Learning-Based Prefetching Strategy

Rizky Ramadhana Putra Kusnaryanto
rizky@vt.edu
Virginia Polytechnic Institute State University
Blacksburg, Virginia, USA

Hansen Idden
hansenidden@vt.edu
Virginia Polytechnic Institute State University
Blacksburg, Virginia, USA

Nathan Roberts
nathan03@vt.edu
Virginia Polytechnic Institute State University
Blacksburg, Virginia, USA

Hyun Myung Kang
dkang1013@vt.edu
Virginia Polytechnic Institute State University
Blacksburg, Virginia, USA

ABSTRACT

The main challenge in the modern data-intensive applications is when the working sets exceed available memory capacity. While prefetching techniques can mask access latency by proactively loading data into faster memory tiers, developing effective prefetching strategies remains challenging due to the need to accurately predict future data requirements while efficiently utilizing limited memory resources. Traditional prefetching approaches rely on fixed heuristics or simple pattern detection, which often fail to capture the complex I/O behaviors exhibited by modern applications. We propose a novel learning-based prefetching strategy that leverages neural networks to identify and adapt to complex access patterns. Our approach formulates prefetching as a regression task, predicting both the offset (where to prefetch) and size (how much to prefetch) of future I/O requests. We implement three neural network models: one predicting offset, one predicting size, and one predicting both simultaneously. Through rigorous evaluation against state-of-the-art prefetching techniques, including stride-based and Markov Chain algorithms, our models demonstrate superior performance, achieving up to 51× improvement over stride-based prefetchers and up to 15.4× improvement over Markov Chain-based approaches. Our best-performing model achieves 78.7% prefetch coverage and 26.1% hit ratio with a buffer size of 1000, significantly outperforming traditional methods. These results demonstrate the potential of learning-based approaches to revolutionize prefetching strategies for modern storage systems.

ACM Reference Format:
Rizky Ramadhana Putra Kusnaryanto, Hansen Idden, Nathan Roberts, and Hyun Myung Kang. 2018. Learning-Based Prefetching Strategy. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Modern computing systems have difficulty dealing with the storage performance gap, where storage access latencies are a lot slower than CPU speeds and DRAM access. Even with high-speed SSDs, a random read can consume in the order of 50–100 µs [8], about 10^3 times slower than an L1-cache hit and three orders slower than a DRAM access. Memory-disaggregation fabrics help by exposing larger pools, but even state-of-the-art CXL and RDMA setups report remote-page latencies of tens of microseconds, leaving the fundamental storage-performance gap largely intact [1]. Thus, I/O is still a critical bottleneck in various fields. Prefetching is known as a tool to bridge the gap of the delayed latency by anticipating future data accesses and loading data into faster memory ahead of time [3]. An effective prefetcher can mask much of the SSD access latency, improving throughput and reducing application stall time. Designing a high-accuracy prefetcher remains a challenge. Unlike simple sequential memory access patterns, real-world I/O workloads exhibit complex, non-linear access patterns across vast address spaces. Specifically, flash-based SSDs operate over huge logical block address ranges that are often sparse, making it difficult to predict the next accessed block. Additionally, Prefetching must be done at the right time, as the data should arrive right before it is needed, meaning the predictor must accurately predict when and how big of data to fetch [3].

Conventional prefecting techniques struggle to tackle this problem effectively. Simpler heuristics, such as sequential read-ahead or stride detection, work well for linear access patterns but struggle when accessing irregular or data-dependent patterns. Traditional prefecters often assume a limited working set or a small set of repeating patterns; they cannot predict the complex correlations in large data storage. The precision decreases when strides overlap, the address space grows into billions of logical blocks, and the table sizes grow beyond practical limits [4]. More complex profile-guided engines like vRPG2 raise CPU-cache hit rates with run-time-injected prefetches, but they operate on program counters and offer little assistance once requests spill past DRAM into the block layer [20]. When these heuristics mis-speculate, they face a drastic trade-off: small settings let stalls bleed through, while aggressive settings over-fetch, wasting useful data and saturating bandwidth.

