THE UNIVERSITY OF CHICAGO

ADVANCED STORAGE OPTIMIZATIONS FOR DEEP RECOMMENDATION SYSTEMS

A DISSERTATION SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES

IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

DANIAR HERI KURNIAWAN

CHICAGO, ILLINOIS

JUNE 2024

Dedicated to Ilma, Razin, Zayn, and my family (Bapak Sumiran, Ibu Muslimah, Azam, and Haki) in Durenan-Trenggalek.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Alhamdulillah. This Ph.D. journey has been made possible through the generous support and guidance of numerous individuals, to whom I extend my heartfelt appreciation and thanks in this special section.

First and foremost, I am profoundly grateful to have Prof. Haryadi Gunawi as my advisor. Under his guidance, I came to realize that conducting research and solving problems is satisfying and fulfilling. He also inspired me to become a great mentor; that's why I mentored 10+ students throughout my Ph.D. career, and it is indeed very fulfilling to see someone grow and improve their skill set. During my time doing research with him, he often asked for more results and evaluations. From this, I learned that the journey will never end. There is always a bigger goal or question in life waiting for us to explore.

To my committee members, Prof. Hank Hoffmann and Prof. Ymir Vigfusson, thank you for the feedback on my dissertation that has made it much stronger. Hank provided great feedback about Heimdall and TinFetch, which led to a stronger paper positioning. Ymir helped me a lot with EVStore, leading to its acceptance in ASPLOS. Ymir is also such a role model for business-minded academia, as he has founded many startup companies while excelling in his research career as a professor at Emory University. During my Ph.D. defense, Ymir asked me, "What have you learned from your Ph.D. journey?" I said, "Having blurry vision is very important," meaning that we shouldn't be fixated solely on achieving a specific outcome, but rather concentrate on the process, refining our vision as we progress.

To UCARE members, (alphabetically ordered): Cesar Stuardo, Huaicheng Li, Huan Ke, Jeffery Lukman, Mingzhe Hao, Meng Wang, Martin Lutta Putra, Ray Andrew, Riza Suminto, Roy Huang, Ruidan Li, and Tanakorn Leesatapornwongsa. Thank you for your friendship and for engaging in many insightful and spontaneous discussions with me on various topics, which have helped me to grow as an individual and broaden my perspective on different subjects. I always seek input from UCARE students before consulting Haryadi, so your contributions have been invaluable to me.

To my friends at University of Chicago and Institut Teknologi Bandung: Saad Ansari, Aris Wakhyudin, Setyo Legowo, Vincentius Martin, Muhammad Iqbal Rochman, Muhammad Husni Santriaji, Bogdan Alexandru Stoica, Neng Huang, Xu Zhang, Tushant Mittal, etc. Thank you for the many insightful conversations we shared and the fun we had together.

I am fortunate to have mentored several outstanding undergraduate and master students: Rex Wang, Fandi Azam Wiranata, Maharani A. P. Irawan, Kahfi Zulkifli, Dimas Shidqi Parikesit, Peiran Qin, Fawadawn Zeyuan Yang, Hongzhen Liang, etc.. Thank you for your contributions to the research projects!

I would also like to express my gratitude to the many other collaborators, mentors, and friends with whom I have enjoyed working. I have been very fortunate to have been a three-time research intern at Seagate throughout my Ph.D., working with many smart people: John Bent, Greg Larrew, Lingzhi Yang, and James Fantin-Hardesty. John and Greg are among the best managers one could have. I also spent a few months working with Junchen Jiang and Siddharta Sen of Microsoft Research on the E2E project. I enjoyed many insightful discussions with them. Additionally, I worked at VMware Research Group under Mihai Budiu's mentorship and learned first-hand about the complexity of managing open-source software with fully integrated components. Many thanks to everyone who helped me along the way and gave me the opportunity to be part of their teams.

Lastly, I want to express my gratitude to my big family for their unconditional love. They trust me and give me the opportunity to explore something that seems so risky and random. We all believe that the harder the challenge, the greater the reward. I dedicate this dissertation to them.

ABSTRACT

With recommendation systems playing an increasingly pivotal role in guiding user decisions online, their impact on user engagement cannot be overstated. Recent studies show that a significant amount of content—30% of all traffic on Amazon's website, 60% of the videos on YouTube, and 75% of the viewed movies on Netflix came from suggestions made by recommendation algorithms. As users become more reliant on these systems, they expect higher quality recommendations that are tailored to their individual preferences. To meet this demand, a Deep Recommendation System (DRS), which is a neural network-based recommendation algorithm, must increase the size of its embedding vector tables (EV tables) to encode richer semantic relationships between input features. This has led to a tripling of EV table sizes every two years (1.5× annual growth). Consequently, the space management of EV tables becomes challenging: many real-world EV tables contain billions of embedding vectors that require tens of TBs of memory capacity. Such DRAM-heavy architectures account for significant operational costs for DRS users measured in millions of dollars—nearly 80% of all AI-related deployments in Meta's data centers in 2020 directly supported DRSs. A natural solution to this problem is by moving the large EV tables to the backend/secondary storage (*i.e.*, SSDs). However, it can introduce performance instability, particularly for large-scale DRSs tasked with meeting stringent microsecond-scale tail latency Service Level Objectives (SLOs). The current trend of microservices and machine learning deployment further exacerbates these challenges, with tail-latency SLOs expected to become even tighter over time. Additionally, achieving SSDs with deterministic latency remains challenging due to the unpredictable behavior of internal activities such as garbage collection, wear leveling, and internal buffer-flush.

The storage industry and research community have dedicated significant efforts to address the issue of unpredictable latency in SSDs. Various approaches, including white-box, gray-box, and black-box techniques, have been proposed to mitigate this challenge. While each approach offers unique advantages and trade-offs, the optimal evolution of the storage stack to accommodate

the evolving needs of deep recommendation systems remains a critical area of exploration. In this dissertation, we seek to answer the question: How should the storage stack evolve to meet the growing demands for low and predictable latencies and cost-efficiency in the context of the burgeoning usage of deep recommendation systems? To support this statement, we develop a set of solutions each delving into distinct facets of improving storage supports mechanisms for deep recommendation systems.

We start developing our solutions from the inference layer of the deep recommendation system, where the performance variance originates. This layer is critical as it directly impacts the user experience by determining the relevance and accuracy of the recommendations provided. By focusing on optimizing this foundational layer, we aim to enhance the overall performance and efficiency of the recommendation system. To realize this goal, we built EVSTORE: A caching system that exploits groupability pattern between items for scaling and enabling cost-efficient recommendation system deployment. EVSTORE's main contributions lie in EVSTORE's 3-layer "L1-to-L3" caching design (EVCache, EVMix, and EVProx). We have fully integrated EVSTORE within Facebook (Meta)'s DLRM, including various implementation-level optimizations and offline supporting tools ($\approx$9 KLOC) that are released publicly. Our evaluation based on real production DRS traces shows that EVSTORE can reduce the average and p90 latency by up to 23% and 27% respectively, while increasing the throughput by 4$\times$ at only 0.2% accuracy reduction. Collectively, fully optimized EVSTORE implementation can achieve a 94% reduction of the DRS memory footprint. These memory savings correspond to hundreds of millions of dollars for a large cloud provider.

Although EVSTORE's caching layers facilitate a performant and cost-efficient deployment of deep recommendation systems, its reliance on SSDs as backend storage can introduce performance instability due to unpredictable internal SSD activities such as garbage collection (GC), wear leveling, and buffer-flush. These background activities can disrupt user I/O requests and lead to tail latencies. For instance, GCs alone can cause up to a 60$\times$ increase in latency, easily violating SLOs. To tackle this challenge, we developed HEIMDALL: an accurate and efficient I/O admission policy

for flash storage empowered by an extensive ML pipeline. We make domain-specific innovations in various ML stages by introducing accurate period-based labeling, 3-stage noise filtering, in-depth feature engineering, and fine-grained tuning, which together improve the decision accuracy from 67% up to 93%. We perform various deployment optimizations to reach a sub-µs inference latency and a small, 28KB, memory overhead. With 500 unbiased random experiments derived from production traces, we show HEIMDALL delivers 15-35% lower average I/O latency compared to the state of the art and up to 2× faster to a baseline. HEIMDALL is ready for user-level, in-kernel, and distributed deployments.

With HEIMDALL tackling the tail latency issue, we target further enhancements in SSD performance stability by improving the read latency. Our solution, TINFETCH, presents a Tiny Neural Network Prefetcher that utilizes an adaptive heuristic method to understand complex I/O streams and leverages a pretrained neural network to predict future LBA accesses. Evaluated on various production and synthetic traces, TINFETCH achieves a 3% to 27% improvement in hit rate compared to other prefetchers. Moreover, it has the highest performance score, hit rate per storage bandwidth load, surpassing the best state-of-the-art heuristic- and learning-based prefetchers by 1.5x and 2.6x respectively. Additionally, the low-overhead and optimized implementation of TINFETCH on a C/C++ language is capable of achieving inference latency in sub-μs on a consumer-level CPU node.

CHAPTER 1

INTRODUCTION

With recommendation systems now deeply integrated into modern online platforms, guiding users in their decision-making processes, their importance cannot be overstated. These systems employ a complex algorithm that plays a crucial role in influencing user engagement by presenting personalized advertisements, ranking news articles, suggesting products, and others. Research shows that a substantial portion of content consumed on major platforms like Amazon, YouTube, and Netflix originates from recommendations made by these algorithms.

Despite the effectiveness of deep recommendation systems (DRSs) in delivering high-quality recommendations, they encounter challenges in handling sparse categorical input features. For instance, Facebook's post recommendation platform often struggles with managing numerous categorical features, each potentially encompassing millions or billions of categories. To simplify the complexity of deep neural networks (DNNs), sparse categorical data is typically converted into dense vectors through embedding vector tables (EV tables). However, this conversion reduces DNN complexity while posing challenges in space management and incurring significant operational costs associated with memory-intensive architectures.

The demand for enhanced recommendation accuracy necessitates larger EV tables to encode richer semantic relationships. However, current state-of-the-art DRSs are ill-equipped to handle the exponential growth in EV table sizes. Although some open-source DRS platforms store full EV tables in DRAM, this approach presents drawbacks, including reduced resource utilization and increased operational costs. Consequently, there is growing interest in migrating large EV tables to SSDs as backend storage, prompting recent research to focus on optimizing backend storage for EV table lookups. However, existing storage solutions face limitations in adoption due to the requirement for customized devices such as custom SSDs or FPGA implementations.

While incorporating off-the-shelf SSDs to reduce DRAM footprints may seem cost-effective, it can introduce performance instability, particularly for large-scale DRSs tasked with meeting

stringent microsecond-scale tail latency Service Level Objectives (SLOs). Moreover, the current trend of microservices and Machine Learning deployment exacerbates these challenges, with tail-latency SLOs are expected to become even tighter over time. Additionally, achieving SSDs with deterministic latency remains challenging due to the unpredictable behavior of internal activities such as garbage collection, wear leveling, and internal buffer-flush.

The storage industry and research community have dedicated significant efforts to address the issue of unpredictable latency in SSDs. Various approaches, including white-box, gray-box, and black-box techniques, have been proposed to mitigate this challenge. While each approach offers unique advantages and trade-offs, the optimal evolution of the storage stack to accommodate the evolving needs of recommendation systems remains a critical area of exploration.

1.1 Thesis Statement

In this dissertation, we seek to answer the question: *How should the storage stack evolve to meet the growing demands for low and predictable latencies and cost-efficiency in the context of the burgeoning usage of deep recommendation systems?* In particular, we make the following thesis statement:

Enhancing the performance predictability and cost-efficiency of deep recommendation systems require the optimization of fundamental storage supports such as caching, I/O admission, and prefetching techniques, while seamlessly integrating efficient machine learning methods to bolster the adaptability and learning capability of the storage system in handling the ever-changing workload patterns.

To support this statement, the dissertation is structured into three main parts, each delving into distinct facets of building storage supports for deep recommendation systems. In the first segment (Chapter 2), we delve into innovative techniques and designs aimed at architecting an efficient EV table lookup system. This system capitalizes on both structural regularity in inference operations and domain-specific approximations to bolster the performance and cost-efficiency of hybrid

storage systems. In the second part (Chapter 3), we think further to design the next-generation machine learning-based I/O admission system to improve the performance predictability of SSD devices when being used as the secondary storage for storing the EV table. In the third part (Chapter 4), we discuss our prefetching solution to enhance the performance of the secondary storage system. In summary, this dissertation will cover the following systems we built:

• **EVSTORE:** A caching system that exploits groupability pattern between items for scaling and enabling cost-efficient deep recommendation system deployment.

• **HEIMDALL:** A machine learning-based I/O admission policy which combines a simple yet powerful neural network model to proactively and deterministically redirect I/O when the device is busy.

• **TINFETCH:** A "tiny" neural-network block-I/O prefetcher which leverages an adaptive heuristic and machine learning methods to further improve SSD's performance.

For the rest of the chapter, we will briefly introduce the scalability and predictability challenge of deep recommendation system (Section 1.2), then develop multiple storage supports, including caching support (Section 1.3.1), I/O admission support (Section 1.3.2), and prefetching support (Section 1.3.3). In Section 1.4, we summarize the contributions of this dissertation. In Section 1.5, we introduce the outline for the rest of the dissertation.

1.2 Motivation

Recommendation systems are used prominently across modern online services to help people make decisions. They capture user behavior and preferences to display personalized advertisements [138, 139], rank news [38, 92], and recommend products [318]. The impact of recommendation systems on user engagement is tremendous. Recent studies show that a significant amount of content—30% of all traffic on Amazon's website, 60% of the videos on YouTube, and 75% of the viewed movies on Netflix came from suggestions made by recommendation algorithms [32, 35, 288, 341].

Deep recommendation systems (DRSs) are widely used to deliver high-quality recommendations [139, 371], but tackling *categorical ("sparse") input features* is their Achilles' heel. Modern DRSs, such as Facebook's post recommendation systems [139], often contain hundreds or thousands of categorical features (*e.g.*, users, posts, or pages), each of which can contain millions or even tens of billions of possible categories. To make the complexity of the deep neural network (DNN) tractable, sparse categorical data is usually converted to ("dense") vectors of numbers before being fed to the model. The most popular conversion is via *embedding vector tables*, or "**EV tables**" for short.

