Leveraging AI for Faster Storage Access: a Graph-Neural-Network-Based Prefetcher

Zeyuan Yang

Advisor: Professor Haryadi Gunawi

Abstract

Despite the widespread adoption of SSDs in cloud environments, I/O access (disk reads and writes) remains a significant performance bottleneck. To mitigate access latency, various prefetching techniques have been developed. However, as servers increasingly host multi-user workloads, the efficacy of traditional prefetchers diminishes. Fortunately, with the ascendance of artificial intelligence, there is an escalating interest in utilizing machine learning methods to predict access patterns more accurately. Nonetheless, these methods face challenges: large model sizes, high inference latency, and unstable hit rates. In this paper, we introduce “Spectral Prefetcher”, a spectral graph neural network for rapid and precise predictions. This approach 1) reduces training and inference latency by **79.9%**, 2) decreases memory usage by **33.3%**, and 3) achieves a state-of-the-art hit rate, surpassing the baseline by **21.8%.**

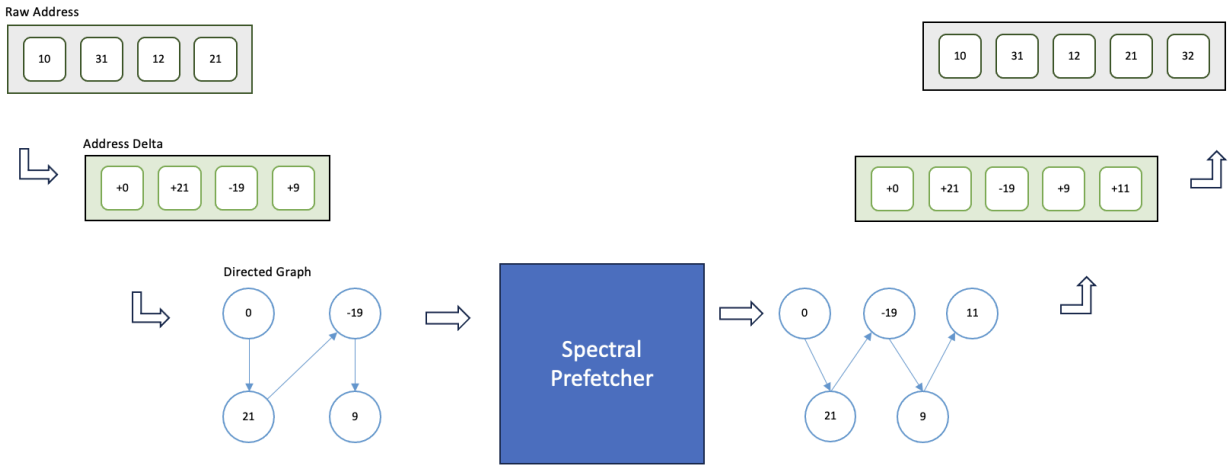
Section 1: Introduction

In the era of cloud computing, I/O (input/output) operations remain a significant performance bottleneck [9]. When users request data from a server, the server must retrieve it from the disk, a process that is notably slow. To address this, storage-level caches were developed to store data likely to be accessed in the future. However, data is only loaded into the cache upon a miss, resulting in a delay during the first access. Prefetching was introduced to pre-load data into the cache, yet it faces challenges: 1) non-sequential access patterns are hard to detect, 2) workloads have become increasingly complex and irregular, 3) prefetching latency cannot be high.

01 - Non-sequential pattern detection. Traditional prefetchers, often based on heuristic algorithms, struggle with non-sequential pattern detection [1]. For instance, the prefetcher in the Linux kernel uses the last two cache misses to predict the next access [10]. However, this method falters in complex, multi-user environments with interleaved I/O streams, suggesting the potential of spatial feature analysis for pattern mining. Our graph-based prefetcher leverages this approach to identify spatial patterns in access history.

02 - A learning-based approach. As workloads grow more complex, machine learning-based methods have emerged as superior solutions for adapting to any access pattern without preconceived assumptions. They can learn an access pattern from any workload without prior assumption to the trace. Thus, we employ a neural network to conduct prediction.

03 - Achieving fast inference. Existing ML-based algorithms, such as DeepPrefetcher [6], LSTM [3], and SGDP [5], suffer from high training and inference latency due to their complex architecture and data processing. We seek to create a lightweight model that can achieve similar performance. Our pipeline is depicted in the figure below.



