even a seemingly easy to predict operation on the application layer, such as reading a file sequentially, may result in a very hard to predict access patterns on the LBA level, as perceived by the SSD. Finally, the storage device is generally accessed by different application threads simultaneously, resulting in multiple interleaved I/O streams that are indistinguishable by the SSD. In summary, the existing storage stack architecture renders predicting future I/O accesses a challenging problem. Predictive models need to be able to separate multiplexed I/O streams and then predict future LBAs from within the hard to predict sequences. In addition, they need to provide information on the number of data blocks to prefetch, starting from the initial predicted LBA.

4 Proposed Prefetching Technique

Learning SSD storage accesses for prefetching is a challenging task for the following reasons. As SSDs are increasing their storage capacity to 16TB and beyond, drives are now supporting billions of logical block addresses. As prefetching is

only successful if every bit of the logical block address is predicted accurately, models are required to predict which LBA to prefetch with perfect accuracy within a very large LBA space. This space is often sparse, as the operating system allocates blocks within the filesystem layer, and hence, even sequential data within files may be mapped to arbitrary LBAs within the SSD. Furthermore, as prefetches need to be timely, predicting only the next LBA and the requested I/O size is not sufficient, and it is required to predict several accesses into the future. Finally, to support dynamically changing workloads, we evaluate our proposed address mapping learning technique to determine whether prefetching models can learn generalized patterns within complex I/O access patterns.

4.1 Data Preparation for Reducing the Output Label Space

We preprocess the input dataset to address the problem of large logical block address space. The number of unique memory addresses within an SSD is typically of the order of billions, rendering a separate class for each memory address impractical. To reduce the address space, we take the l_1 norm of each pair of consecutive LBAs (LBA delta). For example, if consecutive I/O accesses starting from LBA 10000 are requested as 10001, 10003 and 100006, the corresponding LBA deltas were recorded as 1, 2, and 3, respectively. This significantly reduces the number of classes that our model needs to predict. We identify the top 1000 frequently occurring LBA deltas and assign each one of them to a class in decreasing order of frequency. All remaining LBA deltas are assigned to a separate class representing a "no prefetch" operation, thus limiting the number of classes for model to predict to 1001. The reason for choosing LBA deltas over actual addresses is to increase the coverage of LBA deltas in the data. For example, for Microsoft Research Cambridge traces [27] (MSR_1), the top 1000 most frequently occurring LBAs covered only 2.77% of all the LBA accesses, whereas the top 1000 most frequently occurring LBA deltas covered 91.66% of all LBA accesses. The coverage of top 1000 frequent LBA deltas for the datasets used in this study ranged between 54% and 92%, as seen in Table 1. Expanding the number of classes to beyond 1000 is possible with more computational power, however, for our datasets, we chose 1000, as it provides a considerable coverage for LBAs and is a sufficiently large size to prove the practicality of our approach.

The requested I/O sizes for the analyzed real world applications ranged from 4KB to several MBs with up to 10,000 different I/O sizes for an individual application. In order to reduce the number of possible target I/O size values, we round off each observed I/O size to the nearest number that is a power of 2,

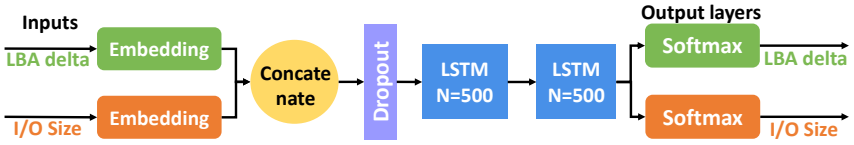

Fig. 2. Model architecture

2^n, and use n as an I/O size class. This reduces the number of possible target I/O sizes for most applications to 16 while still supporting requests of size up to 64MB. A limitation of this approach is that, in the worst case, roughly twice as many as required 4KB blocks may be prefetched from the SSD.