EV tables are large and growing. Today's recommendation models have enormous feature sets to capture complex user behavior and preferences [85, 92, 139, 374, 379, 380]. Each categorical feature could assume 10^7–10^{10} different possible values [146, 268, 371], implying that billions of embedding vectors are needed in practice to represent every unique feature. A billion embedding vectors (rows) with 400 dimensions (columns) [205, 247, 376] of `fp32` type (cell size) would easily occupy 1.5 TB of memory. Furthermore, industry's insatiable appetite for improved recommendation accuracy demands more rows, extra columns, and larger vectors (cells) to encode richer semantic relationships. Thus, the models are growing rapidly—the sizes are tripling every two years (1.5× annual growth), following Moore's Law [55, 161]. Such DRAM-heavy architectures account for significant operational costs for DRS users measured in millions of dollars—nearly 80% of all AI-related deployments in Meta's data centers in 2020 directly supported DRSs [139].

DRS pipelines are up against a scaling wall. Crucially, all trends point to the continued burgeoning of DRS system sizes. Recent projections predict that EV table sizes will imminently be dozens of TB for some companies [55], flirting with the limits of even the greatest memory capacity cloud instances available[1]. To continue scaling DRS, a different approach is required.

Unfortunately, the state-of-the-art DRSs are simply not equipped to handle the exponential growth of EV table sizes. Open-source DRSs platforms like Meta's DLRM [254] and Google's

1. At the time of writing, high-memory instances top out at 24 TiB (AWS), and 12TiB (Azure/Google Cloud).

DCN [320, 321], for example, store the *full* EV tables in DRAM and lack support for responding to lookups from backend storage when memory is exhausted. This brings several downsides. When the entire memory is mostly occupied by EV tables of a specific DRS model, the server is not able to run other DRSs concurrently, potentially reducing resource utilization of the server and the overall throughput of the recommendation service. Furthermore, storing the entire EV tables in memory is costly as the price of DRAM keeps increasing, especially due to shortages in global supply [43]. A natural solution to this problem is by moving the large EV tables to the SSDs as the backend storage. There are recent publications in this space that focus on optimizing the backend storage for EV table lookups [110, 317, 331]. While existing storage solutions advance the state of the art, their adoption is limited due to the need for customized devices (*e.g.*, custom SSDs or FPGA implementations).

The performance instability of SSD devices. Incorporating off-the-shelf SSDs to reduce DRAM footprints may seem like the most cost-effective option [238], but it can introduce performance instability. This becomes particularly problematic for large-scale Deep Recommendation Systems (DRSs) tasked with serving billions of users [92, 254, 295] while maintaining low latency to meet strict microsecond-scale tail latency Service Level Objectives (SLOs) [153, 165, 364]. Adding to this challenge is the current trend of microservices and Machine Learning deployment, where these already stringent tail-latency SLOs are expected to become even tighter over time [151, 166, 237]. Recent studies highlight the difficulty in achieving SSDs with deterministic latency [23, 62, 100, 269] due to the non-deterministic behavior of SSD's internal activities such as the garbage collection (GC), wear leveling, and internal buffer-flush [21, 112, 348]. These background activities will disturb users' I/O requests and cause tail latencies. For instance, GCs can cause up to 60x latency increase [211] which can easily violate the SLOs.

The storage industry and the community are aware of this problem and have devoted a vast amount of research to address the issue of unpredictable latency in SSDs. One approach, referred to as the "white-box" method, involves restructuring the internal architecture of the device [88,

159, 163, 175, 225, 308, 336, 349]. While this approach is powerful, it may not be widely adopted by SSD vendors due to various constraints. Another strategy, known as the "gray-box" approach, suggests making partial modifications at the device level alongside changes at the operating system or application level [179, 181, 182, 286, 365, 369]. However, the success of this solution largely depends on the willingness of vendors to modify the device interface, which may pose challenges in implementation. Lastly, "black-box" techniques aim to conceal unpredictability without altering the underlying hardware or its level of abstraction. Some of these techniques focus on optimizing file systems or storage applications for SSD usage [90, 177, 183, 192, 198, 252, 297, 337, 342], while others rely on speculative execution [8, 19], which introduces additional I/Os and latency due to waiting times.

All of the observations above point out that storage systems are now faced with significant challenges in supporting the ever-growing demands of recommendation systems. Not only must they efficiently manage the vast amounts of data generated by these systems, but they must also ensure low latency, high availability, and cost-effectiveness. Therefore, there is an opportunity to create innovative storage systems that can seamlessly integrate with recommendation systems, providing scalable and reliable storage solutions while also optimizing performance and resource utilization. These storage systems supports should be capable of dynamically adapting to fluctuating workloads and evolving requirements, thereby enhancing the overall deployment efficiency of recommendation systems.

1.3 Contributions

Our analysis underscores the significant challenge of addressing performance instability within our storage infrastructure to meet the escalating demands of recommendation systems. Recognizing the complexity inherent in modern storage architectures, we have developed a suite of storage support mechanisms, each targeting different layers of optimization. In this section, we first introduce our caching layer support to provide a performant and cost-efficient deep recommendation

system deployment (Section 1.3.1), then describe our machine learning-based I/O admission support designed to cut performance variability "at the source" (Section 1.3.2), and finally detail how we further enhance SSD's performance with our block-I/O prefetcher support which leverages an adaptive heuristic and machine learning methods to predict intricate block I/O stream patterns (Section 1.3.3).

1.3.1 Caching Support with EVSTORE

We start developing our solutions from the inference layer of the Deep Recommendation System, where the performance variance originates. This layer is critical as it directly impacts the user experience by determining the relevance and accuracy of the recommendations provided. By focusing on optimizing this foundational layer, we aim to enhance the overall performance and efficiency of the recommendation system. Our approach begins by addressing fundamental questions specific to the DRS platform itself: How can we reframe this challenge within the context of the DRS platform? Is it feasible to integrate a novel caching layer within the DRS platform, one that operates seamlessly with a commodity storage backend? Can the caching layer be optimized specifically for EV access patterns? To address these questions, we built EVSTORE: a novel EV table caching layer in DRS inference pipelines that exploits available DRAM and the structure of EV lookups to optimize end-to-end DRS inference latency. EVSTORE's main contributions lie in EVSTORE's 3-layer "L1-to-L3" caching design (EVCache, EVMix, and EVProx):

(L1) EVCache: We built a caching layer (EVCache) where EV tables are stored as key-values in the DRS memory and backend storage. We harness an all-or-nothing EV access property: an inference will query a set of keys to *all* of the EV tables, hence a cache miss on just *one* of the keys will make the entire inference slow. State-of-the-art cache replacement algorithms do not fit this lookup pattern. Hence, we introduce the concept of *groupability* and extend existing algorithms with "group scores" to rank keys that are likely accessed together and retain them in the cache, which in turn increases the chances of getting a "perfect-hit" where all of them simultaneously can

be found in memory.

(L2) EVMix: To accommodate diverse latency and accuracy tradeoffs, we delegate some space from the L1 into an L2 segment that stores lower precision (16, 8, or 4 bits instead of 32-bit floating point) embedding values. For instance, whereas the first layer stores 32-bit floating point values (`fp32`), the second layer can store lower precisions (*e.g.*, in 16, 8, or 4 bits). We call this combination of L1 and L2 as EVMix, a *mixed-precision* caching. This brings several advantages: allowing more key-value pairs to be cached, increasing hit rates, accelerating inferences, and boosting throughput in trade for a minor loss of accuracy.

(L3) EVProx: Finally, we leverage another unique characteristic of embedding values: The value for a key that is not in the cache can be replaced by a *surrogate* key whose value is "*approximately similar*" to the original key's value. We add a *key-to-key* caching layer (L3) that maps each key to a surrogate key with a similar embedding value. Furthermore, we choose surrogates that are likely to reside in the L1/L2 cache to help alleviate accesses to the backend storage. To the best of our knowledge, the closeness of embedding keys, computed using well-established statistical methods for similarity analysis [96, 221, 273], has not been previously used for DRS performance optimization.

We have fully integrated EVSTORE within Facebook (Meta)'s DLRM [254], including various implementation-level optimizations and offline supporting tools (≈9 KLOC) that are released publicly [11]. Our evaluation based on real production DRS traces shows that EVSTORE can reduce the average and p90 latency by up to 23% and 27% respectively, while increasing the throughput by 4× at only 0.2% accuracy reduction. Collectively, fully optimized EVSTORE implementation can achieve a 94% reduction of the DRS memory footprint. These memory savings correspond to hundreds of millions of dollars for a large cloud provider [209].

1.3.2 I/O Admission Support with HEIMDALL

Although EVSTORE's caching layers provide a performant and cost-efficient recommendation system deployment, its reliance on SSDs as the backend storage can introduce performance instability due to the unpredictable SSD's internal activities such as garbage collection (GC), wear leveling, and buffer-flush. These background activities will disturb users' I/O requests and cause tail latencies. For instance, GCs can cause up to 60x latency increase which can easily violate the SLOs.

To make our solution both effective and generic, we turn to extensive machine learning exploration to build an efficient and accurate I/O admission method. In the last 2+ years, we have built HEIMDALL, an extensive machine learning pipeline designed for an important storage subdomain: the I/O admission policy for flash storage. Here, the storage system needs to decide for every I/O whether to admit it to the underlying storage device or reroute it to other devices (§3.2). As highlighted in the last row of Table 3.1, HEIMDALL's pipeline covers more machine learning approaches compared to the state of the art. For each method, we must apply it in a careful, meticulous, and domain-specific way that improves the accuracy and performance of the I/O admission policy. Every decision must be justifiable and understandable, in terms of how they improve the model's accuracy and deployment performance.

On accuracy (§3.3), we make domain-specific innovations related to I/O admission policy in various ML stages. We justify how supervised learning, such as *a neural network,* suits well for fast admission decisions. To help our supervised learning, we show the limitation of simple labeling methods that are based on latency cutoffs and introduce a *more accurate period-based labeling method*. Further, we introduce a *3-stage noise filtering* using a domain-specific understanding of how device-specific features and characteristics, such as device-level caches, retries, and error check-and-correction (ECC), can lead to noisy data. With cleaner data, we perform *in-depth feature engineering* where we show that even in its sub-steps like feature scaling, there are many scaling options, such as normalization and standardization, that need to be evaluated. Finally, we perform *fine-grained tuning* of our neural network parameters to determine the optimal number of layers

and neurons, as well as the appropriate activation and output functions to choose among various options to balance between accuracy and inference overhead.

On deployment performance (§3.4), we perform various levels of optimization, from Python-to-C++ conversion, `gcc`-flag usage, to quantization, in order to reduce inference time from a naive 45,000µs to *sub-µs latency*. This is very important as we target real-world storage systems such as the Linux block layer and Ceph [326]. We also provide *joint/group inference* with various efficient models capable of making a single inference for multiple I/Os. This lean design leads to negligible memory and CPU overhead.

We built HEIMDALL in ≈21 KLOC (§3.5), providing three levels of integration: user-level storage (for fast and large-scale evaluation), Linux kernel (for mimicking real deployments), and Ceph-Rados (for distributed storage settings). Our comprehensive evaluation (§3.6) uses 2 TB of real-world I/O traces and generates 11 TB of intermediate data for all the experiments. We evaluate our model *unbiasedly* with 500 random experiments, demonstrating that HEIMDALL delivers 15-35% lower average latency compared to popular algorithms, such as hedging and advanced admission heuristic and ML models [142, 307], and up to 2× faster to a baseline.

Behind this optimal performance is HEIMDALL's impressive accuracy improvement, from a raw average accuracy of 67% up to 93%. We also compare HEIMDALL with AutoML and show that the 16 models generated by AutoML have 22% lower accuracy than HEIMDALL, showing that meticulous design and fine-tuning still win for our problem domain. We also show that AutoML models are too heavyweight and impractical for our deployment scenarios.

1.3.3 Prefetching Support with TINFETCH

Given that the tail latency is handled by HEIMDALL, we can still further improve the performance stability of the secondary storage (SSD/HHD) by utilizing prefetching methods. In the contemporary landscape of high-performance servers tailored for data centers and the demanding field of AI/ML training, the dependence on solid-state drives (SSDs) and spinning disks (HDDs) as

secondary storage devices takes center stage. The integral role played by these storage devices in dictating the overall system performance accentuates the need to address and minimize their I/O latency, especially in a hybrid storage system [176]. Caching and prefetching emerge as prevalent strategies to mitigate the high access latency of storage devices. Caching involves using faster but less dense memory to store frequently accessed data [189, 298]. Prefetching [77], whether implemented in software within the operating system [213, 249, 378] or directly within the SSD/HDD firmware [343], is introduced as a crucial technique for reducing latency by fetching data from their original storage in slower memory to cache before they are needed.

Typical block-level cache prefetchers operate by receiving input in the form of logical block address (LBA) sequences, represented as integer numbers. These prefetchers predict the LBA of the data that is likely to be accessed shortly and decide on whether to prefetch it or not. Efficient prefetcher design encounters two primary challenges. First, real-world applications demonstrate complex LBA access sequences due to random accesses from diverse users or applications, which are common in modern large-scale storage systems [63, 217]. Second, the precision of prefetchers is vital for their effectiveness. Inaccurate prefetchers result in inefficient use of I/O bandwidth and cache space [144]. Consequently, the development of effective prefetchers holds significant importance for storage systems.

Prefetchers are mostly heuristics going back 1-2 decades ago [91, 104, 152, 167, 197, 223, 258, 259, 266, 282, 299, 338, 382] where they heavily rely on predefined rules to prefetch data based on Logical Block Address (LBA) access sequences. However, they struggle with adapting to complex real-world scenarios due to their rigid pattern detection technique. For example, the read-ahead prefetcher [114, 193] is restricted to prefetch the next data item within a file for faster sequential accesses. To address this limitation, various learning-based methods are developed [83, 325, 335], including recent Long Short-Term Memory (LSTM) techniques like DeepPrefetcher [124] and Delta LSTM [80], which model the LBA delta (difference between successive access requests) and cover a broader range of LBAs. However, these methods ignore internal temporal

correlation during concurrent accesses, leading to limited performance. More advanced prefetchers [140, 357] capable of learning complex I/O access patterns are often hard to deploy due to their computational cost. Accurately predicting future I/O accesses remains challenging, especially in real-world applications where sequential I/Os are mixed with random requests from multiple users performing independent tasks simultaneously. The transformation of straightforward accesses on the application side into seemingly random accesses on the SSD level adds complexity to the prediction challenge [71].