[Figure 1: Spectral Prefetcher's Architecture]

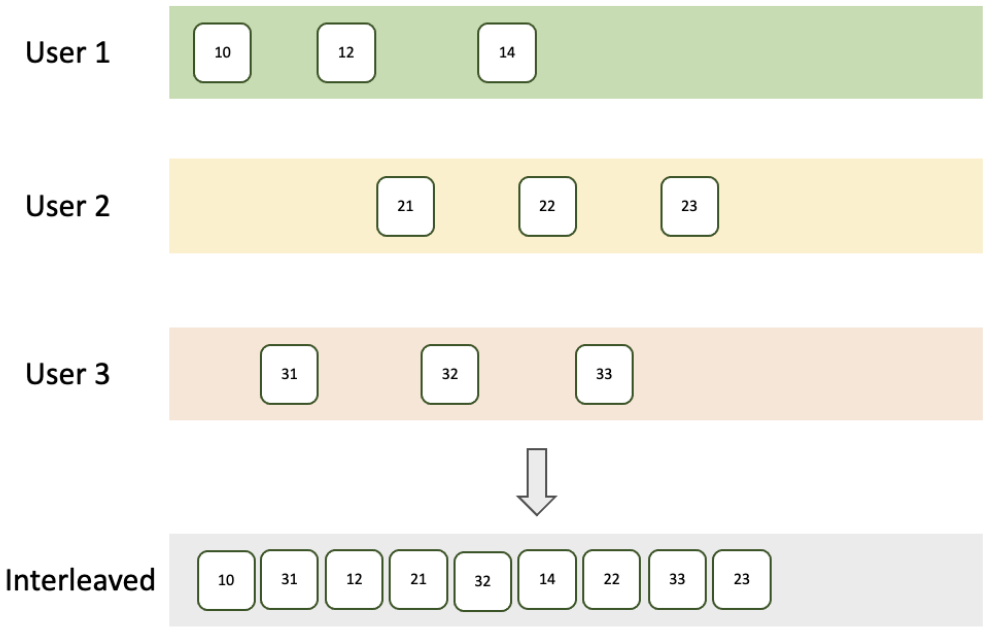
Our graph prefetcher's pipeline, depicted in Figure 1, begins by transforming a sequence of raw addresses into deltas (offsets from previous addresses). We maintain a sliding window, e.g., of length 32, of these address deltas. From there, we construct a directed graph with vertices representing address deltas and edges representing transitions. This graph is then processed by a spectral graph neural network (SpectralGNN), outputting a node class that represents an address delta. Using this

delta, we calculate the real address to prefetch. Benchmarking on extensive production server workloads from Microsoft, Alibaba, and Seagate Technology demonstrates that our model significantly reduces inference overhead while maintaining state-of-the-art prediction accuracy. We summarize our contributions as follows:

- To the best of our knowledge, Spectral Prefetcher is the first cache prefetcher that utilizes a spectral graph network for prediction. It avoids local irregularities and mines a global pattern.
- It surpasses the baseline (stride prefetcher) by 21.8% hit rate. It also maintains high memory utilization and low overhead.
- It lowers inference latency by 79.9% and reduces model size by 33.3% compared to the state-of-the-art ML-prefetcher (SGDP).

Section 2. Background and Related Works

Problem Formulation. In today's cloud environments, servers are increasingly multi-tenant, handling interleaved requests from numerous concurrent users and application streams. Consequently, even if an individual application's logical I/O sequence results in physically sequential accesses, or if the application's pattern is highly predictable, these patterns can be challenging to recognize at the storage system level. This complexity is evident in scenarios such as concurrent execution of multiple applications on a shared network-attached storage, as illustrated in Figure 1, and also in single applications with multiple threads that exhibit varied access patterns, such as a database application running multiple queries, as also depicted in [Figure 2].



[Figure 2: Interleaved Access Pattern of Modern Workloads]

Related work. The literature on prefetching techniques can be categorized into two main groups. The first category comprises sequential prefetchers, which analyze the entire sequence as a single stream for prediction purposes. This includes Stride Prefetcher, which we use as our baseline, and LSTM. The second category involves graph-based prefetchers, which maintain a global historical view, like a Markov chain's table, a probabilistic graph noting each address and recording the most probable subsequent nodes. This category includes Markov Chain and SGDP.

01 - Stride Prefetcher. It uses a historical table to store 128 LBA (Logical Block Address) streams. Each entry tracks the last 3 LBAs of that stream. When a new request arrives, the prefetcher hashes the most significant bits of the request's LBA and puts it to one of the 128 streams. If the difference between the last 3 LBA accesses matches, it will detect a stride and conduct a prediction [2].