Machine-learning approaches promise a more complex understanding of access behavior. Google's *Voyager* neural prefetcher lifted temporal-coverage rates from 51 % to nearly 74 % on complex SPEC workloads by using a hierarchical sequence model [16], while CACHEUS applied learning to cache-replacement decisions and surpassed well-tuned LIRS and ARC across diverse server traces [14]. Yet most published machine-learning prefetchers demand tens

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

of megabytes of weights and milliseconds-scale interface budgets, making them impractical for SSD controllers that typically house only a few megabytes of SRAM and sub-GHz ARM cores [5].

To meet those deployability constraints while retaining machine learning's predictive power, we deployed a dual-head LSTM prefetcher. A single lightweight LSTM backbone ingests recent address-delta tokens and I/O size data, then incorporates into two cooperative heads: one classifies the next offset delta—pinpointing *where* the next access will land—while the other regresses request size and lead time—specifying *how much* data to fetch and exactly *when* it must arrive. Careful dimensioning and quantization shrink the entire network to 30 KB. Because the model is compact, we can refine it online, allowing the prefetcher to track phase shifts and mixed-tenant contention without offline retraining.

2 RELATED WORK

Prefetching aims to minimize memory latency by anticipating future data accesses. Over decades, researchers have explored heuristic, machine learning, and adaptive control approaches. Each category balances prediction accuracy, hardware cost, and adaptability differently—trade-offs that show a clear direction for a regression-based neural design. Early prefetching mechanisms relied on simple heuristics to predict future memory accesses which exploit edsimple spatial or stride patterns by fetching blocks ahead of demand. For example, Palacharla and Kessler evaluate stream buffers that cache sequential lines evicted from the cache [13], and Michaud's Best-Offset (BO) prefetcher dynamically tracks high-scoring address deltas to predict future accesses [11]. Correlation-based schemes such as Nesbit and Smith's PC/DC prefetcher (using a global history buffer) and Joseph and Grunwald's Markov predictor learn from recent delta streams [12, 7]. These heuristic techniques are lightweight and hardware-friendly, but rely on fixed rules: they work well for regular or moderately irregular streams, yet they generally fail to capture highly irregular access patterns or adapt to changing behavior.

Recent work replaces fixed heuristics with learned predictors. Hashemi et al. were among the first to apply LSTM networks for prefetching by treating memory access prediction as a classification problem over address deltas [6]. Building on this idea, Shi et al. (Voyager) use a hierarchical LSTM+attention model that jointly predicts an address page and its offset from past accesses [15]. Zhang et al. (TransFetch) employ a Transformer to generate multiple future addresses via fine-grained address segmentation [19]. Bapat et al. (Pythia) develop a neural model for database buffer prefetching in SQL workloads [2]. More recently, Zhang et al. (DART) train a large neural prefetcher and then distill it into a compact table-based predictor for efficient hardware deployment [18]. These neural approaches can learn complex, irregular patterns beyond the reach of simple stride prefetchers. However, they typically require offline training and large models, and their outputs are inherently discrete (one class or label per address or delta) [6, 18]. This large output space and model size complicate implementation and can limit practical performance.

Some works apply reinforcement learning (RL) to adapt prefetching at runtime. For instance, Yang et al. propose RL-CoPref, an RL-based controller that adjusts multiple prefetchers' activation and degree by observing program context and cache behavior [17]. RL-CoPref learns online to maximize a reward (e.g., IPC or hit rate) and shows strong adaptability to changing workloads. In general, RL methods can automatically tune prefetching strategies to diverse patterns, but they introduce overhead: learning agents require on-chip state and training (or inference) logic, and designing effective rewards and states is complex.

In summary, traditional prefetchers are fast but inflexible, neural predictors are powerful but heavyweight and rely on classification outputs, and RL controllers adapt at runtime but add complexity. These trade-offs motivate alternate formulations. Our work treats address prediction as a regression problem: the neural model directly outputs continuous address or offset values. By avoiding a large discrete output layer, a regression-based prefetcher can reduce model complexity while still learning nonlinear access patterns, offering a new balance between coverage and efficiency.