In this work, we take the following position: *an ideal solution should be the combination of an efficient heuristic approach and an accurate learning-based method that can dynamically adjust its aggressiveness to minimize cache pollution*. Moreover, an efficient prefetcher can be deployed into the production system with negligible computation and space overhead. Unfortunately, existing prefetching algorithms fall short for several reasons. First, they are designed to only utilize either the heuristic (pattern-based) approach or the complex learning-based method. Second, the deployment of the learning-based methods is complicated due to their intricate environment set up for enabling the continuous training pipeline. Finally, they cannot quickly adapt to temporal changes in page access patterns which results in over-polluting the cache when being optimistic and under-utilizing the cache as a result of being pessimistic.

To this end, we propose TINFETCH, a **Ti**ny **N**eural Network for block pre**fetch**ing via precise address separation and suffix prediction. Unlike existing prefetching algorithms that rely on detecting and learning specific LBA delta patterns, TINFETCH works by employing an adaptive heuristic method to disentangle complex interleaved I/O streams and using a pretrained Neural Network model to predict future LBA access based on the suffixes. Specifically, TINFETCH divides the prefetching algorithm into two modules to form a cohesive hybrid solution, the adaptive (heuristic) module and the predictive (learning-based) module. First, the adaptive module focuses on disentangling the random accesses and tuning the prefetching aggressiveness. While the disentanglement makes TINFETCH resilient to short-term irregularities in access patterns (e.g., due

to multi-threading), the aggressiveness tuning helps it to minimize cache pollution. Second, the predictive module focuses on predicting the future LBA access with the assistance of a small pre-trained model. This design enables TINFETCH to be deployed in any storage system without the necessity of a pre-installed ML/AI pipeline, often requiring GPU or TPU (Tensor Processing Unit).

We evaluate TINFETCH against practical real-time prefetching algorithms (Stride, Read-ahead, Markov Chain) and various state-of-the-art algorithms (SGDP [357], Leap [235], Pythia [67], Delta LSTM [80]) on real production traces from three major companies (Alibaba [40], Microsoft [25], and Tencent [33]) and FIO-generated [12] traces. The result shows that TINFETCH achieves a 3% to 27% improvement in hit rate compared to state-of-the-art heuristics and ML-based prefetchers. Additionally, it has the highest hit rate per storage bandwidth load of 0.21, surpassing the best state-of-the-art learning-based and heuristic-based prefetchers by 2.6x and 1.5x respectively. Furthermore, our multi-dimensional evaluation shows that TINFETCH strikes a remarkable balance between a high hit rate and minimal storage bandwidth load, outperforming the second-best by 10% in hit rate. We provide a low-overhead, practical implementation of TINFETCH on a C/C++ language that is optimized for production deployment which achieves inference latency in sub-μs, tested on 2.6 GHz Intel Core i9.

1.4 Contributions

Overall, this dissertation makes the following contributions:

- We introduce a full-stack storage supports, from novel caching layer at application level to low-level OS/SSD firmware for cutting tail latency and block-level prefetching. And we demonstrate its superior performance through extensive evaluations over a wide range of storage/data workloads and comparisons with many state-of-the-art works.

- We introduce a novel caching abstraction to harness an all-or-nothing EV access property: an inference will query a set of keys to all of the EV tables, hence a cache miss on just one of

the keys will make the entire inference slow. We advocate a groupability concept to extend existing algorithms with "group scores" to rank keys that are likely accessed together and retain them in the cache, which in turn increases the chances of getting a "perfect-hit" where all of them simultaneously can be found in memory.

- We design and implement a 3-layer EV table lookup system that harnesses both structural regularity in inference operations and domain-specific approximations to provide optimized caching for Deep Recommendation System deployment.

- We perform an extensive exploration of machine learning pipeline to build HEIMDALL, a robust I/O admission and redirection policy for flash storage. We provide three levels of integration, for user-level storage (for fast and large-scale evaluation), Linux kernel (for mimicking real deployments), and Ceph-Rados (for distributed storage settings).

- We make domain-specific innovations related to I/O admission policy in various ML stages. We justify the use of *supervised-learning such as a neural network* to perform I/O admission decision. We show the limitation of simple labeling methods that are based on latency cutoffs and introduce a *more accurate period-based labeling method*. Further, we introduce a *3-stage noise filtering* using domain-specific understanding of how device-specific features and characteristics such as device-level caches, retries, and error check-and-correction (ECC) can lead to noisy data.

- We develop TINFETCH, a hybrid prefetcher that combines the effectiveness of heuristic methods and the learning capability of a Neural Network model. To the best of our knowledge, TINFETCH is the first prefetching solution that pioneers the use of a pretrained model and a precise address separation method for disentangling interleaved I/O streams.

- We introduce *storage bandwidth load* as an important metric for effectively evaluating prefetching algorithms, given its agnostic nature towards cache size and caching algorithms. This

metric, when combined with the hit rate, provides a comprehensive assessment of both cache pollution and storage bandwidth utilization.

1.5 Thesis Organization

The remainder of this dissertation is structured as follows: Chapter 2 delves into the designs and evaluation of our EVSTORE caching solution. In Chapter 3, we present our solution to mitigate the tail latency problem in SSD with HEIMDALL. Chapter 4 discusses our TINFETCH prefetching solution to enhance the performance of the secondary storage system. Chapter 5 provides a summary of other contributions. Lastly, Chapter 6 outlines potential future work, leading to the conclusion in Chapter 7.

CHAPTER 2

EVSTORE: STORAGE AND CACHING CAPABILITIES FOR SCALING EMBEDDING TABLES IN DEEP RECOMMENDATION SYSTEMS

2.1 Overview

Recommendation systems are used prominently across modern online services to help people make decisions. They capture user behavior and preferences to display personalized advertisements [138, 139], rank news [38, 92], and recommend products [318]. The impact of recommendation systems on user engagement is tremendous. Recent studies show that a significant amount of content—30% of all traffic on Amazon's website, 60% of the videos on YouTube, and 75% of the viewed movies on Netflix came from suggestions made by recommendation algorithms [32, 35, 288, 341].

In the age of Deep Learning, Deep Recommendation Systems (DRSs) are widely used to deliver high-quality recommendations [139, 371], but tackling *categorical ("sparse") input features* is their Achilles' heel. Modern DRSs, such as Facebook's post recommendation systems [139], often contain hundreds or thousands of categorical features (*e.g.*, users, posts, or pages), each of which can contain millions or even tens of billions of possible categories. To make the complexity of the deep neural network (DNN) tractable, sparse categorical data is usually converted to ("dense") vectors of numbers before being fed to the model. The most popular conversion is via *embedding vector tables*, or "**EV tables**" for short (section 2.2).

By reducing the DNN complexity, EV tables sacrifice space for faster computation, and thus require significant memory. Consequently, the space management of EV tables becomes challenging: many real-world EV tables contain billions of embedding vectors [146, 318] that require tens of TBs of memory capacity. Such DRAM-heavy architectures account for significant operational costs for DRS users measured in millions of dollars—nearly 80% of all AI-related deployments in Facebook's data centers in 2020 directly supported DRSs [139]. Additionally, industry's insatiable

appetite for improved recommendation accuracy is driving the rapid growth of EV tables in DRS. As users become more reliant on these systems, they expect higher quality recommendations that are tailored to their individual preferences. To meet this demand, recommendation systems must be able to encode richer semantic relationships, which requires larger EV tables. This has led to a tripling of EV table sizes every two years (1.5× annual growth) [55, 161].

Unfortunately, the state-of-the-art DRSs are simply not equipped to handle the exponential growth of EV table sizes. Open-source DRSs platforms like Facebook's DLRM [254] and Google's DCN [320, 321], for example, store the *full* EV tables in DRAM and lack support for responding to lookups from backend storage when memory is exhausted. This brings several downsides. When the entire memory is mostly occupied by EV tables of a specific DRS model, the server is not able to run other DRSs concurrently, potentially reducing resource utilization of the server and the overall throughput of the recommendation service. Furthermore, storing the entire EV tables in memory is costly as the price of DRAM keeps increasing, especially due to shortages in global supply [43]. A natural solution to this problem is by moving the large EV tables to the backend storage (SSDs or HDDs). There are recent publications in this space that focus on optimizing the backend storage for EV table lookups but not that many [110, 317, 331]. While existing storage solutions advance the state of the art, their adoption is limited due to the need for customized devices (*e.g.*, custom SSDs or FPGA implementations).

In this work, *we take a different approach*: How should we revisit this problem from the context of the DRS platform itself? Can we add a novel caching layer within the DRS platform (that works on commodity storage backend)? Can the caching layer be optimized specifically for EV access patterns? To address these questions, we built EVSTORE: a novel EV table caching layer in DRS inference pipelines that exploits available DRAM and the structure of EV lookups to optimize end-to-end DRS inference latency. EVSTORE's main contributions lie in EVSTORE's 3-layer "L1-to-L3" caching design (EVCache, EVMix, and EVProx):

(L1) EVCache: We built a caching layer (EVCache) where EV tables are stored as key-values

in the DRS memory and backend storage. We harness an all-or-nothing EV access property: an inference will query a set of keys to *all* of the EV tables, hence a cache miss on just *one* of the keys will make the entire inference slow. State-of-the-art cache replacement algorithms do not fit this lookup pattern. Hence, we introduce the concept of *groupability* and extend existing algorithms with "group scores" to rank keys that are likely accessed together and retain them in the cache, which in turn increases the chances of getting a "perfect-hit" where all of them simultaneously can be found in memory.

(L2) EVMix: To accommodate diverse latency and accuracy tradeoffs, we delegate some space from the L1 into an "L2" segment that stores lower precision (16, 8, or 4 bits instead of 32-bit floating point) embedding values. For instance, whereas the first layer stores 32-bit floating point values (`fp32`), the second layer can store lower precisions (*e.g.*, in 16, 8, or 4 bits). We call this combination of L1 and L2 as EVMix, a *mixed-precision* caching. This brings several advantages: allowing more key-value pairs to be cached, increasing hit rates, accelerating inferences, and boosting throughput in trade for a minor loss of accuracy.

(L3) EVProx: Finally, we leverage another unique characteristic of embedding values: The value for a key that is not in the cache can be replaced by a *surrogate* key whose value is "*approximately similar*" to the original key's value. We add a *key-to-key* caching layer (L3) that maps each key to a surrogate key with a similar embedding value. Furthermore, we choose surrogates that are likely to reside in the L1/L2 cache to help alleviate accesses to the backend storage. To the best of our knowledge, the closeness of embedding keys, computed using well-established statistical methods for similarity analysis [96, 221, 273], has not been previously used for DRS performance optimization.

We have fully integrated EVSTORE within Facebook (Meta)'s DLRM [254], including various implementation-level optimizations and offline supporting tools (≈9 KLOC) that are released publicly [11]. Our evaluation based on real production DRS traces shows that EVSTORE can reduce the average and p90 latency by up to 23% and 27% respectively, while increasing the throughput

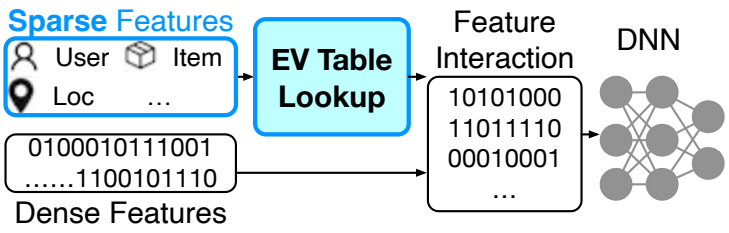


Figure 2.1: **DRS and EV Tables (section 2.2).** *EV tables are used to accurately translate the sparse categorical data into dense vectors of numbers by revealing hidden relationships between input features. These dense vectors can then be combined with other dense features before being fed into the DNN model to obtain the inference result.*

by 4× at only 0.2% accuracy reduction. Collectively, fully optimized EVSTORE implementation can achieve a 94% reduction of the DRS memory footprint. These memory savings correspond to hundreds of millions of dollars for a large cloud provider [209].

2.2 Background and Motivation

Consider a system asked to make product recommendations related to the query "`food that kitty likes`". After processing the natural language string with standard NLP methods like tokenization and stemming [168, 316], the system is provided with a set of sparse (categorical) and dense (numerical) input features. These features include high-dimensional representations of the words in the sentence from the NLP engine, as well as supplemental information, such as user attributes and location (**Figure 2.1**).

Deep recommendation systems (DRS) are recommendation engines that leverage deep neural networks (DNNs). Unfortunately, sparse categorical data, in particular those resulting from processing text data, are a poor match for the DNNs due to the unwieldy space and time complexity they impose during training. Instead, input data is usually condensed before being consumed by the DNNs—sparse text data, for instance, undergoes *word embedding* into lower-dimensional vector space.

Embedding vectors (EV) are the most popular method for densifying sparse input features for the DRS, effectively translating sparse categorical data into dense vectors of numbers [57]. Inter-

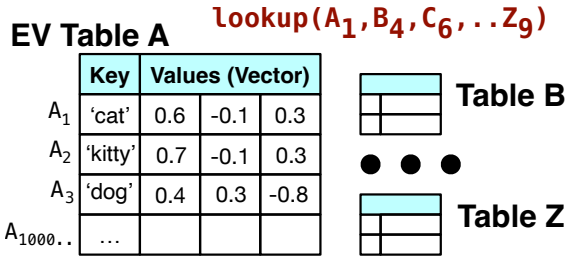


Figure 2.2: **EV table structure and lookup (section 2.2).** *An example of EV tables A–Z in a DRS. Each EV table represents the conversion for a single type of categorical feature. A lookup involves finding a key in each of the N tables ($N = 26$ if the DRS model has 26 categorical features which correspond to EV table A–Z).*

nally, the translation is done by means of an *EV table* in memory that simply returns the appropriate vector value, say $(0.7, -0.1, 0.3)$, corresponding to a given key, say '`kitty`', as illustrated in **Figure 2.2**. By reducing the dimensionality of the data, EV tables also reveal hidden relationships between inputs. For example, note that "`kitty`" and "`cat`" are practically synonyms in EV table *A* in Figure 2.2 because of the proximity of the corresponding embedding vectors. The DNN itself need not recognize the synonymy of "`kitty`" and "`cat`": since similar words cluster together in the embedding space, the queries "`food that kitty likes`" and "`food that cat likes`" will produce comparable results.