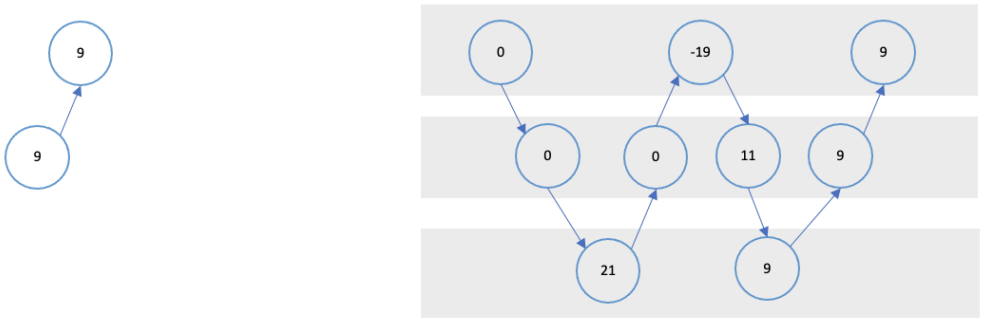
02 - LSTM. It utilizes a long-term, short-term memory ML model to predict the next LBA delta based on the last several (e.g. 128) LBA delta. It treats the addresses as a sequence and conducts a RNN-like prediction [3].

03 - Markov Chain Prefetcher. It uses a state transition model to predict the next block to be accessed. The transition model is a table that, for each block (LBA), records the probabilities of all the next potential blocks [4].

04 - SGDP. It builds a graph out of the last several (e.g. 128) LBA deltas, treating each delta as a node. Then it uses a gated graph neural network to transform each delta into a vector. Finally, it uses the vectors to predict the next LBA delta. This is the state-of-the-art ML model in terms of prefetching accuracy [5].

Section 3: Methodology

Why graphs can offer a global view of a sequence. Modeling the problem as a graph provides a comprehensive perspective on the data. Focusing only on the immediate one or two past accesses, discerning a discernible pattern becomes challenging. However, by examining the historical stream of data, distinct clusters of addresses or streams emerge. The graph-based approach leverages these patterns by representing them in a structure that highlights the relationships and interactions between different data points, thus providing a clearer understanding of the underlying trends and behaviors. See the figure below for an illustration [Figure 3].

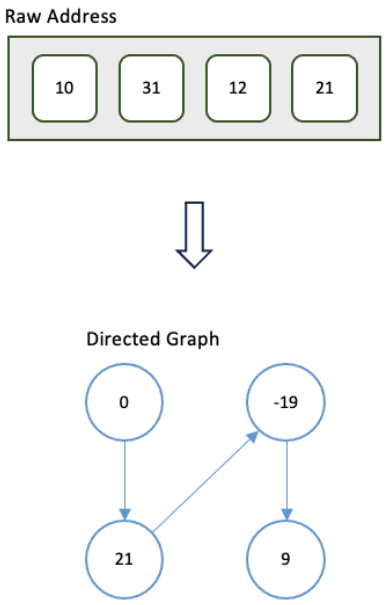


[Figure 3: Local view (left) doesn't show a pattern versus the global view (right) does]

The initial insight. The initial approach involved maintaining a historical table that recorded, for each address, which other addresses it was connected to. Expanding upon this concept, it's akin to managing a linked list where, for each node, the connected nodes are identified, and the edge weights could represent the probabilities.

This concept has inspired the use of graphs to model the I/O stream in recent studies, such as Probabilistic Graph [11] and SGDP [5].

How to construct a graph from a sequence. We employ a sliding window of size 32, meaning the graph can contain up to 32 vertices at any time. For example, if the access sequence moves from address 10 to address 31, a directed edge from vertex 10 to vertex 31 is created. Should there be another instance of transitioning from address 10 to address 31, the weight of the existing edge is incremented by one. This method enables the modeling of I/O stream patterns in a graph, providing insights into the frequency and likelihood of specific transitions. Notably, we use address deltas rather than raw addresses for graph construction, which facilitates the creation of a higher-order representation that captures the nuances of address sequences more effectively.



[Figure 4: How to build a graph from data stream]

Why use address delta. Using address deltas instead of raw addresses for constructing the graph offers significant advantages in the context of prefetching. Firstly, a stream may contain 32 unique addresses, but the number of unique offsets (deltas) between these addresses is typically lower, effectively reducing the graph's complexity. This reduction in vertices simplifies the graph, making it more manageable for analysis and prediction. Furthermore, predicting absolute addresses, which can range from 0 to \(2^{64}\) in a 64-bit address space, creates an impractically large prediction space. By focusing on the most frequent 1000 deltas as prediction classes, the approach narrows down the prediction space, making it feasible for the model to generate accurate predictions.