3 METHOD

3.1 Data Exploration

The SSD tracing dataset [9] basically has six columns. First, timestamp that describes when the requests are arrived. Second, response that describes how long a request is completed. Third, IOType that describes the kind of request, whether read or write. Fourth, LUN that describes on which device a request is sent. Fifth, offset that describes the location of a request. Lastly, size that describes how many bytes are requested.

It is clearly shown in Figure 1 that I/O requests form some patterns. Subsequent I/O requests tend to land on neighboring locations. These patterns are the ones that are being predicted in this project.

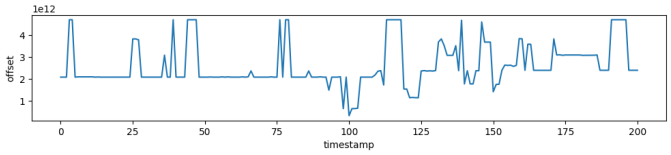

Figure 1: I/O Request Pattern

3.2 Data Preprocessing

There are several important steps to do before training the model. First, scale the dataset as each column varies in scale significantly. As an illustration, the size column has scale of thousands and the offset column has scale of trillions. To solve this problem, MinMaxScaler is used to clip the value of all features to between 0 and 1.

Second, create lagging features to capture the temporal relationship. The fundamental intuition is to predict next I/O request based on previous I/O requests. So, we decided to create lagging features that consider previous five I/O requests.

3.3 Model Training

After preprocessing the dataset, we have 20 features that come from 5 previous I/O request with each 4 features being considered–IO

size, timestamp, type, and offset. We picked lightweight neural network model with two hidden layers, each with 16 and 4 neurons, and one output layers with one neurons. We picked ReLU as the activation function in the hidden layers so that the model can capture non-linear relationship. For the output layer, we picked linear as our activation function.

As shown in Figure 2, the training and validation loss follow a nice curve towards more epoch. It indicates a good learning process with no sign of underfitting or overfitting. However, this measured loss is meaningless since we only measure the mean average error of predicted offset and real request offset. To get more realistic performance description, we should run a simulation that is described in the next section.

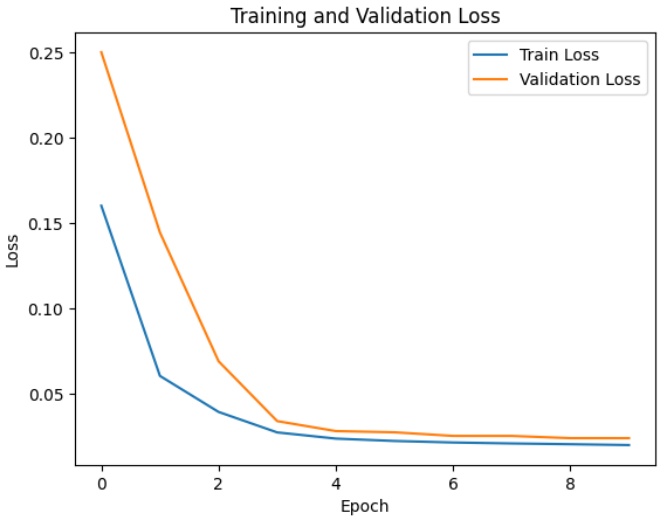

Figure 2: I/O Request Pattern

4 EVALUATION

We evaluated our Neural Network-based prefetching approach against conventional prefetcher, Markov chain algorithm and stride-based algorithm, using a block I/O trace of 12,724 I/O accesses. We chose to compare our approach with Markov chain and stride because those 2 algorithm are the most common prefetching algorithms used by system with a decent performance. We tested each of the algorithm with 10, 100, and 1000 buffer size, meaning we can hold 10, 100, and 1000 blocks in the prefetcher. Since in real systems, buffer size represents a trade-off between performance and resource utilization. The ideal prefetcher achieves high hit ratios even with small buffers, which means maximizing performance while minimizing resource consumption. We measured prefetch coverage (percentage of accesses for which an I/O being prefetched) and hit ratio (percentage of accesses that were successfully prefetched).