EV tables are crucial components of a DRS, so let us consider their structure and anatomy in more detail. Internally, each row in an EV table consists of a "key" index and a number of columns of floating point values representing the embedding vector corresponding to the key. Under NLP word embedding, for instance, the key may be a dictionary word like "`cat`". The key could also represent a more complex category, such as the hash of a compound string. The embedding vector columns are the values for latent features or *dimensions*. Each cell is typically a 32-bit floating point number (`fp32`). The cells are initialized as random values and gradually updated via backward propagation during training towards higher fidelity embedding vectors. The number of latent features is a design decision: more dimensions increase the lookup precision at the expense of larger tables.

A DRS lookup is the top-level inference query. Because each EV table represents the conversion for a single type of categorical feature, such as word-embedding within an NLP model, a single inference may involve dozens of different EV tables, each with potentially millions of rows [205, 247, 376]. In Figure 2.2, for example, 26 different EV tables must be consulted for a single inference. We denote DRS lookups by:

$$\texttt{lookup(}\texttt{A}_1\texttt{,B}_4\texttt{,C}_6\texttt{,..,Z}_9\texttt{)},$$

where the number in the subscript represents a key in the table. For instance, $\texttt{B}_4$ refers to key number 4 in Table B.

EV tables are large and growing. Today's recommendation models have enormous feature sets to capture complex user behavior and preferences [85, 92, 139, 374, 379, 380]. Each categorical feature could assume 10^7–10^{10} different possible values [146, 268, 371], implying that billions of embedding vectors are needed in practice to represent every unique feature. A billion embedding vectors (rows) with 400 dimensions (columns) [205, 247, 376] of `fp32` type (cell size) would easily occupy 1.5 TB of memory. Furthermore, industry's insatiable appetite for improved recommendation accuracy demands more rows, extra columns, and larger vectors (cells) to encode richer semantic relationships. Thus, the models are growing rapidly—the sizes are tripling every two years (1.5× annual growth), following Moore's Law [55, 161], while the underlying DRAM-hungry DRS implementations already weigh heavily in company budgets [139].

DRS pipelines are up against a scaling wall. Crucially, all trends point to the continued burgeoning of DRS system sizes. Recent projections predict that EV table sizes will imminently be dozens of TB for some companies [55], flirting with the limits of even the greatest memory capacity cloud instances available[1]. To continue scaling DRS, a different approach is required.

1. At the time of writing, high-memory instances top out at 24 TiB (AWS), and 12TiB (Azure/Google Cloud).

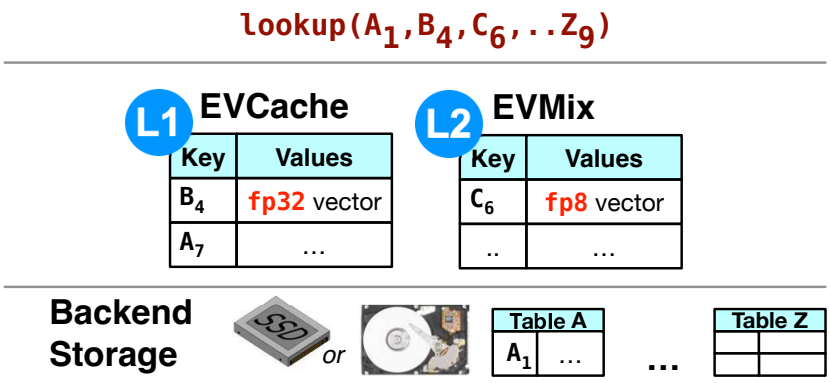


Figure 2.3: **EVSTORE design overview (section 2.3).** EVSTORE *is composed of EVCache (L1), an EV table caching layer with various cache replacement options; L2, a second caching layer which stores lower precision embedding such as* `fp8` *to enables EVMix, (subsection 2.3.2+section 2.5); and EVProx (L3), an embedding approximation layer that caches mapping to surrogate keys (subsection 2.3.3+section 2.6). The* $\texttt{lookup(A}_1\texttt{,B}_4\texttt{,C}_6\texttt{,..Z}_9\texttt{)}$ *will lead to* $\texttt{B}_4$ *hit in L1,* $\texttt{C}_6$ *hit in L2,* $\texttt{Z}_9$ *"hit" in L3 as it is replaced with the value from a surrogate key* $\texttt{A}_7$*, and* $\texttt{A}_1$ *miss that will incur a disk access.*

2.3 EVSTORE Design Overview

We present EVSTORE, a rethinking of DRS pipelines to accommodate large EV tables. With EVSTORE, EV tables are no longer required to completely fit in memory, allowing operators to grow their DRSs or improve inference throughput by packing multiple DRS pipelines among machines without running into rigid memory size constraints of individual machines. To the best of our knowledge, EVSTORE is the first system that adds powerful caching capabilities within a real-world DRS pipeline, including various implementation-level optimizations. There are three key components to the EVSTORE design, depicted in **Figure 2.3**:

1. EVCache provides the first level of caching ("L1") with various cache replacement options that are specifically tailored to handle EV lookup patterns.

2. EVMix adds support for multi-tier caching layer ("L1+L2") with mixed precisions (*e.g.*, 32, 16, 8, and 4 bit) across different layers to provide better performance.

3. EVProx accelerates lookups via a novel "L3" layer that caches approximate embedding to opportunistically replace a missing key with a surrogate key that is likely to reside in L1 or

L2.

2.3.1 EVCache

By adding a caching layer for EV lookups to the DRS pipeline, the cache replacement policy begins to dominate the performance of the lookup workload. Cache replacement algorithms have primarily been designed for items with independent request patterns (such as key-value stores), or where accesses concern ranges of consecutive memory (such as virtual memory and storage systems). Unlike traditional caches, however, DRS lookups exhibit the aforementioned "all-or-nothing" property when accessing cached EV tables. That is, for every inference request, the key-value lookup must be done across all constituent EV tables at the same time, *e.g.* `lookup(`A_1`,`B_4`,`C_6`,..,`Z_9`)`—a cache miss for just *one* of the keys (*e.g.*, A_1) will make the entire inference slow. This uncompromising attribute stems from the neural network (NN) architecture: the output value from each EV table is a portion of the input vector into the NN, without which the NN yields ill-defined results.

We evaluated both popular and state-of-the-art caching algorithms (LRU, LFU, ARC, CAR, Cacheus, ClockPro [61, 199, 240, 281]) against DRS workloads with the all-or-nothing property and found their performance to leave an opportunity for improvement (subsection 2.4.1). We noticed that existing cache algorithms could be infused with a novel notion of *"groupability"*. That is, EV-friendly algorithms ought to consider the fact that keys are accessed as a *group* in EV lookups. In a departure from ordinary caching systems, the input into our EVCache layer involves multiple keys at once as a group, rather than just a single key. With grouped keys, the objective of our caching system is then to maximize the chance of getting a "***perfect hit***" where all of the keys are found in the cache (subsection 2.4.2). We then also speak of *perfect hit rate* instead of just *hit rate* for single key lookups.

To demonstrate the flexibility of the groupability notion, we extended three popular algorithms (LFU, CAR, and ARC) into EV-LFU, EV-CAR, and EV-ARC, respectively (subsection 2.4.3).

These three EVCache variants have different implementations and characteristics that offer adaptability and choices in handling a variety of DRS workloads. For example, EV-CAR and EV-ARC both adapt well to EV-based and classical individual lookups in that it bolsters perfect hit rates without sacrificing the individual hit rates, whereas EV-LFU is highly optimized for DRS workloads at the expense of lower individual hit rates. Maximizing the perfect hit rate poses an interesting algorithmic question: what simple online heuristics can factor in groupability without undue computational overhead? We detail our approach in Section 2.4.2.

2.3.2 EVMix: Mixed-Precision Caching

Another family of approaches for increasing cache performance, besides improving the replacement policy, is to conduct domain-specific packing, either through lossless or lossy compression of values [50, 150, 310]. To balance EVSTORE's all-important latency goal with recommendation accuracy, we delegate some space from the L1 into an "L2" segment that stores lower precision embedding values. We call this combination of L1 and L2 as EVMix, a *mixed-precision* caching. Moreover, the two cache tiers will have different sizes and data precision but run the same cache replacement policy. Recalling that EV are stored as 32-bit floating point values (`fp32`), there is an opportunity to lower the resolution of the floating point value to 4, 8, or 16 bits—allowing the cache to keep more values in memory in exchange for a minor reduction in accuracy. For instance, whereas the first layer (L1) stores 32-bit floating point values (`fp32`), the second layer (L2) can store lower precisions (*e.g.*, in 16, 8, or 4 bits). Users can adjust the resolution and size to balance the desired accuracy and performance. EVMix uses fast coding optimizations that harness the specifics of embedding vector management, detailed in Section 2.5.

2.3.3 EVProx: Approximate Embedding

Another unique characteristic of embeddings that differentiates them from typical key-value data: embedding vectors reside in relatively smooth (high-dimensional) metric spaces with well-defined

distances between vectors. Thus, building on ideas from nearest-neighbor clustering, the original value of a key may be *approximated* by the "similar" value of a nearby neighbor. That the neighbors have comparable values stem from an empirical smoothness property called the *embedding value similarity* [149]. While such clustering techniques are popular for analyzing and reducing the complexity of high-dimensional data, we are not aware of any work that exploits them explicitly for performance optimization.

Using these ideas, we propose another layer, EVProx, that allows a key-value cache miss to be replaced by a *surrogate* key whose value is likely to be cached in L1/L2, hence avoiding a lookup to the backend storage. Without this L3, if a key is not available in L1 and L2, slow disk access would be needed. Accordingly, L3 can be viewed as a *key-to-key* caching layer that maps a key to a surrogate key with a similar embedding value. For example, in **Figure 2.3**, the key $\mathtt{Z}_9$ is a miss on L1 and L2. Before going to the disk, we check L3 and find that $\mathtt{A}_7$ is the surrogate key of $\mathtt{Z}_9$. Since $\mathtt{A}_7$ is already stored in L1, the disk access is prevented. Furthermore, since having a key-to-key caching requires much less space compared to caching the whole embedding value, EVProx needs minimal space and will only occupy a small percentage ($\leq 5\%$) of the total cache size. Section 2.6 further describes the challenges of implementing the L3. For instance, for every key, how do we establish the appropriate surrogate keys? Also, which key is more likely to reside in L1/L2?

2.3.4 Implementation and Integration

Our final contribution is in the implementation and integration of EVSTORE in a real DRS platform, specifically the Facebook DLRM framework [254]. We explored various implementations along several dimensions including supporting various storage backends (RocksDB[20], SQLite[37], CORTX [42], and UNIX files [311]). We then embedded a new caching layer inside the DLRM through two approaches: via the tensor library and via the EVCache layer with our optimized data structures. We also migrated our Python implementation to C++ to better support mixed precision, harness multithreading, and optimize data reuse with the help of dynamic memory

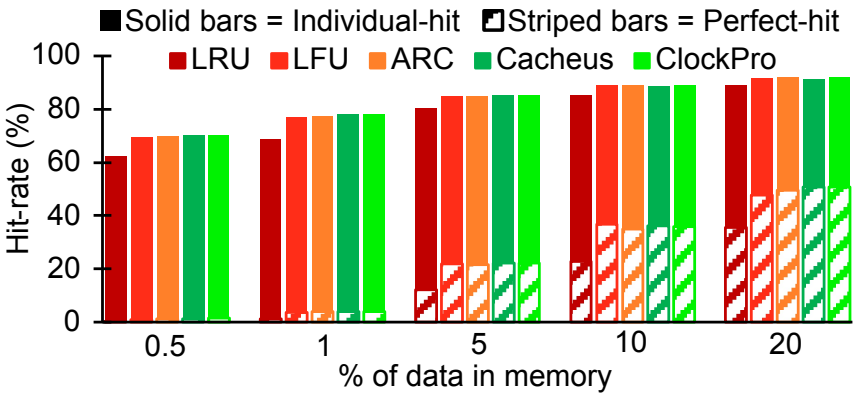


Figure 2.4: **Individual vs. perfect hit rates (subsection 2.4.1).** *Existing algorithms have high individual hit rates (solid bars), but relatively low perfect hit rates (striped bars) across various cache sizes.*

allocation and pointer manipulations. We added ≈7 KLOC to the Facebook DLRM and ≈2 KLOC of offline tools for benchmarking EVCache algorithms and EVProx approximate embeddings.

2.4 EVCache (L1)

In this section, we evaluate the performance of different caching algorithms on EV lookup workloads (subsection 2.4.1), describe how we apply our groupability principle to improve the perfect hit rates of these algorithms (subsection 2.4.2), and demonstrate how the principle can be adopted across various caching policies (subsection 2.4.3).

2.4.1 The Importance of Perfect Hits

The caching literature is replete with algorithms, from the basic policies (such as LRU, CLOCK, and LFU [95, 199, 257, 311]) to the more dynamic/adaptive variants (such as LIRS [158], CAR [61], ARC [240], ClockLIRS [158], and ClockPro [157]), and finally the machine-learning based ones (such as Cacheus [281] and LeCAR [315]). To understand how they relate to our problem domain, we evaluate the performance of these algorithms on EV lookup workloads. Recall that to serve a single inference request with N sparse features, the DRS must convert those sparse features to N dense features by doing EV lookups to N different EV tables. *Any* cache miss on one of the

EV tables requires access to the backend storage (*e.g.*, SSD and HDD) which generally is orders of magnitude slower than memory access, thus slowing down the entire inference.

To quantify caching performance, we use two metrics. First, the **individual hit** rate, the typical metric used when evaluating caching algorithms, concerns the ratio of key-value lookups that are found in memory, regardless of how many embedding tables are used in a single inference. Next, the **perfect hit** rate is the ratio of how often *all N* keys (from a single inference request) are found in the memory, a scenario where no data needs to be fetched from the disk before running pass forward phase in DRS pipeline.