How Spectral GNN Works. After constructing the graph with a sliding window technique, the resulting adjacency matrix is input into the neural network. Spectral Graph Neural Networks (Spectral GNNs) operate on the principle of leveraging the spectral properties of graphs, which are derived from the eigenvalues and eigenvectors of their adjacency or Laplacian matrices. The output of a Spectral GNN, particularly a two-layer Graph Convolutional Network (GCN), can be generalized by an equation that integrates these spectral properties to perform convolution operations in the spectral domain. These operations effectively capture the topological structure and node feature information within the graph, enabling the network to learn and make predictions based on the complex patterns of I/O stream accesses represented within the graph structure. See formula below [Figure 5].

$$H^{(2)} = \sigma_2\left(\hat{A}\sigma_1\left(\hat{A}XW^{(0)}\right)W^{(1)}\right)$$

[Figure 5: Formula for Spectral GNN]

Where:

- X is the input feature matrix, where each row represents a node's features.
- W's are the weight matrices for the first and second layer, respectively.
- A is the normalized adjacency matrix with self-loops.

Graph Convolutional Networks (GCNs) leverage the spectral properties of graphs to learn node representations that encapsulate both structural and feature information of the graph. Utilizing the adjacency matrix and its normalization, GCNs perform

convolution operations that are inherently sensitive to the graph's topology. Through the application of successive layers, these networks aggregate information from broader neighborhoods, enabling the capture of global graph properties alongside local connections. This approach enriches node representations, allowing GCNs to effectively understand and predict complex relationships and patterns within graph-based data, such as in I/O stream prediction tasks.

Section 4: Experimental Setup

Workload. Our dataset comprises traces from enterprise production servers of four companies: Microsoft, Alibaba, Tencent, and Seagate. Notably, the Microsoft dataset, collected by Microsoft Research Cambridge, includes a one-week sequence of logical block addresses (LBAs) from live production servers. These workloads accurately represent the multi-tenant, multithreaded computing environments prevalent in today's cloud enterprises.

Model Architecture & Training. We developed our Spectral Prefetcher using PyTorch, incorporating two graph convolution layers followed by a ReLU activation and dropout layer. The chosen architecture, with a hidden layer size of 150, represents the optimal minimal design for achieving high prediction accuracy. Our training configuration includes a batch size of 128, the Adam optimizer, and cross-entropy loss, catering to our model's classification of the most probable next deltas out of a compressed prediction space of the 1000 most frequent deltas. This approach transforms the problem into a classification task rather than regression, significantly enhancing accuracy and efficiency by reducing the model's complexity.

Metrics. We use the following five metrics to evaluate our Spectral Prefetcher against the baselines.

1. Duration refers to the total time required to execute a workload, encompassing both the training phase and the inference phase of the model.

   $$\text{Duration} = \text{time to run a workload.}$$

2. Model size indicates the complexity and memory footprint of a machine learning model, quantified by the total count of its parameters. A larger model size typically means higher memory consumption.

   $$\text{Model size} = \#\text{parameters}$$

3. Hit rate is a performance metric that calculates the ratio of cache hits to the total number of I/O requests. A higher hit rate signifies a more effective prefetching strategy, as it indicates a greater proportion of successful cache accesses.

   $$\text{Hit rate} = \frac{\#\text{hits}}{\#\text{total I/O}}$$

4. Memory utilization assesses the effectiveness of prefetched data by measuring the proportion of prefetched blocks that are subsequently accessed (hits) in later operations. Higher memory utilization reflects better prediction accuracy and prefetching efficiency.

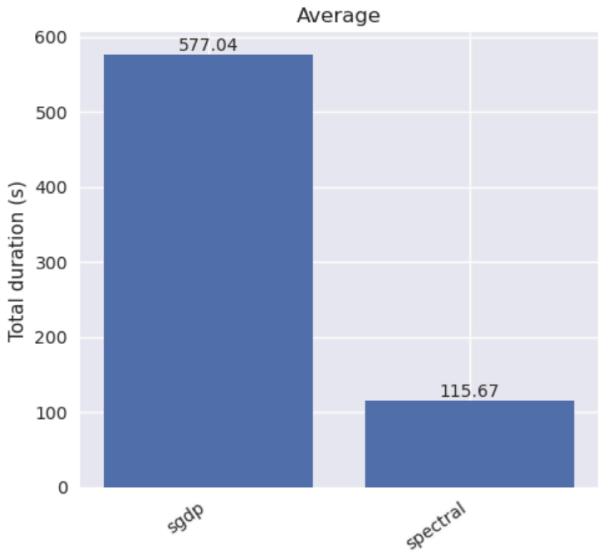
   $$\text{Memory utilization} = \frac{\text{bytes hit}}{\text{total prefetched bytes}}$$

5. Prefetch overhead is a measure of inefficiency in the prefetching process, quantified by the amount of data prefetched into cache that is never accessed by subsequent user operations. Lower prefetch overhead indicates a more precise prefetching strategy, reducing wasted resources and cache pollution.