4.1 Prefetch Coverage

In the Figure 3 shows the prefetch coverage of each approach. Our approaches achieve substantially higher coverage, with the

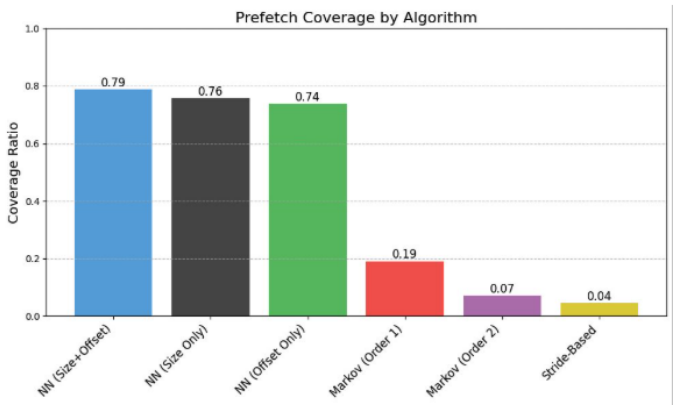

Figure 3: Prefetch Coverage Comparison Between Our Approach VS. Conventional Approach

Size+Offset model achieving 78.7% coverage, followed by the Size-only model at 75.7%, and the Offset-only model at 73.9%. This significantly outperforms the conventional approaches that 19.1$ coverage for Markov Order 1, 7.1% coverage for Markov Order 2, and 4.5% for stride-based prefetching. We can see the contrast of our models coverage achieving up to 16.5× higher coverage, highlights the fundamental limitation of conventional prefetchers in identifying prefetchable patterns within complex I/O workloads.

The huge difference in coverage reveal an important findings about each algorithm's pattern recognition capabilities. Traditional stride-based prefetcher, they have a limit to detecting constant stride patterns, identify prefetching opportunities in less than 5% of whole accesses. This indicates that regular sequential patterns that the stride-based algorithm stored only represent a small fraction in our workload. First-order Markov chains perform better by capturing simple transition probabilities but it still miss a lot of details over 80% of the potential prefetch opportunities. We also observe that for Second-order Markov the prefetching coverage drop from First-order Markov, from 19.1% coverage to 7.1% coverage), this finding suggest that longer historical context of the accesses is actually reducing the model's ability to make decent predictins, likely due to the state space explosion and data sparsity issues of the trace. In contrast, our Neural Network models maintain high coverage regardless of the complexity of the sequence, demonstrating the ability to learn from longer sequence without being punish from the limitations of Markov Chain statistical approaches. The Size+Offset model's higher coverage compared to our other models indicates that leveraging both parameters provides additional context for identifying prefetch opportunities.

4.2 Hit Ratio Across Buffer Sizes

Figure 4 presents the hit ratio achieved by each prefetching algorithm across three buffer sizes to clearly shows the effectiveness of the number of prefetched blocks in anticipating future I/O requests. At the smallest buffer size, 10 blocks, our models demonstrates hit ratios between 10.7-13.7%, which already significantly outperform the other algorithm. Where first-order Markov only able to achieved