Figure 2.4 shows the results when we have $N = 26$ using the Criteo dataset [28] (details in the evaluation section). Here we only show 5 algorithms for readability. For the individual hit rate (solid bars), as expected, the algorithms can reach 60–90% hit rate (vertical axis) when the cache size is 0.5–20% of the size of all the tables (horizontal axis). *However, the perfect hit rate is significantly lower, ranging only from 1% to 50%* (the striped bars), mainly because existing algorithms do not take into account the *group*-based access pattern. Moreover, as the cache size increases, the individual hit rate tends to increase in a lower rate than the perfect hit. This demonstrates that while current algorithms may be effective at finding individual items in the cache, they are less effective at finding all items in a set.

2.4.2 Replacement Policy Extension

While traditional cache lookups rely on one key per lookup, EVCache operates on multiple keys for every single inference (we call them "**grouped keys**"). Fortunately, in a DRS the cardinality of the group is fixed (*e.g.*, 26 keys whose values will be supplied to a constant number of features in the neural network model). EVCache introduces the concept of "groupability" into embedding cache management by adding a scoring metric `groupScore` for every key in the cache. Keys with high scores will remain in the cache while those with lower scores will likely be evicted. Therefore, we need a caching algorithm that prioritizes embeddings with high group scores over the ones with

low group scores, hence increasing the perfect hit. Below we describe how EVCache works from the perspectives of four fundamental caching operations: cache lookup, state update, insertion, and eviction.

Cache lookup: An inference will trigger a grouped-keys lookup, *e.g.* `lookup(`A_1`,`B_4`,..,`Z_9`)`. EVCache will calculate the total cache hits among the 26 individual key lookups. Let's suppose, 20 out of the 26 are cache hits. EVCache will memorize the group score of 20 and use it in the next caching operations.

Cache state update: For every key with a cache hit, *e.g.* B_4, its value stored in the cache will be read and prepared to be supplied to the neural network. EVCache will then update the B_4's group score in the cache with the `max` of the current and the new score. For example, if key B_4 is a hit and its current group score is 15, then EVCache will update B_4's score to 20 (the memorized score). The detail on the "`max`-based" group scoring and other scoring methods are covered at the end of this section.

Cache insertion: For every key with a cache miss, EVCache looks up the value from the backend storage and inserts the key-value to the cache with a score value of 20 (the memorized score). If the cache is full, EVCache needs to evict some key-values from the cache, *even if* they have higher scores than the scores of the to-be-inserted keys. This is because decades of caching research have shown that recency (introduced by the newly inserted keys) is an important factor in caching performance [95, 157, 158, 257].

Cache eviction: The key-values in the cache are *sorted based on the group scores*. EVCache by default evicts keys with the lowest group scores. Note that within one group score, there could be any arbitrary number of keys. Since eviction will happen frequently, we must use the appropriate data structure to avoid any bottleneck and minimize the overhead. Thus, we pick an `unordered_set` data structure to store those keys efficiently. Specifically, there is one `unordered_set` per group score. This data structure gives minimum overhead during eviction because it has an O(1) runtime.

Summary: We keep our "`max`-based" group scoring method relatively simple for two reasons: it is computationally cheap while giving the best perfect-hit rate improvement compared to other scoring methods we tried, including average, sum, median, static, and dynamic-based ones. Score calculation based on average, sum, and median will not only increase the metadata size but also the computational cost. We also tried an incremental update with a static increase of x (*e.g.*, $x = 1$) but struggled to define an optimal value of x in a dynamic workload. Furthermore, since every newly inserted key has the same value of x, highly groupable keys may readily be evicted soon after they are inserted. Defining a dynamic value of x likely requires a more complex implementation—some of the approaches we tried decreased the perfect hit rate by 50% despite being 30× slower.

Overall, our groupability concept targets the relationships between cached embedding data that were requested at the same time. Harnessing relationships between items have been extensively explored in the cache literature, ranging from long-standing observations about the relative recency of requested data [66, 95], tenuring highly-frequent items [199], exploiting other data attributes [64, 65, 66], and even learning request histories through non-linear machine-learning approaches [302]. To the best of our knowledge, the systems literature has not before considered caches where requests arrive together as a set of items. Under such a model, the relationship between items in the same set adds a dimension to the analysis that transcends the traditional dynamical notions of frequency and recency that abound in the cache literature. Our intuition is to strongly inform cache eviction by providing *fate sharing of friends* through a scoring function– to have items that are accessed together reinforce, or abate, the scores of one another. The next section shows how we integrated the scoring extension (as part of the groupability concept) into popular cache replacement policies.

```
                                EVCache Pseudocode
-------------------------------------------------------------------------------------
Global variable:
    struct EVData = an embedding vector data struct
    List<EVData> ArrCachedEV = Array to cache EVData.

EVData[] funcRequest(List<string> keys[]):
```

```
    EVData[] arrEVData = []
    int aggHit = funcGetAggHit(keys[])

    /** Update data in the ArrCachedEV **/
    for each k in keys:
        if (k in ArrCachedEV):
            /** A HIT **/
            /** Update position of the key in the ArrCachedEV **/
            currEVData = funcUpdate(k, aggHit)
        else:
            /** A MISS **/
            currEVData = funcFetchData(k)
            funcInsert(currEVData, aggHit)
        endif
        /** Insert a new page to arrEVData **/
        arrEVData.append(currEVData)
    endfor
    return arrEVData

int funcGetAggHit(List<string> keys[]):
    int aggHit = 0
    for each k in keys:
        if (k in ArrCachedEV):
            aggHit ++
        endif
    endfor
    return aggHit

void funcUpdate(String key, int aggHit):
    /** Use aggHit to update position of the data. The higher the aggHit, the more
     secure the position. We will use funcGetEVData() to get the EV-data based on the
     given key. **/
    return EVData

void funcInsert(EVData newEVData, int aggHit):
    if (ArrCachedEV is full):
        funcEvict()
    endif
    /** Insert the newEVData to ArrCachedEV. Use the aggHit to position the data. The
     higher the aggHit, the more secure the position. **/

void funcEvict():
    /** Evict the Least Groupable data at ArrCachedEV **/

EVData funcGetEVData(List ArrCachedEV, string key,
        int _aggHit):
    /** Find the requested embedding data at ArrCachedEV[] based on the given key. If
     the _aggHit > data's aggHit, we will replace it with the _aggHit. **/
    return EVData

EVData funcFetchData(string key):
    /** Get the embedding data from secondary storage based on the given key **/
```

```
    return EVData
-------------------------------------------------------------------------------------
```

2.4.3 EVCache Variants

To show generality, we implemented our extension to three popular (base) algorithms: LFU [199], CAR [61], and ARC [240]. Our three EVCache variants (**EV-LFU**, **EV-CAR**, and **EV-ARC**) have different implementations and characteristics. The main differentiator is how the base algorithms could accommodate group scores into their data positioning mechanism which also influences their eviction policy. In the interest of space, we will not describe the base algorithms in detail (interested readers can refer to our code [11]).

1. EV-LFU: This algorithm is the modified version of the Least Frequently Used (LFU) cache replacement policy. We replace the default frequency counter in LFU [199] with a group score. This means that upon a cache miss, EV-LFU will evict the cached item with the lowest group score. If there are multiple items with the same group score, EV-LFU will evict the least recently inserted item. The group score used in EV-LFU has a maximum value (*e.g.*, 26 in our main experiment), which ensures that the scores of items in the cache do not become too large over time. When most of the cached items reach the maximum score, recently cached keys with lower group scores start to face higher eviction pressure. To avoid class imbalance, EV-LFU implements a flushing mechanism with a tunable knob. Specifically, if the number of `maxScoreKey` (key with maximum group score) is higher than the "`maxScoreKeyCapacity`" (*e.g.*, 20%), EV-LFU will reduce the population of the `maxScoreKey` by $X\%$ (where X can be adjusted dynamically).

Furthermore, both LFU and EV-LFU are categorized as *stack algorithms* which makes them free of Belady anomaly. Specifically, in a stack algorithm, the items evicted by a larger cache will be a subset of those evicted by a smaller cache if both were to see the same request sequence—a property known as cache inclusion—independently of those cache sizes. Conveniently, the hit rate of stack algorithms increases monotonically with cache size [283], which provides a further degree of robustness to EV-LFU in practical settings.

EV-LFU Algorithm

```
Global variable:
    struct EVData {string key, float[] value, int aggHit } = mbedding vector struct.
    int cacheSize = capacity of the cache.
    int maxAggHit = The upper limit of aggHit value.
    List<EVData> ArrCachedEV = Array to cache EVData.
    int nPerfectPage  = The number of perfectPages.
    float flushingRate  = The ratio of perfectPage to delete during the flushing.
    float perfectPageCapacity = The threshold to initiate the flushing phase.

Initialization:
    flushingRate = 0.1
    perfectPageCapacity = 0.9

EVData funcUpdate(string key, int aggHit):
    EVData currEVData = funcGetEVData(key)
    if (currEVData.aggHit < aggHit) then
        currEVData = funcSetAggHit(currEVData, aggHit)
    endif
    return currEVData

void funcInsert(EVData newEVData, int aggHit):
    if (nPerfectPage >= perfectPageCapacity * cacheSize):
        funcFlushPerfectPages()
    else if (ArrCachedEV is full):
        funcEvict()
    endif
    newEVData = funcSetAggHit(newEVData, aggHit)
    ArrCachedEV.insert(newEVData)

void funcFlushPerfectPages():
    for(int i = 0; i < flushingRate * nPerfectPage; i++):
        /** Evict the least recent perfectPage.**/
    endfor
    nPerfectPage -= flushingRate * nPerfectPage

EVData funcSetAggHit(EVData currEVData, int aggHit):
    if (aggHit == maxAggHit):
        nPerfectPage ++
    endif
    currEVData.aggHit = aggHit
    return currEVData

void funcEvict():
    /** Evict the least recently used data at ArrCachedEV, similar to LFU's method **/
```

2. EV-ARC: ARC [240] is an adaptive algorithm designed to recognize access recency and frequency by dividing the cache into two lists: `R-list` (recency-based) and `F-list` (frequency-

based). R-list holds items accessed once while F-list keeps items accessed more than once since admission. To dynamically adjust the size of the probationary segment (R-list) and the protected segment (F-list), ARC uses information about recently evicted cache items (stored as R-ghost and F-ghost lists). For EV-ARC, we add group score as a metadata to every cached item. We then modify the F-list to use EV-LFU's counting, eviction policy, and flushing mechanisms. The difference is that cached items flushed from the F-list will be transferred to the tail of the R-list. The ghost cache size will be adjusted so that the number of the cached pages in R-list and R-ghost is equal to the number of the cached page in the F-list and F-ghost.

EV-ARC Algorithm

```
Global variable:
    struct EVData { string key, float[] value, int aggHit} = Embedding vector struct.
    int cacheSize = capacity of the cache.
    int maxAggHit = The upper limit of aggHit value.
    List<EVData> listRecentEV = recency List
    List<EVData> listFrequentEV = frequency List
    int nPerfectPage  = The number of perfectPages.
    float flushingRate  = The ratio of perfectPage to delete during the flushing.
    float perfectPageCapacity = The threshold to initiate the flushing phase.

Initialization:
    flushingRate = 0.1
    perfectPageCapacity = 0.95

EVData funcUpdate(string key, int aggHit):
    /** Check whether the "key" exists at the recency or frequency list. **/
    if ( listRecentEV.contains(key) ):
        return funcGetEVData(listRecentEV, key, aggHit)
    else if ( listFrequentEV.contains(key) ):
        return funcGetEVData(listFrequentEV, key, aggHit)
    else:
        /** This is a MISS, must insert a new page. **/
        return funcInsert(key, aggHit)
    endif

void funcInsert(EVData newEVData, int aggHit):
    if (nPerfectPage at frequencyList >=
        perfectPageCapacity * cacheSize):
     funcFlushPerfectPages()
    /** This function will insert the "newEVData" and its aggHit to the recency/
     frequency list based on various conditions (such as recencyTarget value). However,
     we didn't change any of those behaviors. Also, the minimum frequency counter might
     be updated, similar to the EV-LFU case. **/
```

```
    return newEVData

void funcFlushPerfectPages():
    for(int i = 0; i < flushingRate * nPerfectPage; i++):
        /** Evict the least recent perfectPage.**/
    endfor
    nPerfectPage -= flushingRate * nPerfectPage

void funcEvict():
    /** Depends on the recencyTarget, the eviction could happen at listRecentEV or
     listFrequentEV. To evict the data at listRecentEV, it will remove the least
     recently used data. If the eviction is performed at listFrequentEV, it will find
     the group of data that has smallest "counter" and evict the least recently used
     within that group. **/
```

3. EV-CAR: CAR [61] is an algorithm that combines ARC and the popular CLOCK second-chance algorithm. For EV-CAR, we modify the reference bit, R variable, so that it will store the group score instead of just storing 0 or 1. During the eviction phase, the CLOCK hand will only evict the cached item that has $R = 0$, otherwise, it will be challenged by the incoming key. If the incoming key's group score is larger than the current item (pointed by the CLOCK hand), EV-CAR will not evict that item, but give a second chance to the current item by setting its R to 0. EV-CAR also modifies the CLOCK mechanism by introducing a "progressive decrement" method which allows the R value to be decreased regardless of the group score of that item. This method guarantees the CLOCK hand to find an item to evict within a single rotation ($O(n)$ complexity where n is the number of items in the cache). In a cache hit, EV-CAR applies max-based scoring (subsection 2.4.2) which replaces the current group score if the new score is bigger.