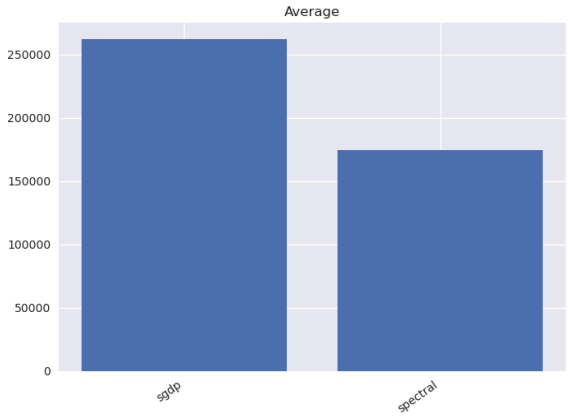
$$\text{Prefetch overhead} = \frac{\text{extra prefetched data}}{\text{user requested data}}$$

Section 5: Experimental Results

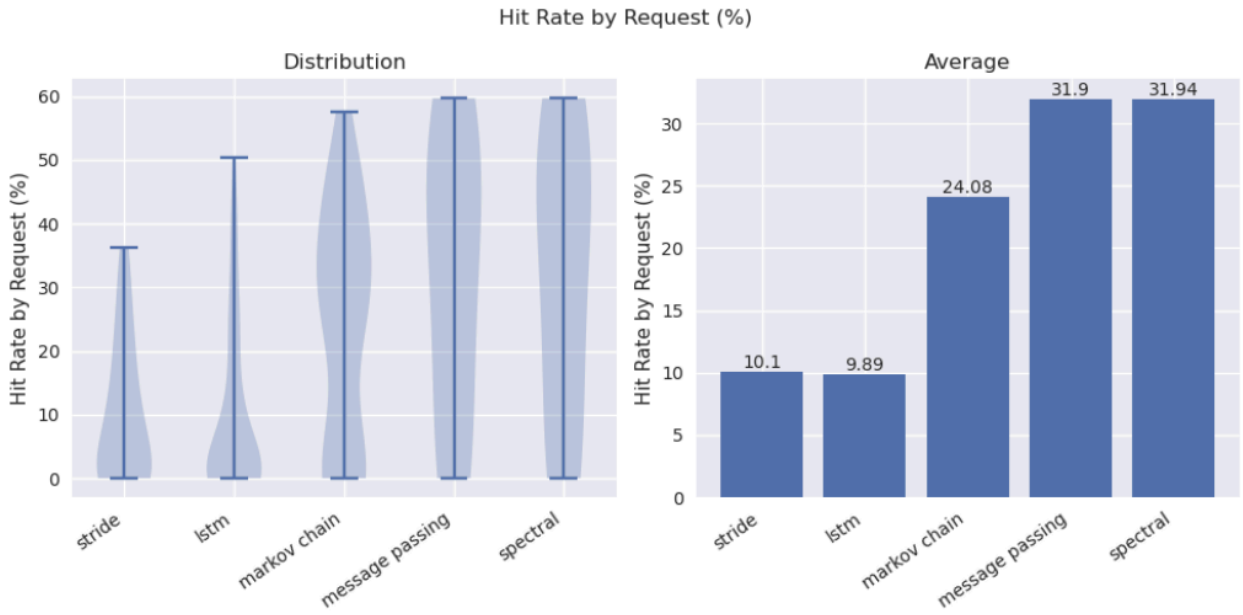
01 Spectral Prefetcher saves inference time by 79.9%. Compared to SGDP, the leading ML-based model, Spectral Prefetcher significantly lowering both training and inference durations. This substantial efficiency improvement underscores the Spectral Prefetcher's viability for deployment in high-performance cloud environments, where minimizing latency is crucial.