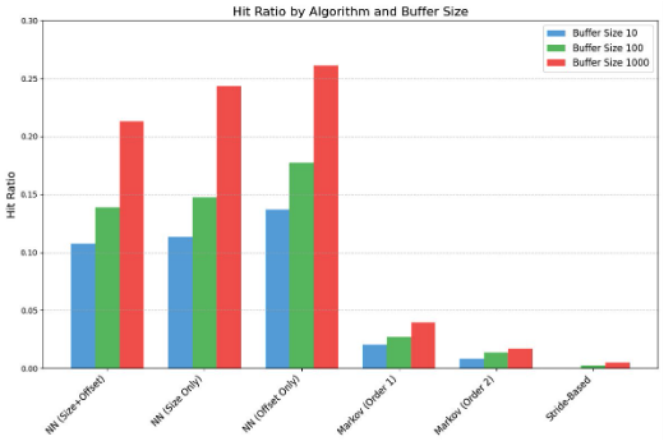

Figure 4: Hit Ratio Comparison Between Our Approach VS. Conventional Approach

2%, second-order Markov achieved 0.8%, and stride-based prefetching will not even prefetch any I/O access. As the buffer size increases to 1000 blocks, the performance gap widens drastically. Our best model in terms of hit ratio, the Offset-only model, achieved a 26.1% hit ratio compared to just 4% for Markov Order 1, 1.7% for Markov Order 2, and 0.5% for stride-based prefetching. This shows 6.6× to 51.2× improvement over conventional methods, which clearly shows our models ability to not only issues more prefetches, as shown in Figure 3, but to issue more accurate accesses.

While all prefetchers benefit from increased buffer sizes, our models shows the strongest scaling-with up to 51.2×. This indicates that our Neural Network predictions remain valuable over longer time, capturing both short-term and long-term I/O patterns.

What particularly interesting is our Offset-only model. Despite having a lower coverage, the Offset-only model able to achieves the highest hit ratios across all buffer sizes, suggesting that precise offset prediction is the most important feature for this workload.

5 CONCLUSION

In this project, we successfully implemented and evaluated a machine learning based prefething strategy for SSD storage systems that can significantly outperforms traditional techniques. Our neural network models effectively predict future I/O access patterns by learning the relationships of the previous offset and size of the storage requests.

Despite the challenges of predicting both the offset (location) and size of the future storage requests with the enormous address space, our models demonstrated remarkable accuracy and efficiency compared to the conventional prefetching algorithm. We developed three approaches in this project-predicting offset only, size only, and both offset and size-each showing significant improvements over the conventional prefetching approach.

Our implementation achieves up to 51× better performance than stride-based prefetching algorithm and up to 15.4× better performance than Markov Chain-based prefetching algorithm approaches. The high prefetching coverage, around 78%, combine with high hit ratios, up to 26%, demonstrates that our models efficiently utilize prefetching resources.

These result confirms the viability of applying machine learning techniques to storage systems, particularly for prefetching mechanism challenges that conventional heuristic-based approaches struggle to address. our work establishes a foundation for future research into learning-based storage optimization that can adapt to diverse and evolving workload.

6 FUTURE WORKS

Based on our promising results, we identify several important directions for future research.

SSD Emulator Implementation Our first goal is to deploy our model in FEMU SSD emulator environment [10]. However, FEMU does not have any caching mechanism within the emulator to utilize our prefetcher. To do so, we need to develop our custom caching system inside the emulator. By testing it on an emulator, it will provide insights into real-world performance characteristics and system integration challenges for our approach.

Real System Deployment Given the 51× improvement over a conventional stride-based algorithm and 15.4× improvement over a conventional Markov-chain-based algorithm, we aim to implement our prefetching strategy in a real storage system. This will allow us to evaluate our approach in a real system while carefully measuring and minimizing the latency overhead.

Workload-Specific Optimization Our current model was trained on general block I/O traces from a server. we plan to explore workload-specific optimization, particularly on machine learning workloads. By training our models on I/O traces from Machine Learning model training and inference operations, we may accelerate these workloads by exploiting their unique I/O patterns. This specialized approach could provide significant performance benefits for AI systems with intensive storage requirements.

REFERENCES

[1] Hasan Al Maruf and Mosharaf Chowdhury. 2023. Memory disaggregation: advances and open challenges. 57, 1, (June 2023), 29–37. DOI: 10.1145/3606557.3606562.

[2] Akshay A. Bapat, Saravanan Thirumuruganathan, and Nick Koudas. 2025. Pythia: a neural model for data prefetching. In *Proc. 28th Int'l Conf. on Extending Database Technology (EDBT)*. OpenProceedings.org, 384–396. DOI: 10.48786/EDBT.2025.31.