EV-CAR Algorithm

```
Global variable:
    int recencyTarget = the maximum page that should be stored at clockRecentEV.
    int decPartitionSize = cacheSize/maxAggHit; which is the number of pages need to
            be checked before increasing the decrementRate.
    struct EVData {
        string key, float[] value, boolean referenced, int aggHit
    } = The struct of embedding vector.
    CLOCK<EVData> clockRecentEV = recency CLOCK
    CLOCK<EVData> clockFrequentEV = frequency CLOCK
```

```
EVData funcUpdate(string key, int aggHit):
    /** Check whether the "key" exists at the recency or frequency clock. **/
    if ( clockRecentEV.contains(key) ):
        return funcGetEVData(clockRecentEV, key, aggHit)
    else if ( clockFrequentEV.contains(key) ):
        return funcGetEVData(clockFrequentEV, key, aggHit)
    else:
        /** This is a MISS, must insert a new page **/
        return funcInsert(key, aggHit)
    endif

EVData funcInsert(string key, int aggHit):
    EVData newEVData = funcFetchData(key)
    if ( cache is full):
        funcReplace(aggHit)
    endif
    /** This function will insert the "newEVData" to the recency clock data structure
     and might adjust the recencyTarget (which will define the size of recency and
     frequency clock) based on various conditions. However, we didn't change any of
     those behaviors. The noteworthy difference is that EV-LFU adds aggHit value to
     the page's metadata when inserting a new page to the cache. **/
    return newEVData

void funcReplace(int aggHit)
    /** This is the Eviction phase where the clock "hand" is trying to find the
     least-groupable page to evict. **/
    int counter = 0;
    int decrementRate = 1;
    while (true):
        if ( clockRecentEV.size > recencyTarget):
            /** Get the data pointed by the "hand" **/
            EVData data = clockFrequentEV.getFirstData()
            /** If data.referenced bit is 1, the data will be moved to clockFrequentEV.
             Otherwise, it will be removed from the clockRecentEV and then break the
             loop. This is the same as the original mechanism at CAR policy **/
        else:
            /** Get the data pointed by the "hand" **/
            EVData data = clockFrequentEV.getFirstData()
            if (data.referenced):
                if (data.aggHit <= aggHit):
                    data.reference = false
                else if (data.aggHit-decrementRate <= 0):
                    /** Progressive Decrement Phase **/
                    data.aggHit -= decrementRate;
                    counter++;
                    if (counter >= decPartitionSize):
                        decrementRate++;
                        counter = 0;
                    endif
                /** advance the hand to the next page **/
                endif
            else:
```

```
                funcEvict(clockFrequentEV);
                break;
            endif
        endif
    endwhile

void funcEvict(CLOCK<EVData> clockData):
    /** Evict the data pointed by "hand" at clockData. **/

EVData funcGetEVData(CLOCK<EVData> clockData, string key,
        int aggHit):
    EVData currEVData = clockEV.get(key)
    if (currEVData.aggHit < aggHit):
        currEVData.aggHit = aggHit
    endif
    return currEVData
```

2.5 EVMix (L2)

To make our caching layer more versatile in addressing various latency and accuracy tradeoffs, we introduce EVMix, a multi-tier mixed-precision EV caching system. In this section, we first describe the advantages of EVMix (subsection 2.5.1), its design (subsection 2.5.2), and the bit coding optimizations (subsection 2.5.3).

2.5.1 Advantages of Mixed Precisions

An embedding vector is stored as floating point values. In most systems such as DLRM [254] and DCN [320], the default precision is `fp32` (32 bits). However, EVMix caching layer can store those values in a lower precision format such as in 16, 8, or even 4 bits depending on the target accuracy.

EVMix can bring several advantages. ***(a)*** *Faster inference latency*. By accessing smaller bit representations, we can improve the average EV lookup latency by 15%, which is significant because EV lookup can cover 40% of the end-to-end inference latency. ***(b)*** *More cached items and higher cache hits*. With lower precisions, we can cache more embeddings (*e.g.*, 8x more cached items when the 32 bit EV is converted to 4 bit), and by implication, the cache hits will be higher, which in turn increases the throughput of the caching layer. ***(c)*** *Configurability via multiple layers*.

With multi-tier caching, one can adjust the size and the precision of the first level cache and the second level cache based on the latency-accuracy tradeoffs to make caching more versatile. (more in the evaluation section).

2.5.2 Multi-Tier, Mixed-Precision Design

As shown earlier in Figure 2.3, EVMix is the combination of L1 and L2 which collectively forms a mixed-precision caching. Each tier runs the same cache replacement policy. L1 stores high or medium precision data (*e.g.*, 32 or 16 bit) and L2 stores lower precision data relative to L1's. (*e.g.*, 4 bit). Users can adjust the precision of L1 and L2 and their sizes based on the performance-accuracy tradeoffs. The size proportion of L1/L2 is fully adjustable. If L1 and L2 have the same memory size, the L2 can carry at least $2\times$ more items (due to the lower precision storage). Upon a cache miss on L1, we try to get a lower-precision data from L2. If we also get a miss in L2, we will fetch the raw data from the backend storage and put their representations to either L1 or L2 based on the group score. To minimize the accuracy loss associated with using EVMix, the popular items are stored in L1 while the less popular ones are packed in L2. Our L1/L2 placement algorithm also ensures that the items are not redundantly stored.

Furthermore, to maximize performance, we implement EVMix in C++ which utilizes multi-threading capabilities to parallelize any atomic operations in both layers. To simplify the logic and to reduce the context switching, we design the thread organization in such a way that the task for L2's threads is triggered and managed by L1's thread. In addition, we only implement event-driven paradigm on specific tasks that require heavy I/O and computation such as reading from files and binary decoding operations. Finally, we utilize a confined memory sharing to capture the results from all threads concurrently with minimum blocking.

2.5.3 Bit Coding Optimization

As part of the process above, EVMix stores the embedding data in an encoded format (4, 8, 16, or 32 bit) and continuously decodes the cached data on every cache hit. The decoded data will be fed to the neural network model in the subsequent phase of the DRS pipeline. To further improve the performance of EVMix, the decoding process must be optimized, especially for the 16, 8, and 4 bit format since there is no default (standardized) floating-point binary format for them.

As EVSTORE is built specifically for caching embedding vector data, it exploits the fact that the values of these vectors range only from -1 to 1, rather than an arbitrary range of values. Moreover, the typical value distribution is a Gaussian bell-shaped curve where the occurrence/frequency is most highly concentrated near 0. Therefore, to make the most efficient use of each bit, we design the coding procedure to better represent this "dense region" of values. We design the coding procedure for simplicity to ensure that decoding remains computationally cheap and does not become a bottleneck in our caching systems.

The scheme works as follows. **(a) 16 bit:** We store the value as an unsigned short. The mapping is straightforward, the smallest-positive EV value will be mapped to 0, while the biggest-positive EV value to 65534. We utilize the last digit as our sign bit to cheaply differentiate the positive and negative embedding. Specifically, if the last digit is odd, the value is considered negative, and if the last digit is even, the value is considered positive. The decoding phase will convert each value into a corresponding floating-point value proportionally. **(b) 8 bit:** In this case, we can only store values ranging from 0 to 255. Similar to the 16 bit, we map the embedding value linearly. The -1 is mapped to 0; the +1 is mapped to 254; and, everything that falls in between will be mapped proportionally (*e.g.*, 0.23 is mapped to 156). As a result, we use 255 values out of 256 which consists of 127 values covering the negative EV, another 127 covering the positive EV, and 1 value that is mapped into 0. **(c) 4 bit:** Although 4 bit can represent 16 values, but we only use 15 values (7 positives, 7 negatives, and a zero mapped value) to cover the EV range. Most of the value mappings are focused near 0. Specifically, we pick -0.0625 to 0.0625 as the dense region range

in a manner similar to Posit's [262] 4-bit mapping. Overall, our encoding mechanism only uses static dictionary mapping and basic operators (`XOR` and `mod`) which result in a negligible ($<1\%$) CPU overhead.

We further explored Posits library (C++) which is specifically designed to encode embedding values and quantize machine learning weights for lower precision. Despite having a well-researched encoding design that better preserves near-zero values, the library induces costly overhead due to its custom binary operations—compared to our encoding design, the Posit library is $3\times$ slower. Given that the decoding operation will be done on every single value retrieval, we decided to use our simple encoding design as described above.

2.6 EVProx (L3)

Recall from subsection 2.3.3 that our caching capabilities are built from the unique characteristics of EV lookup workloads. In EV lookup workloads, a value of a key can be replaced by a *surrogate* key's value that is "*approximately similar*" to the original key's value. The embedding value similarity [149] is calculated through cosine and Euclidean vector distances [96, 221, 273]. These well-established statistical methods are popular for analyzing and reducing the complexity of high-dimensional data. However, we are not aware of works leveraging them explicitly for performance optimization. Thus, we adopt the approximate embedding concept in our last caching layer, EVProx, allowing a key-value cache miss to be replaced by another similar (and popular) surrogate key whose value is likely to reside in L1/L2, thus preventing a lookup to the backend storage. Furthermore, we populate the L3 with the downgraded keys from both L1 and L2 in order to better retain the warm keys in the cache.

Our design ensures heavy non-blocking tasks, especially I/O, are conducted in parallel at massive performance savings. When inserting a new key to L3, we enqueue the incoming keys and batch insertions into the L3. L3 uses dedicated I/O threads to fetch all missing values in parallel. Once all key mappings data are in memory, they are inserted to the L3 sequentially. To best pro-

long hot items in the L3, we add a reference (*R*) bit to every cached item in L3 that is similar to CLOCK policy's implementation of the second-chance eviction mechanism (§2.4.3).

2.6.1 L3 Dataflow

Looking at Figure 2.3, suppose we perform `lookup(`A_1`,`B_4`,`Z_9`)` and Z_9 is the only key not being cached in L1/L2. Before adding L3, we need to read Z_9 and its value from the disk. Now we consult L3, a *key-to-key* caching layer that will tell us whether there is another key (say A_7) that has a "similar" embedding value to Z_9. The A_7 is called a *surrogate* key to Z_9, and may come from different embedding table. Note that there can be many other keys whose values are similar to the missing key Z_9. In that case, L3 will pick the most popular key (as a surrogate) measured based on its group score. In this example, L3 keeps a mapping between C_6–A_7 because of A_7's high group score, hence increasing the likelihood that A_7 will be found in L1/L2.

Remark that L3 is a special key-to-key caching layer that does not cache any value (it only caches the keys). Thus, if a lookup of Z_9 is a hit in the L3 layer, we can retrieve the surrogate key (in this case, A_7). If A_7 is found in L1/L2, an alternative value is found and no disk access is needed. However, if Z_9 lookup is a miss in L3 or A_7 is also missing from L1/L2, then we will fetch Z_9 and its value from the backend storage and store it in either L1 or L2 as explained in subsection 2.5.2 about multi-tier and mixed-precision design.

2.6.2 Preprocessing Surrogate Keys

In designing EVCache, we encountered the following challenges: For every key, how do we determine what other keys are "similar" within the embedding space? Further, among the multiple potentially similar keys, how do we decide which one is most likely to exist in L1 and L2 cache? Finally, how and when should we populate the L3 cache? To answer these questions, we build the key-to-key mapping in an offline preprocessing manner in the following way. Note that we assume throughout that the embedding table remains static during the inference phase.

To perform similarity analysis, we adopt the statistical measures of Euclidean and cosine distances [96, 221, 273] that define similarity in terms of vector-distances [250]. This similarity analysis can be done once and the result can be reused. At the end of this stage, every key in the embedding table has a list of N most-similar neighboring keys (in our setup, the N=10). To produce the L3 key-to-key mapping, we simply pick the most popular key among the top-10 keys. To measure the popularity, we consider the historical accesses and record the access frequency of every key. By the end, supposing there are 1 million keys in the embedding table, then there is a mapping of 1 million keys to another key that is most similar and frequently accessed. Next, those mapping will be stored as a file which will make it easy to perform an online update without any shutdown. The workload type and the size of L1/L2/L3 will greatly affect the remapping frequency. If the popularity ranking is quite stable/static throughout the workload, the remapping can be avoided. In general, the remapping should be done when L3 hit rate drops significantly. The analysis of optimum remapping decisions is out of our scope. It can be studied further in future works. Finally, given that all of these tasks are done in the background, they will not introduce any bottleneck and latency overhead.

2.7 Implementation

EVSTORE is built within the popular Facebook PyTorch-based DLRM framework [254] that supports both recommendation model training and inference. The EVSTORE implementation is ≈9k LOC (≈4k LOC in C++, ≈4k LOC in Python/Bash scripts, ≈1k LOC in Java). The source code of EVSTORE, including various experiment and deployment setups, is publicly available on our GitHub repository [11]. We believe EVSTORE is the first system to support substantial caching capabilities for the EV Table lookups in this DRS framework. The details of each implementation component are explained below.

Storage backend: We extend the DLRM code to include a custom EV lookup from various key-value storage systems (RocksDB, SQLite, CORTX) and Unix files. This extension is written

in Python and is approximately 2KLOC, with the majority of the code being part of the embedding-storage library. The data are stored as a stream of binary values which consists of floating point arrays. To read a specific EV value from a file, we compute the offset of the data using its key, then use seek() to directly jump to the beginning of the bytestream. The data can then be fetched from the file using either a memory map (mmap) or direct IO. Additionally, the data is sent to PyTorch as a bytestream, which eliminates the need for serialization and reduces overhead. We added a module in PyTorch to convert the bytestream into a Tensor format.

Caching layer (Python code in DLRM): The next question is where to implement L1. We first built it inside the storage backends mentioned above, but later realized that the performance could be further improved if it was embedded inside the DRS platform. To find the best place to integrate our caching layer, we must first understand how DRS systems, such as Facebook DLRM, handle the sparse-to-dense conversion. In Facebook DLRM, before the pass forward phase in the inference pipeline, by default the sparse-to-dense feature transformation reads embedding data via the tensor library. Thus, we implement L1's data structure, which mainly utilizes set and hashmap data structures, to replace the default tensor lookup. Turns out, our own choice of data structures is much faster as it is a "thinner" layer compared to the complex tensor library.

Optimized layer (in C++) for EVMix: As we support mixed precisions in L1+L2, we learned that a C++ implementation is easier to manage and optimize, especially for bitwise operations. Furthermore, most of the arrays in the implementation are stored in a plain pointer-to-pointer structure, which has better CPU efficiency than built-in vector data structures.