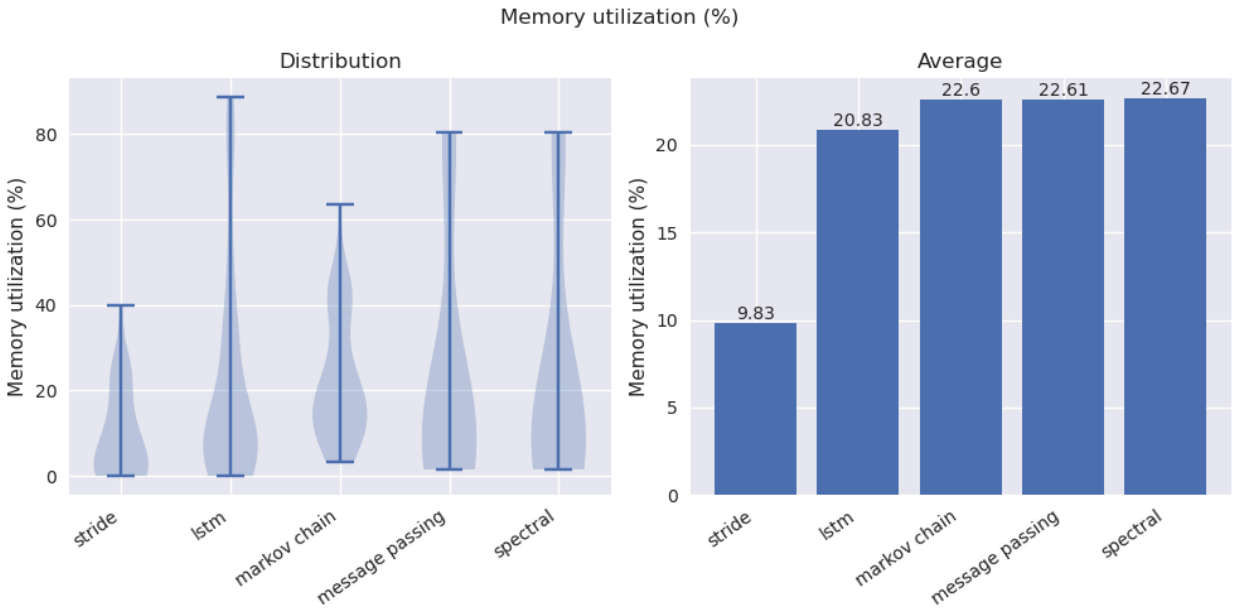
02 Spectral Prefetcher saves memory space by 33.3%. Model size is crucial since the model must be loaded into memory for inference. With a considerably smaller model size measured by the number of parameters, Spectral Prefetcher's size (174,403 parameters) is significantly less than that of the state-of-the-art SDDP (261,800 parameters), highlighting its efficiency and suitability for memory-constrained environments.



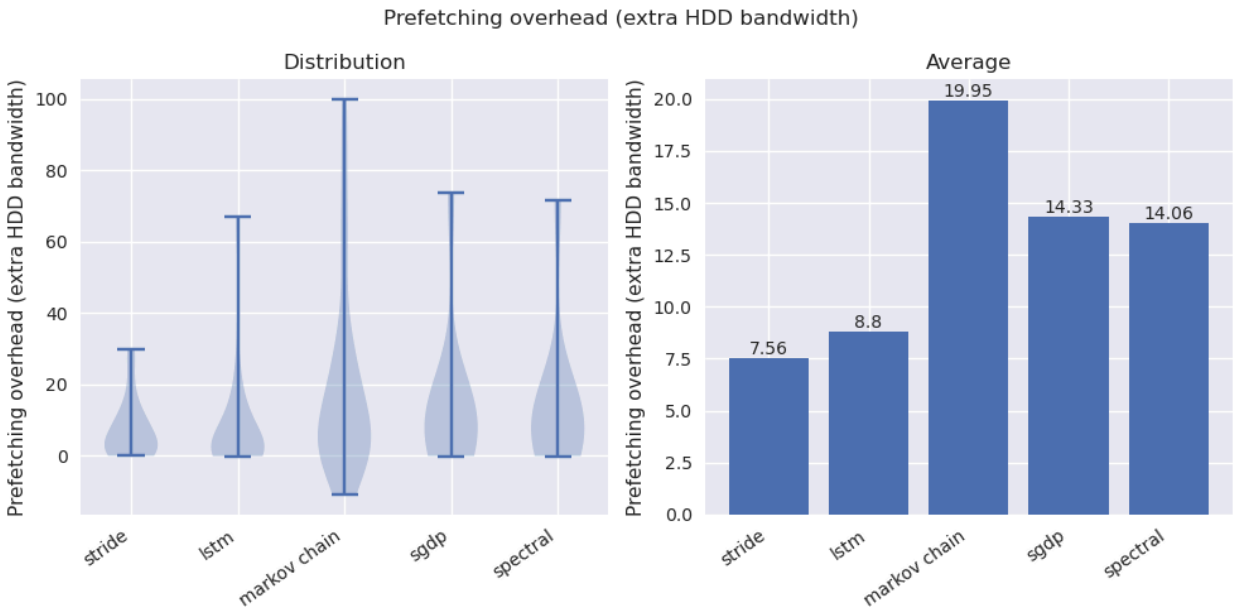
03 Spectral Prefetcher improves hit rate by 21.8%. In comparative analysis, the Spectral Prefetcher reached a hit rate of 31.94%, surpassing the baseline stride prefetcher's 10.1%. Graph-based prefetchers, including the Spectral Prefetcher, demonstrate superior performance over sequential ones, validating the effectiveness of graph-based algorithms in cache prediction. This improvement is reflected across various company traces, with graph-based models achieving higher average hit rates.