4.2 Model Architecture

We designed our proposed neural network model to predict both the I/O size and LBA deltas at the same time. The model has two separate input layers, one for I/O size and one for LBA delta, where each input layer is an embedding layer [49] consisting of 500 neurons. The inputs to the model are categorical, one-hot, representation of the two features, LBA deltas and I/O size, each being fed to a separate embedding layer. The model has two hidden LSTM layers, where each LSTM layer has 500 hidden nodes. The outputs of the two embedding layers are first concatenated and then fed to the shared LSTM layers. The final output layer is split into two branches, where each branch is a dense layer consisting of softmax [32] nodes. The number of neurons in the LBA delta output layer is 1001, representing top 1000 LBA deltas and a "no prefetch" LBA delta, and the number of neurons in the I/O size output layer ranged between 12 and 20, depending on the I/O sizes present in each dataset. The model architecture is shown in Figure 2. The number of neurons in each of the first three layers of the model was set to 500 to ensure a good representation of input features, and we used a dropout [16] of 0.2 to prevent overfitting of the model. Having an initial embedding layer facilitates better representation of the input features and helps the subsequent LSTM layers to learn effectively from sequential data.

4.3 Timeliness

As discussed in Section 3, a prediction from the prefetcher is timely only if the following equation holds: $T_{cand} + T_{read} < PA * T_{demand}$. We empirically determined T_{cand} to be 734μs by measuring the inference latency of our model. We measured the latency of accessing an Intel P3600 NVMe based SSD using the flexible I/O tester (FIO) [6] to be 300μs on average under 80% workload. For the traces that we examined, the average time between two successive I/O requests ranged between 800μs and 1200μs, and the minimum time was 10μs. As a result, a good PA value is in the range of $5 > PA > 100$. We evaluate a range of PA values and its impact on prediction accuracy in Section 6. Predicting further ahead in the future typically reduces the accuracy due to the increased uncertainty. We find that, in order to increase the accuracy in case of a high PA value, training the model with longer history of sequences can improve performance.

4.4 Address Mapping Learning

Different workloads show similar I/O access patterns due to shared design patterns and commonly used data structures. For instance, array-based data structures used by applications generally entail sequential I/O access patterns. Furthermore, as most applications leverage the same underlying filesystem, it is

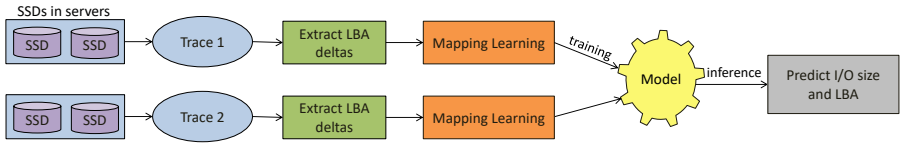

Fig. 3. Block diagram of the Address Mapping Learning process

likely that I/O accesses show common patterns. An ideal prefetcher would be trained once, on a varied set of applications, providing high performance even for previously unseen applications. Such a prefetcher is also likely to be more robust with respect to dynamically changing data inputs or code changes to the original application. To test the idea that applications share common patterns that can be learned, we train the model on traces from one dataset (source) and evaluate the performance of the prefetcher on another dataset (recipient). The mapping of addresses to labels is done by sorting the frequency distribution of LBA deltas from *both* the source and recipient traces and assigning them labels in decreasing order of frequency of occurrences. We call the process of extracting the LBA deltas, training the model on source dataset, and using the model to predict LBA deltas and I/O sizes for the recipient dataset as Address Mapping Learning (AML) and present the block diagram of this process in Fig. 3.

5 Methodology and Experimental Setup

5.1 Model Training

For our experiments, we used a total of 10 block-level I/O traces from three different sources running applications in live production servers. The datasets included traces describing enterprise storage traffic in commercial office virtual desktop infrastructure (VDI) [27], as well as traces from live production servers at Microsoft SNIA [22] and Microsoft Research Cambridge [36]. We did not use any synthetic benchmarks, as used in previous work [26, 48], as those traces do not accurately represent the complexity and interleaved patterns exhibited in real applications. The utilized trace files are open-source and can be obtained online [2, 1, 22]. Table 1 provides information about the datasets used in this study. From the table, we see that the coverage of top 1000 LBA deltas is consistently higher than direct memory addresses (offset), and hence it was selected as one of the features for training the model. The datasets also contained other information such as the I/O size, response time, filename, file location, etc. In this work, we only used the timestamp, offset (LBA), and I/O size as features. We trained our model using Google's Tensorflow [3] library on a Intel Xeon server with 8 CPU cores running at 1.7GHz containing 96GB of DRAM. The server also had 4 NVIDIA Tesla 2080TI GPUs for training the model. We split the dataset into training and test set, where the training set contained the first 70% of the I/O accesses, and the test set contained the last 30% of the I/O accesses. The