In the mixed-precision experiment, it is necessary to encode and decode an unusual size of floating-point data, such as 4, 8, and 16 bit values. By default, C++ aligns floating-point data at 32-bit boundaries, so we used `ushort` and `uchar` to store 16 and 8-bit precision data, respectively. When it comes to storing 4-bit data, it is not possible to use the `ushort` or `uchar` data types, as these can only store values up to 16 and 8 bits, respectively. In order to store 4-bit data, we take advantage of the fact that two 4-bit values can be packed into a single byte of `uchar` data. This

allows us to store and manipulate 4-bit data efficiently, without wasting any bit spaces.

Managing concurrent accesses to L1/L2 required the use of multithreading to minimize the overhead of context-switching and locking. To do this, we stored the results in thread-specific memory regions, which allowed us to avoid interference between threads. Additionally, we explored several interfaces to facilitate the data transfer between DRS and the C++ caching layer, including socket [346] and ctypes [184], which we will evaluate later.

Offline tools: Besides changes to the DLRM platform, we also implemented two offline tools, for cache algorithm benchmarking and approximate embedding (EVProx) preparation. The former is written in Java and built on top of the Cache2K simulator [41]. In this platform, we prototyped EVCache algorithms and all our baseline algorithms including LRU, LFU, LIRS, ARC, CAR, and ClockPro. For EVProx preprocessing, we developed an embedding-similarity analysis framework, chiefly written in Python.

2.8 Evaluation

To evaluate EVSTORE performance, we subjected it to numerous experiments to determine the end-to-end performance while conducting microbenchmarks over multiple dimensions, such as varying the cache algorithms, cache sizes, number of EV tables, workloads, and the use of EVMix + EVProx. We structure our evaluation as a sequence of experimental questions.

2.8.1 Experimental Environment and Setup

The DRS inference pipeline: (1) A user visits a webpage that has an advertisement managed by Criteo. (2) When the user interacts with the ads, it will trigger a request sent to the Criteo's server that contains all info about the user, the ads, and the webpage that is currently visited. (3) Once the inference request arrives, the server will take the sparse features, look up the EV tables, convert them to dense features and feed them to the DNN model. (4) By default, each lookup is for 26 keys to 26 tables. (5) With EVSTORE, if a key-value is not in the cache, the DRS pipeline will fetch the

data from the raw files in the backend storage. (6) Finally, the inference result will influence the personalized advertisement of the user when they open another webpage managed by Criteo.

Datasets/workloads: We primarily use the **Criteo CTR** (Click-Through Rate) datasets, the largest open-sourced CTR dataset (up to 1 TiB in size) that could simulate EV lookups at scale. There are two CTR datasets released by Criteo, the 1TB data (Criteo-Terabyte) [26] and the Kaggle version (Criteo-Kaggle) [28]. It contains feature values and clicks feedback for millions of display ads. There are 13 dense integer features and 26 sparse categorical features (hence **26 EV tables**). All EV tables have the same embedding dimensions of 36. There are a total of 156 billion total (dense) feature values and over 800 million unique attribute values. In addition, we also use **Avazu**'s CTR dataset [27].

Default values: We omit redundant lines and numbers on some of the graphs for improved readability. These are our default values (unless otherwise noted): cache size of 5% of the total working set (the total size of all tables), the Criteo-Kaggle dataset [28] as the workload, and fp32 as the precision of the embedding values. Latency is measured in average latency in milliseconds.

Machine specification: We use Chameleon cloud's `gpu_rtx6000` and `gpu_v100` nodes [9, 171] which have Intel Xeon Gold CPU @2.60 GHz and 240 GiB Samsung SSD SM863a Series. We limit the DRAM using Linux `cgroup` tools to be small enough such that the DRS essential functions could run, but not big enough to store all the EV tables. When evaluating cache size smaller than the available DRAM, we flush the Operating System (OS) page cache every 0.25 ms to avoid any EV tables being cached by the OS. The method has been thoroughly tested to ensure there is no OS cache leak.

2.8.2 EVCache

We begin with experiments on the first layer of the cache.

Experiment #1: How much does the EVCache algorithm affect perfect hit rates? **Figure 2.5** shows that EVCache (EV-$*$) algorithm extensions improve upon state-of-the-art algorithms such

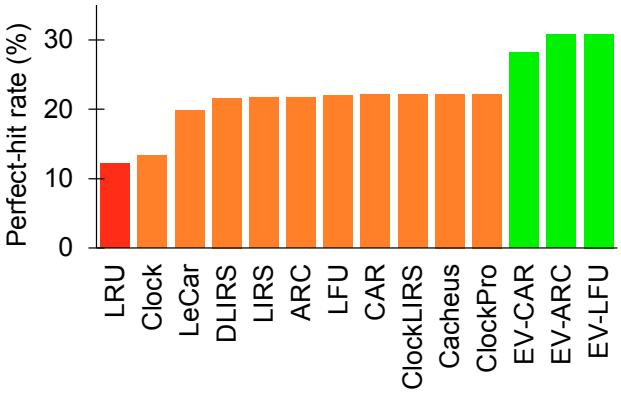


Figure 2.5: **Exp. #1 (§2.8.2): Perfect hit rates across caching algorithms.** *EVCache algorithms (EV-CAR, EV-ARC, EV-LFU) have the highest perfect hit rate compared to others.*

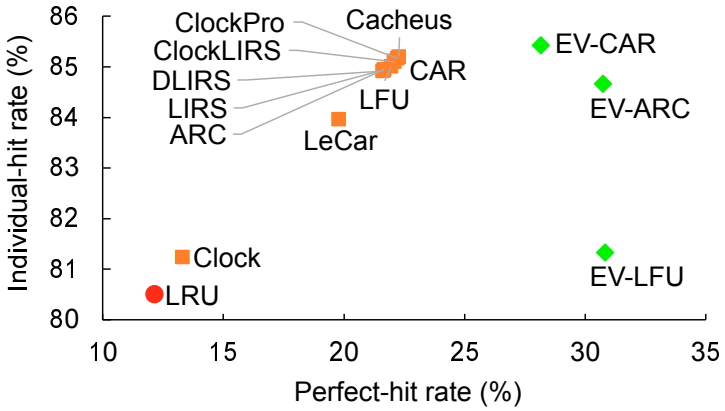


Figure 2.6: **Exp. #1 (§2.8.2): Individual and perfect hit rates across algorithms.** *EV-LFU achieves higher perfect hit rate by sacrificing on individual hits.*

as LRU [95], CLOCK [257], LeCar [315], LIRS [158], ARC [240], LFU [199], CAR [61], ClockLIRS [158], Cacheus [281], and ClockPro [157]. The perfect hit rates are increased by up to 18%, lending support to the need for groupability for EV-based caches. **Figure 2.6** breaks the result down further to compare the perfect and individual hit rates (as defined in Section 2.4.1). Here, EV-CAR and EV-ARC both improve the perfect hit rates without compromising on individual hit rates, suggesting that they can be used as a general caching algorithm too. In contrast, EV-LFU increases the perfect hit rate while sacrificing the individual hit rate for each of the tables (which is acceptable since the perfect hit rate is more significant for DRS).

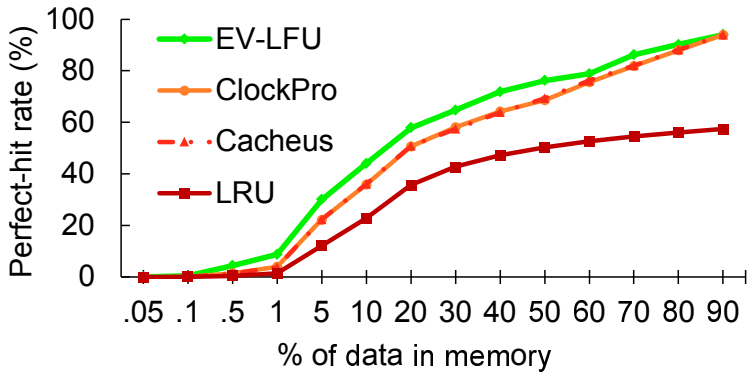


Figure 2.7: **Exp. #2 (§2.8.2): Perfect hit rates across various cache sizes.** *Our EV-LFU has the highest perfect hit rate compared to other representative algorithms across various sizes.*

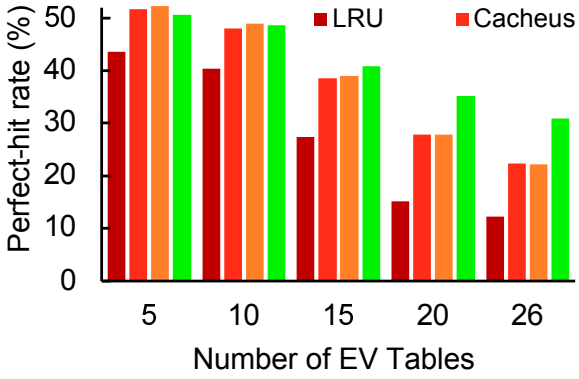


Figure 2.8: **Exp. #3 (§2.8.2): Perfect hit rates on different number of EV tables.** *EV-LFU shows steeper benefit as the number of EV tables grows (e.g., 5 to 26).*

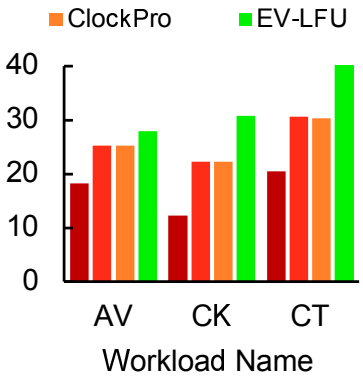


Figure 2.9: **Exp. #4 (§2.8.2): Perfect hit rates across various datasets.** *EV-LFU has the best perfect hit rate*

Experiment #2: How does EVCache perform across various cache sizes? In **Figure 2.7**, we vary the cache size from 0.05% to 90% of the total working set (horizontal axis). To reduce clutter, we show four representative algorithms (LRU as a basic algorithm, ClockPro as an adaptive one, Cacheus as an ML-based algorithm, and EV-LFU as EV-Cache variant). Here, the EVCache (specifically EV-LFU) outperforms others across all cache sizes. Compared to LRU, EV-LFU significantly increases the perfect hit rate by up to 35% while surpassing both Cacheus and ClockPro by up to 10%.

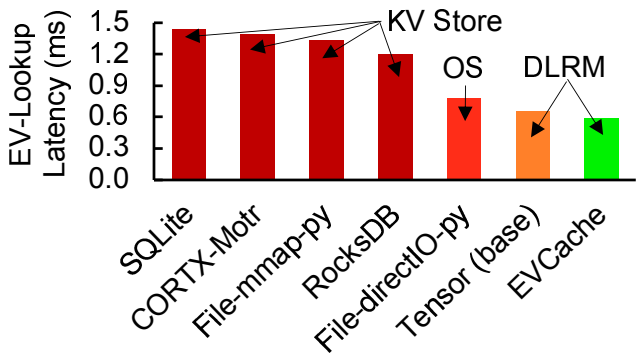


Figure 2.10: **Exp. #5 (§2.8.2: The most efficient place to implement the caching layer.** *An optimum place to deploy EVCache is inside the DLRM framework (e.g., PyTorch) using our own data structures as opposed to using PyTorch tensor library or inside the OS or an external key-value (KV) store database.*

Experiment #3: How does the number of EV tables affect performance? **Figure 2.8** shows that the perfect hit rate improves with more EV tables (horizontal axis) when using EV-LFU. As expected, traditional algorithms, being agnostic to relationships between EV tables, struggle to achieve a high perfect hit rate when the number of EV tables grows.

Experiment #4: How does EVCache perform across various datasets? **Figure 2.9** compares the four representative algorithms across three different datasets. We find that our algorithm extensions improve upon other algorithms across all the datasets: Avazu (AV) [27], Criteo-Kaggle (CK) [28], and Criteo-Terabyte (CT) [26].

Experiment #5: Which layer is the best to implement EVCache? When implementing EVCache on Facebook DLRM (in this case inside PyTorch), we tried various storage backends, including key-value (KV) stores (such as SQLite, CORTX-Motr, and RocksDB) and UNIX files via `mmap` and `read/write` APIs. By default, the EV tables in DLRM are stored as "tensor" data structure. However, we implement our own data structure of choice (set and hashmap), as part of EVCache package, to be compared against the default DLRM's tensor. In this experiment, we put all EV tables in the memory, simply to measure the pure cache lookup latency as if we have enough memory to cache all of the EV tables. For key-value stores, we cache the tables in their own caching layers. For UNIX files, we depend on the OS cache. **Figure 2.10** shows that relying on external caching layers in KV stores or OS cache do not give the best latency compared

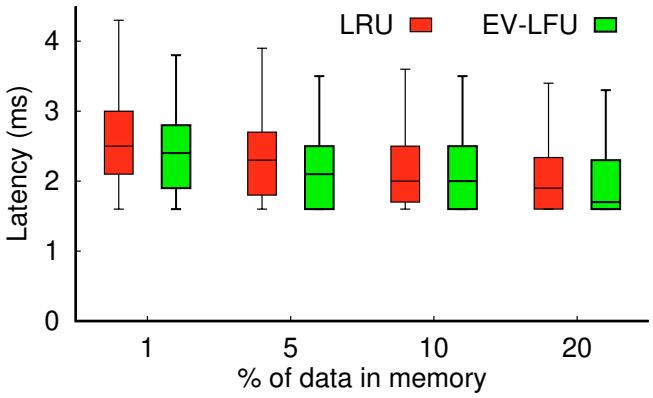


Figure 2.11: **Exp. #7 (§2.8.2): End-to-end DRS inference latency on various cache sizes.** *Each bar uses the 1^{st}, 25^{th}, 50^{th}, 75^{th}, and 99^{th} percentiles. Our EV-LFU delivers lower latency compared to the LRU implementation.*

to adding our own caching layer inside the DLRM pipeline (PyTorch in this case). Furthermore, by implementing our own thin caching layer, we get better performance than using the default PyTorch tensor.

Experiment #6: What EVCache algorithm should be implemented? After deciding the best place for the caching layer, we need to decide which algorithm to implement (EV-CAR, EV-ARC, or EV-LFU). For this, we need to port our implementation from the cache simulator to the Facebook DLRM framework. Among them, EV-CAR gives the smallest perfect hit rate, and between EV-ARC and EV-LFU, they provide comparable performance in small cache sizes but EV-LFU is slightly better at higher ($\geq$ 50%) cache sizes. In our cache simulator, Cacheus, ClockPro, EV-ARC, and EV-LFU are written in 800, 430, 270, and 130 LOC respectively. Thus, EV-LFU is more straightforward to implement by having simpler/less code compared to other algorithms. For this reason, we decided to port EV-LFU to the DLRM.