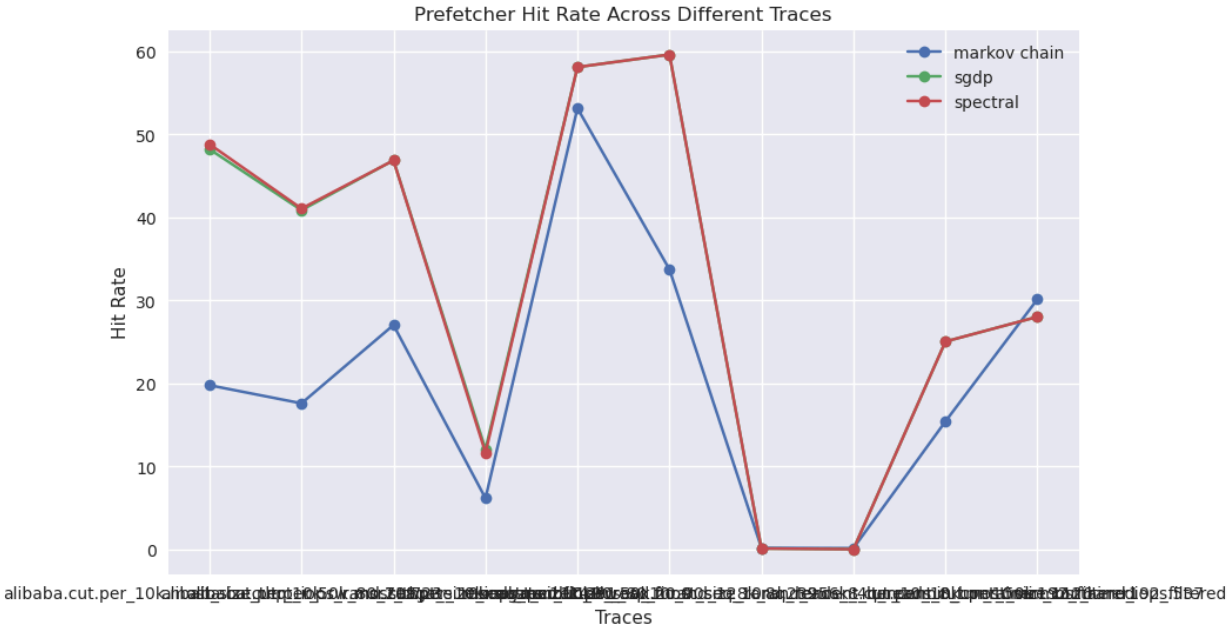
04 Spectral Prefetcher improves memory utilization. By improving memory utilization, the Spectral Prefetcher outperforms other models, notably the stride prefetcher, which suffers from fetching many incorrect blocks into the cache. While LSTM and other graph-based approaches show similar memory utilization levels, the Spectral Prefetcher stands out as the top performer, indicating its precision in prefetching relevant data.



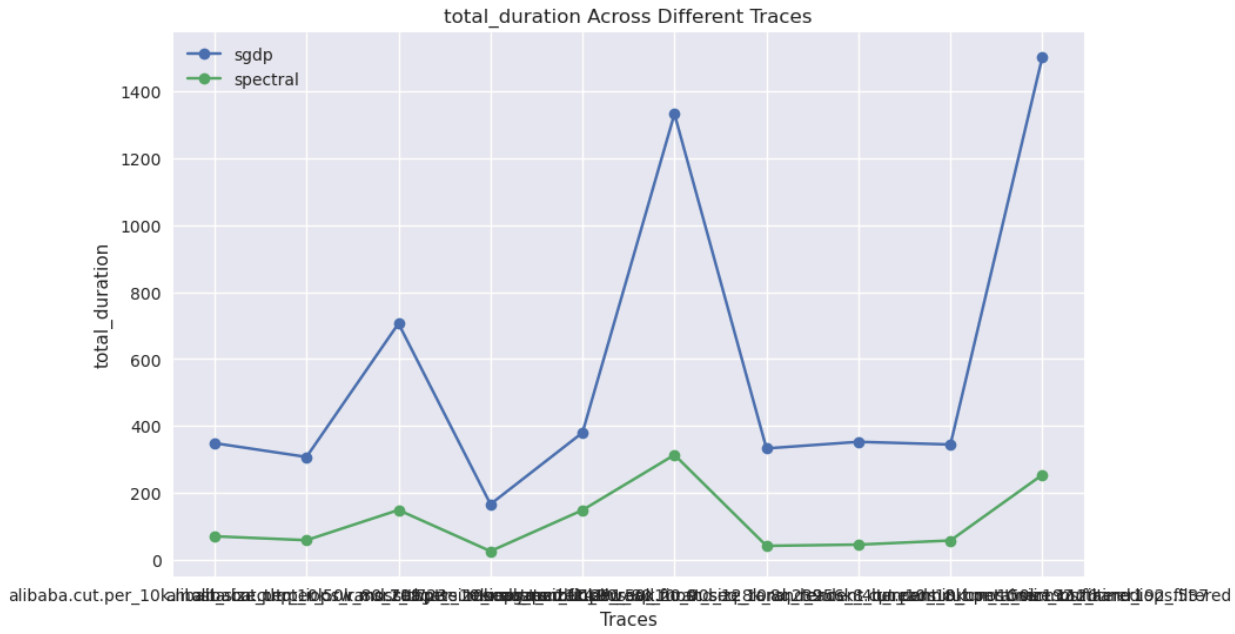
05 Spectral Prefetcher maintains reasonable overhead. Despite generally higher overhead observed among graph-based models, indicating a tendency to prefetch more items into the cache, the Spectral Prefetcher maintains a reasonable overhead. This is less of a concern with sufficient cache size or when prefetching occurs in the background without competing for I/O bandwidth with ongoing requests. The Spectral Prefetcher exhibits lower overhead compared to both SGDP and the Markov Chain, underscoring its efficiency.