Table 1. Dataset Description

Trace Source	Dataset Name	Represented Name	Num obs	Coverage Offset (%)	Coverage LBA Delta (%)
VDI	2016022315.csv	VDI_1	5226120	58.76	66.96
VDI	2016030817.csv	VDI_2	4443487	63.94	70.08
VDI	2016030819.csv	VDI_3	2902328	68.94	69.8
VDI	2016031115.csv	VDI_4	2408227	68.65	72.35
MSR	proj_3.csv	MSR_1	2244642	2.77	91.66
MSR	mds_0.csv	MSR_2	1211034	63.46	76.94
MSR	src1_1.csv	MSR_3	45746222	28.6	77.7
MSR	usr_1.csv	MSR_4	45283980	2.64	82.12
Microsoft	buildserver-2.csv	MS_1	1600430	2.77	28.84
Microsoft	buildserver-7.csv	MS_2	1714151	8.97	55.49

sequence of LBA deltas, ordered by timestamps, is fed to the model for training. For all the experiments, we trained our model using Adam optimizer [46] with a cross-entropy loss function, and a learning rate of 10^{-3} for up to 1000 epochs, and stopped model training if there was no improvement in validation loss, with validation loss not decreasing by at least 10^{-5} for five consecutive epochs.

5.2 Prefetcher Simulation Environment

To enable the comparison of our prefetcher against prior baselines, evaluating only recall and precision is not sufficient. As motivated before, analyzing the prefetcher's timeliness is required to evaluate the end-to-end performance gains of prefetching, as even the most accurate prefetcher will not improve the performance if it lacks timeliness. As shown in Section 4.3, in order to compensate for the model's prediction latency and the latency to perform a read from the SSD, it is required to generate predictions ahead of time (PA). We evaluate the end-to-end performance as follows. As we iterate through the test dataset, the evaluation models continuously generate prefetch candidate predictions that are inserted into the cache.

Every I/O access is checked against the cache to see if the LBA is present, where the access is recorded as a hit, otherwise it is recorded as a miss. We

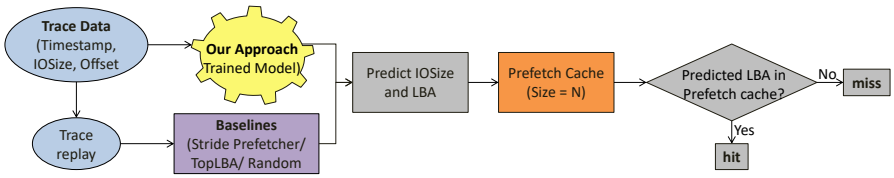

Fig. 4. Block diagram of the evaluation process using our simulator

utilize the Least Recently Used (LRU) [42] eviction policy for our experiments. The architecture of the simulator is presented in Fig. 4. We choose variable cache sizes of LBAs for the stride, Markov-based, and our proposed prefetcher, and run experiments to provide a comparative study in Section 6.

5.3 Baselines

We compare our proposed prefetcher to three baselines. The first, naive prefetcher, baseline always predicts the most common delta of a trace. The second baseline implements a Markov chain predictor [26, 48]. This method treats each LBA access as a state and predicts the next LBA based on the previous state by computing a probability distribution over the probabilities of transition from one state to another. The third baseline is a stride prefetcher which is commonly used in software and hardware systems. The stride prefetcher concurrently observes 128 I/O access streams. Each access is mapped to a stream based on hashing the most significant bits of the LBA. For each stream, the stride prefetcher tracks the last three I/O accesses. If the difference between the three I/O accesses match, the prefetcher detects a stride and prefetches the next access. Note that the stride prefetcher's results are optimistic, as it only prefetches one access ahead of time and does not compensate for timeliness. In the next section, we evaluate our proposed prefetcher in terms of prediction accuracy, timeliness, and capability to generalize to different workloads.