[3] 2021. *Learning i/o access patterns to improve prefetching in ssds*. (Feb. 2021), 427–443. ISBN: 978-3-030-67666-7. DOI: 10.1007/978-3-030-67667-4_26.

[4] Chandranil Nil Chakraborttii and Heiner Litz. 2022. Deep learning based prefetching for flash. In https://api.semanticscholar.org/CorpusID:248883763.

[5] Quang Duong, Akanksha Jain, and Calvin Lin. 2024. A New Formulation of Neural Data Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, (July 2024), 1173–1187. DOI: 10.1109/ISCA59077.2024.00088.

[6] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *Proc. 35th Int'l Conf. on Machine Learning (ICML)*. PMLR, 1919–1928.

[7] Doug Joseph and Dirk Grunwald. 1999. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48, 2, 121–133. DOI: 10.1109/12.752653.

[8] Piyush Kashyap. 2024. Understanding latency numbers: a practical guide to system design. (Nov. 2024). https://medium.com/@piyushkashyap045/understanding-latency-numbers-a-practical-guide-to-system-design-55fa7951cc49.

[9] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2016. Systor '17 traces (SNIA IOTTA trace set 4964). In *SNIA IOTTA Trace Repository*. Geoff Kuenning, (Ed.) Storage Networking Industry Association, (Mar. 2016). http://iotta.snia.org/traces/block-io/4928?only=4964.

[10] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. 2018. The case of femu: cheap, accurate, scalable and extensible flash emulator. In *Proceedings of 16th USENIX Conference on File and Storage Technologies (FAST)*. Oakland, CA, (Feb. 2018).

[11] Pierre Michaud. 2016. Best-offset hardware prefetching. In *Proc. IEEE Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 469–480. DOI: 10.1109/HPCA.2016.7446087.

[12] Kyle J. Nesbit and James E. Smith. 2004. Data cache prefetching using a global history buffer. In *Proc. 10th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 96–105. DOI: 10.1109/HPCA.2004.10030.

[13] Subbarao Palacharla and Richard E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proc. 21st Int'l Symp. on Computer Architecture (ISCA)*. IEEE Computer Society, 24–33. DOI: 10.1109/ISCA.1994.288164.

[14] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, (Feb. 2021), 341–354. ISBN: 978-1-939133-20-5. https://www.usenix.org/conference/fast21/presentation/rodriguez.

[15] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proc. 26th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 861–873. DOI: 10.1145/3445814.3446752.

[16] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '21). Association for Computing Machinery, Virtual, USA, 861–873. ISBN: 9781450383172. DOI: 10.1145/3445814.3446752.

[17] Huijing Yang, Juan Fang, Xing Su, Zhi Cai, and Yuening Wang. 2024. Rl-copref: a reinforcement learning-based coordinated prefetching controller for multiple prefetchers. *Journal of Supercomputing*, 80, 9, 13001–13026. DOI: 10.1007/s11227-024-05938-9.

[18] Pengmiao Zhang, Neelesh Gupta, Rajgopal Kannan, and Viktor K. Prasanna. 2024. Attention, distillation, and tabularization: towards practical neural network-based prefetching. *CoRR*, abs/2401.06362. DOI: 10.48550/ARXIV.2401.06362.

[19] Pengmiao Zhang, Ajitesh Srivastava, Anant V. Nori, Rajgopal Kannan, and Viktor K. Prasanna. 2022. Fine-grained address segmentation for attention-based variable-degree prefetching. In *Proc. 19th ACM Int'l Conf. on Computing Frontiers (CF)*. ACM, 103–112. DOI: 10.1145/3528416.3530236.

[20] Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. 2024. Rpg2: robust profile-guided runtime prefetch generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ASPLOS '24). Association for Computing Machinery, La Jolla, CA, USA, 999–1013. ISBN: 9798400703850. DOI: 10.1145/3620665.3640396.