06 Spectral's high hit rate across all traces. Across a range of workloads, each with its own unique patterns, the Spectral Prefetcher consistently achieves the highest hit rate, as evidenced by the red line in our analyses. This consistency in performance across diverse workloads highlights the Spectral Prefetcher's robustness and adaptability in various environments.



07 Spectral Prefetcher’s low latency across all traces. The Spectral Prefetcher ensures low latency across all traces, accommodating both lengthy and brief workloads effectively. This is illustrated by the green line in our analysis, which shows the Spectral Prefetcher's ability to maintain significantly lower latency across different types of workloads, further demonstrating its potential for enhancing performance in a wide range of cloud computing scenarios.



Section 6: Conclusion

In this paper, we introduced the Spectral Prefetcher, an innovative spectral graph-based model designed to accurately predict future I/O accesses by users. The Spectral Prefetcher has demonstrated superior performance, achieving state-of-the-art results in hit rate, memory utilization, and overhead management. Notably, it offers a substantial 79.9% reduction in latency and a 33.3% decrease in model size. Its efficacy has been validated across diverse workloads from major cloud companies, including Microsoft and Alibaba. The Spectral Prefetcher's design makes it a practical solution for deployment in production storage servers, where it can deliver significant memory savings and enhanced cache performance.

References

[1] J. Liao, F. Trahay, B. Gerofi, and Y. Ishikawa, “Prefetching on storage servers through mining access patterns on blocks,” IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 9, pp. 2698–2710, 2015.

[2] J. W. Fu, J. H. Patel, and B. L. Janssens, “Stride directed prefetching in scalar processors,” ACM SIGMICRO Newsletter, vol. 23, no. 1-2, pp. 102–110, 1992.

[3] C. Chakraborttii and H. Litz, “Learning i/o access patterns to improve prefetching in ssds,” ICML-PKDD, 2020.

[4] P. G. Harrison, S. Harrison, N. M. Patel, and S. Zertal, “Storage workload modelling by hidden markov models: Application to flash memory,” Performance Evaluation, vol. 69, no. 1, pp. 17–40, 2012.

[5] Y. Yang, R. Li, Q. Shi, X. Li, G. Hu, Xing Li, M. Yuan. “SGDP: A Stream-Graph Neural Network Based Data Prefetcher.” Available at https://arxiv.org/abs/2304.03864, 2023.

[6] G. O. Ganfure, C.-F. Wu, Y.-H. Chang, and W.-K. Shih, “Deep- prefetcher: A deep learning framework for data prefetching in flash storage devices,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3311–3322, 2020.

[7] G. Soundararajan, M. Mihailescu, and C. Amza. “Context-Aware Prefetching at the Storage Server.” In ATC'08: USENIX 2008 Annual Technical Conference, pages 377–390, June 2008.

[8] MSRC. Microsoft Research Cambridge. http://iotta.snia.org/traces/388.

[9] H. Kim and U. Ramachandran, “Flashfire: Overcoming the performance bottleneck of flash storage technology,” Georgia Institute of Technology, Tech. Rep., 2010.

[10] H. Maruf and M. Chowdhury. “Effectively Prefetching Remote Memory with Leap.” In Proceedings of the USENIX Annual Technical Conference (USENIX ATC), 2020.

[11] J. Griffioen and R. Appleton, “Reducing file system latency using a predictive approach,” Proc. USENIX Summer 1994 Technical Conf., Jun. 6–10 1994.