6 Results

6.1 Prefetcher Accuracy, Precision and Recall

Table 2 shows the comparative performance of our neural network based prefetcher against the three chosen baselines. The table lists the dataset name, number of samples in the dataset, and the accuracy for the three chosen baselines, Naive prefetcher, Stride prefetcher, and Markov chain based prefetcher. For our approach, we provide the accuracy, precision, and recall results. For each sample, our prefetcher predicts both LBA and I/O size in increments of 4KB blocks, as the minimum block size for a drive operation in SSD is typically of 4KB size [33]. We only count the actual blocks that are correctly prefetched. For each data sample, we prefetch only the top predicted LBA and I/O size using the prediction with the highest confidence. We used a batch size of 64, look back of 64, and predict-ahead of 64 in this experiment. Each prefetcher has a cache size of 1000 for this experiment. In the next section, we present a more detailed analysis of the impact of cache size on the performances of the prefetchers.

As shown in Table 2, our proposed prefetcher consistently outperforms all three baselines delivering up to 11× improvement over the stride prefetcher using Microsoft SNIA traces with the same cache size. For VDI traces, our proposed prefetcher achieves the highest accuracy, providing 800× improvement over the stride prefetcher. Our prefetcher also achieved the highest precision and recall

Table 2. Performance comparison of Our proposed prefetcher against baselines

Dataset Name	No. Samples	Naive Prefetcher	Stride Prefetcher	Markov Prefetcher	Our (Accuracy)	Our (Precision)	Our (Recall)
VDI_1	5226120	0.17	0.01	0.09	0.73	0.76	0.71
VDI_2	4443487	0.21	0.01	0.07	0.59	0.75	0.49
VDI_3	2902328	0.19	0.02	0.12	0.66	0.73	0.57
VDI_4	2408227	0.21	0.05	0.09	0.73	0.77	0.69
MSR_1	2244642	0.14	0.01	0.21	0.41	0.66	0.31
MSR_2	1211034	0.09	0.21	0.17	0.49	0.65	0.33
MSR_3	45746222	0.12	0.001	0.16	0.79	0.89	0.46
MSR_4	45283980	0.33	0.007	0.15	0.53	0.66	0.38
MS_1	1600430	0.27	0.02	0.25	0.63	0.79	0.53
MS_2	1714151	0.41	0.003	0.07	0.77	0.83	0.61

compared to the baselines. The Markov chain based prefetcher performed considerably worse compared to our prefetcher, with the accuracy ranging between 7% and 25%, performing even worse than the Naive prefetcher in several cases.

6.2 Impact of Cache Size, Look-Back, and Predict-Ahead

In this section, we present an analysis of the impact of look back, predict-ahead, and cache size on our proposed prefetcher's performance. In order to ensure the availability of data in the cache when the data block is requested, we trained the model to predict N steps ahead for varying values of N, and evaluated the performance of the prefetcher. Higher values of N typically resulted in lower accuracy due to the increased uncertainty in predicting further ahead in the future, while improving timeliness. To improve our prefetcher's predict-ahead performance, we found that it is necessary to increase the look back size for increasing values of PA, where, as described in Section 4.3, good values for PA are in the range of $5 < PA < 100$. Low values (< 5) of PA result in cache misses as the data cannot be fetched soon enough, whereas higher values of PA (> 100) result in untimely predictions as the data gets evicted before requested. Table 3 shows the performance of our prefetcher for different values of PA showing the accuracy of predicting the LBA and I/O size, as well as the cache hit ratio (Net Hit ratio). We measured accuracy as the actual number of 4KB data blocks that were correctly prefetched for three different values of PA, 32, 64, and 128.

In general, the accuracy of predictions decreases as we predict further ahead, producing the worst performance when predicting 128 samples ahead. For MS SNIA traces, the performance was comparable for PA equal to 32 and 64, and the accuracy degraded significantly for PA=128, whereas for VDI and MSR Cambridge traces, the performance degradation was gradual. These results show that our approach is successful in prefetching SSD accesses, as PA equal to 32 or 64 is generally sufficient to ensure timeliness in real-world settings. Nevertheless, to support upcoming storage devices that support even higher request ratios, reducing the inference latency and predicting even further ahead will be required.

Table 3. Impact of different predict values on our prefetcher performance

Dataset	Predict Ahead = 32			Predict Ahead = 64			Predict Ahead = 128		
	Accuracy (LBA)	Accuracy (Size)	Net Hit ratio	Accuracy (LBA)	Accuracy (Size)	Net Hit Ratio	Accuracy (LBA)	Accuracy (Size)	Net Hit Ratio
VDI_1	0.72	0.65	0.71	0.69	0.65	0.73	0.42	0.6	0.33
VDI_2	0.76	0.51	0.58	0.64	0.51	0.59	0.41	0.42	0.29
VDI_3	0.73	0.88	0.69	0.48	0.88	0.66	0.42	0.67	0.37
VDI_4	0.71	0.66	0.71	0.71	0.66	0.73	0.32	0.34	0.31
MSR_1	0.65	0.49	0.41	0.65	0.49	0.41	0.34	0.19	0.29
MSR_2	0.59	0.69	0.49	0.59	0.69	0.49	0.19	0.61	0.33
MSR_3	0.95	0.67	0.66	0.91	0.61	0.79	0.13	0.61	0.19
MSR_4	0.59	0.77	0.51	0.49	0.77	0.53	0.49	0.47	0.28
MS_1	0.93	0.67	0.61	0.93	0.52	0.63	0.62	0.52	0.49
MS_1	0.89	0.71	0.73	0.88	0.69	0.77	0.57	0.69	0.47

Table 4. Impact of cache size on the accuracy of our and two baseline prefetchers

Dataset Name	Cache Size = 10			Cache Size = 100			Cache Size = 1000		
	Markov Prefetcher	Stride Prefetcher	Our Prefetcher	Markov Prefetcher	Stride Prefetcher	Our Prefetcher	Markov Prefetcher	Stride Prefetcher	Our Prefetcher
VDI_1	0.05	0.001	0.68	0.05	0.001	0.69	0.09	0.011	0.73
VDI_2	0.05	0.0001	0.55	0.05	0.0001	0.55	0.07	0.0015	0.59
VDI_3	0.04	0.0001	0.64	0.04	0.0001	0.64	0.12	0.0014	0.66
VDI_4	0.01	0.006	0.7	0.01	0.006	0.71	0.09	0.005	0.73
MSR_1	0.12	0.00005	0.39	0.12	0.00005	0.39	0.21	0.0011	0.41
MSR_2	0.09	0.1	0.41	0.09	0.1	0.41	0.17	0.21	0.49
MSR_3	0.07	0.0002	0.75	0.07	0.0002	0.76	0.16	0.001	0.79
MSR_4	0.06	0.0005	0.51	0.06	0.0005	0.51	0.15	0.007	0.53
MS_1	0.16	0.004	0.57	0.16	0.004	0.57	0.25	0.02	0.63
MS_1	0.02	0.0003	0.71	0.02	0.0003	0.71	0.07	0.003	0.77

Table 4 presents the impact of varying cache size on our prefetcher's performance. The table shows the accuracy of our approach compared to the Markov and Stride prefetchers for cache sizes of 10, 100, and 1000 LBAs, respectively. From the table, we can see that our prefetcher consistently outperforms the baselines for each cache size, and the performance improvement using VDI traces is as high as 800× over the Stride prefetcher, and 8× over the Markov prefetcher. While the baselines show marginal improvements using larger cache sizes, our prefetcher benefits significantly from a larger cache size. This suggests that while our prefetcher provides high accuracy and coverage, its timeliness can still be improved. For a large cache, prefetched blocks remain in the cache for a longer time and hence, prefetching exactly at the time when the LBA is requested is less important. Achieving perfect timeliness would require adjusting PA dynamically, as the inter-arrival time between requests varies at runtime.

6.3 Evaluation of Address Mapping Learning

In this section, we evaluate whether our prefetcher can learn common patterns among workloads to predict accesses for previously unseen workloads. In the previous sections, we obtained the training and test datasets from different portions of the same workload and the trace file. In this section, we define two types of dataset sources. *Similar* sources are those where the training and test data

Table 5. Performance of Address Mapping Learning (AML)

	Similar Source					Dissimilar Source		
Source Trace	MSR_3	MSR_1	MS_1	VDI_1	VDI_3	MSR_3	MS_1	VDI_4
Recipient Trace	MSR_2	MSR_4	MS_2	VDI_2	VDI_4	VDI_3	VDI_1	MSR_2
Accuracy on Source Trace	0.95	0.63	0.93	0.75	0.87	0.92	0.92	0.82
Acuracy on Recipient Trace	0.59	0.59	0.87	0.72	0.75	0.75	0.75	0.72
AML Accuracy	0.37	0.39	0.84	0.52	0.47	0.31	0.22	0.35

are from the same application, however, with different data inputs, different execution times, and only small run time modifications in applications. *Dissimilar* sources are those where the training and test data are from completely different applications. Table 5 shows the prediction accuracy for different types of sources. We show the accuracy of the model when it is trained and tested on *similar* source traces, and also when it is trained and tested on the *dissimilar* source traces. In Table 5, for our proposed AML technique, the model is trained on the source trace and tested on the recipient trace. For instance, when training on MS_1 and evaluating on MS_2 trace files, the accuracy of our address mapping approach is 84% which is only 3% less than training and evaluating both on MS_2 (fourth column). The overall effectiveness of AML depends on the frequency distribution of LBA deltas in the two datasets. The results in Table 5 show that our approach can be applied to diverse workloads, as long as they share some similar characteristics. This increases the practicality of our approach, as we can train specific models for various workloads, and expect at least a moderate increase in performance for other workloads.

7 Related Work

Machine learning techniques have been applied to the prefetching problem in multiple domains such as web caching [4] and memory prefetching [18, 50]. While previous work also utilized neural networks for determining prefetch candidates, they operate on very different datasets, as DRAM accesses differ significantly from I/O accesses. For instance, I/O accesses are not tagged with the source instruction for stream disambiguation, I/O accesses do not have a fixed size [41] and, in contrast to I/O, memory accesses are not intercepted by the OS. Prior work on SSD prefetching utilized algorithmic approaches, typically using a data-range-table to detect usable strides and memory access streams [23]. Several variations of stride prefetchers have been proposed [25, 20] taking into account the spatial locality [20], feedback [40], and context [8]. However, as we showed in this work, algorithm based prefetchers do not perform well on real world applications due to their limited ability to learn complex patterns. The only prior research we are aware of that applies machine learning for prefetching in SSDs is based on Markov chains [48, 26], which we used as a baseline in this work. Finally, machine learning techniques have been applied to improve SSDs in other ways, for instance, by optimizing garbage collection [45], for predicting

device failures [37, 21], for improving SSD virtualization [13], for managing SSDs in large clusters [28], and for improving the quality of service of SSDs [10]. These prior works are orthogonal to our work.

8 Conclusion

In this paper, we showed how to leverage neural network models to predict future storage I/O accesses to improve SSD performance via prefetching. We addressed several challenges such as the large and sparse logical block address space, ensuring timeliness of prefetching, predicting both the address and size of I/O accesses, as well as the challenge of training predictive models that can generalize across different workloads. We achieved generalization across workloads by leveraging a large set of real world cloud application traces. We compared the performance of our prefetcher to existing techniques and used an in-house simulator developed to test the accuracy, coverage, and timeliness of our proposed prefetcher. Our proposed model outperforms prior approaches such as the stride prefetcher by up to 800× and Markov chain based prefetcher by up to 8×.

Acknowledgements

This work was supported in part by Samsung Semiconductor, Inc. and in part by NSF grants CCF-1823559 and CCF-1942754.

References

1. Microsoft snia: Traces. `http://iotta.snia.org/traces/4928`
2. Msr cambridge traces. `http://iotta.snia.org/traces/388`
3. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
4. Ali, W., Shamsuddin, S.M., Ismail, A.S., et al.: A survey of web caching and prefetching. Int. J. Advance. Soft Comput. Appl **3**(1), 18–44 (2011)
5. Averbouch, I., Birnbaum, A.J., Hsieh, J.T., Shum, C.L.K.: Automatic pattern-based operand prefetching (Feb 10 2015), uS Patent 8,954,678
6. Axboe, J.: Fio-flexible i/o tester synthetic benchmark. URL https://github.com/axboe/fio (Accessed: 2015-06-13) (2005)
7. Boboila, S., Desnoyers, P.: Performance models of flash-based solid-state drives for real workloads. In: 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–6. IEEE (2011)
8. Bradford, J.P., Kossman, H.F., Mullins, T.J.: Context switch instruction prefetching in multithreaded computer (Nov 10 2009), uS Patent 7,617,499
9. Callahan, D., Kennedy, K., Porterfield, A.: Software prefetching. ACM SIGARCH Computer Architecture News **19**(2), 40–52 (1991)
10. Chakraborttii, C., Sinha, V., Litz, H.: Ssd qos improvements through machine learning. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 511–511 (2018)

11. Chung, K.L.: Markov chains. Springer-Verlag, New York (1967)
12. Da Zheng, R.B., Szalay, A.S.: A parallel page cache: Iops and caching for multicore systems. In: Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems. pp. 5–5 (2012)
13. Dartois, J.E., Boukhobza, J., Knefati, A., Barais, O.: Investigating machine learning algorithms for modeling ssd i/o performance for container-based virtualization. IEEE transactions on cloud computing (2019)
14. Do, J., Kee, Y.S., Patel, J.M., Park, C., Park, K., DeWitt, D.J.: Query processing on smart ssds: opportunities and challenges. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. pp. 1221–1230 (2013)
15. Fengguang, W., Hongsheng, X., Chenfeng, X.: On the design of a new linux readahead framework. ACM SIGOPS Operating Systems Review **42**(5), 75–84 (2008)
16. Gal, Y., Ghahramani, Z.: Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In: international conference on machine learning. pp. 1050–1059 (2016)
17. Han, W.S., Whang, K.Y., Moon, Y.S.: A formal framework for prefetching based on the type-level access pattern in object-relational dbmss. IEEE Transactions on knowledge and data engineering **17**(10), 1436–1448 (2005)
18. Hashemi, M., Swersky, K., Smith, J.A., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., Ranganathan, P.: Learning memory access patterns. arXiv preprint arXiv:1803.02329 (2018)
19. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
20. Iacobovici, S., Kadambi, S., Chou, Y.C.: Multi-stride prefetcher with a recurring prefetch table (Feb 3 2009), uS Patent 7,487,296
21. Iwasaki, T.O., Ning, S., Yamazawa, H., Sun, C., Tanakamaru, S., Takeuchi, K.: Machine learning prediction for 13x endurance enhancement in reram ssd system. In: 2015 IEEE International Memory Workshop (IMW). pp. 1–4. IEEE (2015)
22. Kavalanekar, S., Worthington, B., Zhang, Q., Sharda, V.: Characterization of storage workload traces from production windows servers. In: 2008 IEEE International Symposium on Workload Characterization. pp. 119–128. IEEE (2008)
23. Ki, A., Knowles, A.E.: Stride prefetching for the secondary data cache. Journal of systems architecture **46**(12), 1093–1102 (2000)
24. Kim, H., Ramachandran, U.: Flashfire: Overcoming the performance bottleneck of flash storage technology. Tech. rep., Georgia Institute of Technology (2010)
25. Kondguli, S., Huang, M.: T2: A highly accurate and energy efficient stride prefetcher. In: 2017 IEEE International Conference on Computer Design (ICCD). pp. 373–376. IEEE (2017)
26. Laga, A., Boukhobza, J., Koskas, M., Singhoff, F.: Lynx: A learning linux prefetching mechanism for ssd performance model. In: 2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA). pp. 1–6. IEEE (2016)
27. Lee, C., Kumano, T., Matsuki, T., Endo, H., Fukumoto, N., Sugawara, M.: Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In: Proceedings of the 10th ACM International Systems and Storage Conference. pp. 1–11 (2017)
28. Li, B., Deng, C., Yang, J., Lilja, D., Yuan, B., Du, D.: Haml-ssd: A hardware accelerated hotness-aware machine learning based ssd management. In: 38th IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2019. p. 8942140. Institute of Electrical and Electronics Engineers Inc. (2019)
29. Li, M., Varki, E., Bhatia, S., Merchant, A.: Tap: Table-based prefetching for storage caches. In: FAST. vol. 8, pp. 1–16 (2008)

30. Liu, C.C., Ganusov, I., Burtscher, M., Tiwari, S.: Bridging the processor-memory performance gap with 3d ic technology. IEEE Design & Test of Computers **22**(6), 556–564 (2005)
31. Liu, R.S., Yang, C.L., Li, C.H., Chen, G.Y.: Duracache: A durable ssd cache using mlc nand flash. In: Proceedings of the 50th Annual Design Automation Conference. pp. 1–6 (2013)
32. Liu, W., Wen, Y., Yu, Z., Yang, M.: Large-margin softmax loss for convolutional neural networks. In: ICML. vol. 2, p. 7 (2016)
33. Mehra, P.: Samsung smartssd: Accelerating data-rich applications. Flash Memory Summit
34. Mohan, V., Siddiqua, T., Gurumurthi, S., Stan, M.R.: How i learned to stop worrying and love flash endurance. HotStorage **10**, 3–3 (2010)
35. Mowry, T.C., Demke, A.K., Krieger, O., et al.: Automatic compiler-inserted i/o prefetching for out-of-core applications. In: OSDI. vol. 96, pp. 3–17 (1996)
36. Narayanan, D., Donnelly, A., Rowstron, A.: Write off-loading: Practical power management for enterprise storage. ACM Transactions on Storage (TOS) **4**(3), 1–23 (2008)
37. Narayanan, I., Wang, D., Jeon, M., Sharma, B., Caulfield, L., Sivasubramaniam, A., Cutler, B., Liu, J., Khessib, B., Vaid, K.: Ssd failures in datacenters: What? when? and why? In: Proceedings of the 9th ACM International on Systems and Storage Conference. pp. 1–11 (2016)
38. Nijim, M.: Modelling speculative prefetching for hybrid storage systems. In: 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage. pp. 143–151. IEEE (2010)
39. Nijim, M., Zong, Z., Qin, X., Nijim, Y.: Multi-layer prefetching for hybrid storage systems: algorithms, models, and evaluations. In: 2010 39th international conference on parallel processing workshops. pp. 44–49. IEEE (2010)
40. Patt, S.O.H.Y.: Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers (2006)
41. Pike, R.: Storage mechanism with variable block size (Mar 13 2014), uS Patent App. 13/612,968
42. Puzak, T.R.: Analysis of cache replacement-algorithms. (1986)
43. Rodeh, O., Bacik, J., Mason, C.: Btrfs: The linux b-tree filesystem. ACM Transactions on Storage (TOS) **9**(3), 1–32 (2013)
44. Santos, J.R., Muntz, R.R., Ribeiro-Neto, B.: Comparing random data allocation and data striping in multimedia servers. ACM SIGMETRICS Performance Evaluation Review **28**(1), 44–55 (2000)
45. Smith, K.: Garbage collection. SandForce, Flash Memory Summit, Santa Clara, CA pp. 1–9 (2011)
46. Tato, A., Nkambou, R.: Improving adam optimizer (2018)
47. Wu, G., He, X.: Reducing ssd read latency via nand flash program and erase suspension. In: FAST. vol. 12, pp. 10–10 (2012)
48. Xu, R., Jin, X., Tao, L., Guo, S., Xiang, Z., Tian, T.: An efficient resource-optimized learning prefetcher for solid state drives. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 273–276. IEEE (2018)
49. Xue, B., Fu, C., Shaobin, Z.: A study on sentiment computing and classification of sina weibo with word2vec. In: 2014 IEEE International Congress on Big Data. pp. 358–363. IEEE (2014)
50. Zeng, Y.: Long short term based memory hardware prefetcher (2017)