Experiment #7: How much does EVCache affect the end-to-end inference latency? At this point, in Facebook DLRM, we implemented LRU (as a baseline) and EV-LFU (as a representative of EVCache algorithms). We choose LRU as our baseline because it has the fastest lookup and the most implemented policy in the production systems. While there are more complex policies such as CAR, LIRS, CLOCKPro, etc., but they are up to 2x slower than LRU and have higher

metadata space overhead. Here is the end-to-end inference latency break down: initialization (20%), EV lookup (40%), and the DNN forward propagation (40%). **Figure 2.11** shows a whisker plot comparing the baseline LRU vs. EV-LFU. The Python-based EV-LFU implementation delivers lower latency.

2.8.3 EVMix and EVProx

Next, we evaluate EVMix and EVProx layers of EVSTORE.

Experiment #8: What implementation architecture best supports EVMix? **Figure 2.12** shows various implementation efforts we performed in re-architecting our caching layer in PyTorch and the resulting end-to-end latency. Originally, we implemented our caching data structure in Python. However, Python only supports `fp32` precision, thus we adopted a C implementation to enable storing data in lower resolution (*e.g.*, 16, 8, and 4 bits). "C-socket" refers to the C implementation that uses sockets for DLRM data transfer, "C-Ctypes" as the C implementation that uses Ctypes binding to connect our C caching to DLRM, and "*-*N*-thd" implies the number of threads being implemented to reduce cache contention §2.5.2. Based on our experiment, the "C-Ctypes-6-thd" delivers the best performance compared to other implementation choices.

Experiment #9: What are the latency-accuracy tradeoffs in floating point resolution (32 to 4 bits)? After we finalized our C-Ctypes-6-thd implementation, we can now evaluate the accuracy/latency tradeoffs when using lower precisions. **Figure 2.13** shows that reducing the precision from 32 bit to 4 bit speeds up the end-to-end latency (vertical axis) by 15% and only decreases the "PR-AUC" (horizontal axis) only by 2%. We use "PR-AUC" (Area Under the Precision-Recall Curve) to evaluate the performance of our classifier to counteract label imbalance such as in our Criteo dataset, as is standard practice [99, 156]. Intuitively, PR-AUC measures the extent to which a classifier correctly identifies all positive labels without mistaking too many others as positive.

Experiment #10: How does latency trade off against accuracy in mixed-precision L1+L2 caches? In this experiment, we divide the total cache size to L1 and L2 equally (*i.e.*, 50-50).

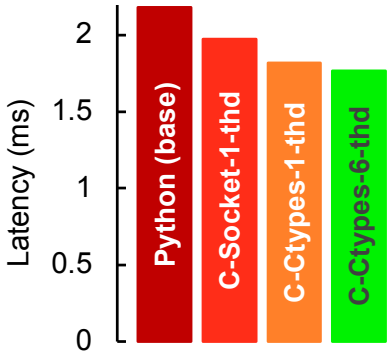


Figure 2.12: **Exp. #8 (§2.8.3): Implementation variants of EVMix.** *Python-based implementation improved by a 6-threads C version with Ctypes.*

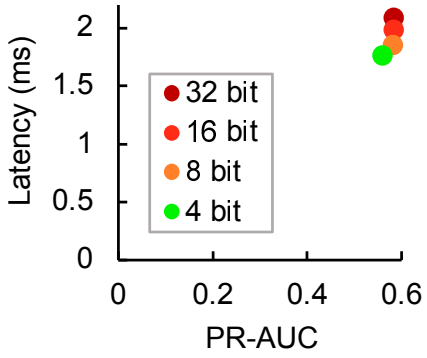


Figure 2.13: **Exp. #9 (§2.8.3): Latency vs accuracy trade-off.** *Reducing the precision from 32 to 4 bits decreases the accuracy only slightly while greatly improves the latency.*

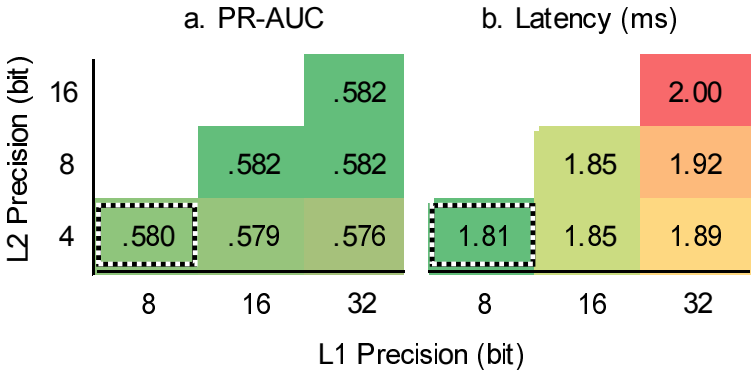


Figure 2.14: **Exp. #10 (§2.8.3): Trade-off between latency and accuracy across various L1/L2 mixed-precision caches.** *We vary the L1 precision (horizontal axis) and L2 precision (vertical axis) and report the resulting accuracy (left) and end-to-end latency (right).*

Figure 2.14 shows the accuracy (the cell content of Figure 2.14a) and average end-to-end latency (in Figure 2.14b) of EVMix as we change the embedding precision in the L1 tier (horizontal axis) and the L2 tier (vertical axis). For example, if we move from 32-bit L1 and 16-bit L2 to a 32-bit L1 and 4-bit L2 (the top-right and bottom-right corners), we improve the average latency from 2 ms to 1.89 ms and reduce the accuracy slightly from 0.582 (best case) to 0.576. Combining 8-bit L1 and 4-bit L2 gives us the best EVMix result as marked by the dotted rectangles where we reduced the latency by 10% with only 0.2% loss of accuracy.

Experiment #11: How much is the tail latency improvement with L3 (EVProx)? In **Fig-**

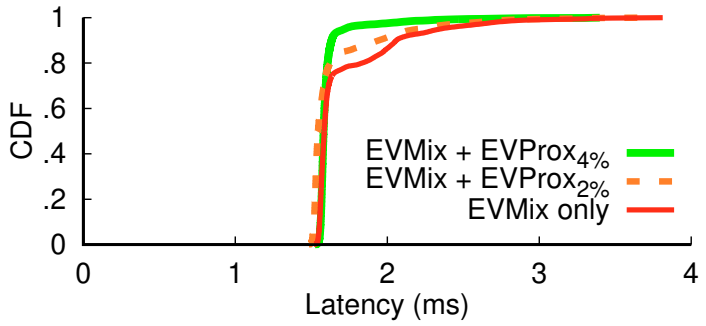


Figure 2.15: **Exp. #11 (§2.8.3): Tail latency improvement with EVProx.** *Compared to standalone EVMix, adding L3 (EVProx) layer reduces the 95^{th} and 99^{th} latency by 27% and 22%.*

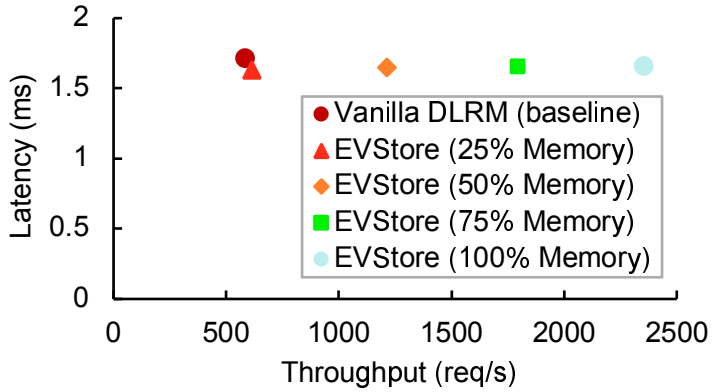


Figure 2.16: **Exp. #12 (§2.8.4): EVSTORE enables multi-DRS deployment in one node.** EVSTORE*'s scale-out deployment quadrupled the throughput while keeping the latency low.*

ure 2.15, we show the latency CDF of EVProx variants compared to EVMix. The "EVMix + $\text{EVProx}_{4\%}$" gives the best latency CDF in which we dedicate 4% of the cache size for L3 (EVProx) key-to-key mapping and split the rest for L1 and L2. Compared to the pure EVMix, adding the "$\text{EVProx}_{4\%}$" successfully reduces the 95^{th} and 99^{th} tail latency by 27% and 22% respectively. This experiment is conducted on 20% cache size.

2.8.4 Putting It All Together

Experiment #12: Does packing multiple DRSs on a machine improve throughput? As EVSTORE removes the memory requirement, in **Figure 2.16**, we show that we can concurrently run 4 DRSs on one machine (limited by the number of 4 GPUs in Chameleon's `gpu_v100` node) by

giving 25% of the memory space to each DRS. As a result, we quadrupled the throughput (inferences/second) of the DRS. The figure also shows our final EVSTORE implementation improves the latency compared to Facebook's vanilla DLRM.

Experiment #13: Can we reduce the memory footprint of the DRS service while meeting typical SLAs? **Figure 2.17** shows that to meet an SLA of 2 ms average inference latency, the vanilla DLRM will require 100% of the data to be present in memory. In contrast, EVSTORE's most optimum implementation with all features enabled (rightmost bar) needs only 6% of data to be in memory, which is a 94% reduction of memory requirement in trade for the 0.2% accuracy drop. Finally, the middle bars show how the range of EVSTORE optimizations and features demand 30% to 80% of the data to be in memory. These results demonstrate the effectiveness of EVSTORE in reducing the memory footprint of the DRS service, while still meeting typical SLAs.

2.9 Related Work

In addition to the studies surveyed throughout the paper, there are some recent publications on optimizing DRAM cache and GPU-resident cache utilization during DRS training [242, 261, 340, 350, 372, 373]. The focus, however, is on training rather than inference, and ignores systems-level nuances of cache policy and optimizations.

Another nascent body of work has extensively studied improving key-value store performance by exploiting the GPU [148, 366], NMP [56, 170, 191], SSD characteristics [53, 222], and lookup query properties [201]. They are orthogonal to EVSTORE in that could help increase the throughput of key-value store operations, which can be beneficial for DRS that rely on these stores. For instance, NMP can help convert the raw EV data into a Tensor format which will reduce the CPU load. However, these techniques do not address the specific challenges associated with the growth of EV table sizes, which is the focus of EVSTORE. Additionally, many papers require either bespoke hardware modifications or emerging memory technologies, which make them elusive for commodity hardware deployments. EVSTORE, on the other hand, is designed to be compatible

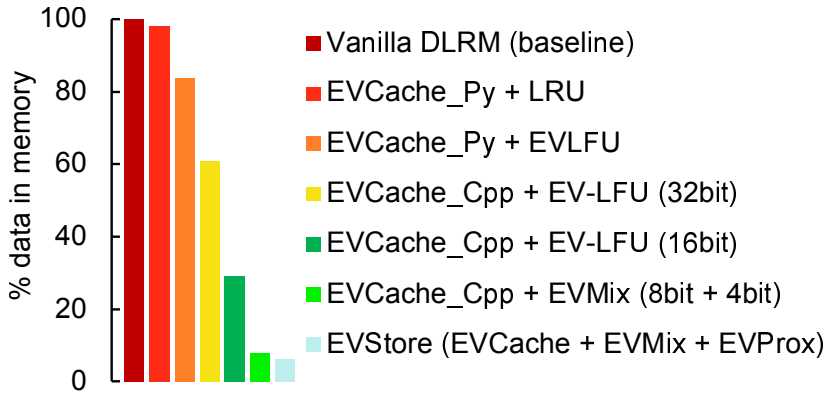


Figure 2.17: **Exp. #13 (§2.8.4): Basic to fully optimized EVSTORE.** *The y-axis shows the minimum memory footprint to satisfy SLA target of 2 ms average end-to-end inference latency. Fully optimized* EVSTORE *implementation reduces 94% of the memory footprint compared to Facebook's vanilla DLRM.*

with commodity hardware, making it a more practical and accessible solution for improving the performance of DRS.

2.10 Conclusion

We have introduced EVSTORE: a novel 3-tier EV caching layer to address the continuous growth of EV tables in deep recommendation systems. EVSTORE is a practical system that brings several advantages. DRS designers no longer need to worry about the memory size limitation of their EV tables since users with low-memory servers can still run DRSs with large EV tables. Recommendation services can also run concurrent DRSs to increase throughput and thus potentially bolster revenue. By dislodging the monolithic DRAM-hungry DRS architectures of today with a scalable systems-oriented approach, carrying relatively modest downsides, EVSTORE has the potential to curb the enormous and ballooning operational costs and expensive resources needed to run a competitive DRS across the industry.

Scientifically, we believe EVSTORE opens several doors for future work, including in the realm of EV caching (are there better policies?) and cache management (what is the best L1-L2-L3 size arrangement?). In addition, there are many components inside EVSTORE that can be further im-

proved, such as the bit-encoding method, the L3 remapping strategies, and the popularity ranking update mechanism. It also spurs questions around the role of emerging memory technologies and GPU-accelerated caching on future recommendation systems.

CHAPTER 3

HEIMDALL: ROBUST I/O ADMISSION AND REDIRECTION DEVELOPED WITH EXTENSIVE MACHINE LEARNING EXPLORATION

3.1 Overview

With the growing success of machine learning (ML) in the last decade, we have seen an increase in research that leverages ML in the storage field, addressing many challenges of complex decision-making in various contexts such as I/O admission [116, 142, 332], caching [172, 187, 281, 324, 351], configuration tuning [78], deduplication [267], failure detection [51, 97, 230], indexing [94, 215, 305], prefetching [49, 293], scheduling [229], and many others. In many cases, ML algorithms prove to work better than design decisions based on human-driven heuristics.

Despite the surge in ML applications for storage domains, we believe that its full potential has yet to be unlocked. The ML literature, including the new discipline of data science, points to the advancement of a more extensive pipeline of advanced ML methods [10, 18, 70, 106, 127]. **Figure 3.1**, *which by no means is complete*, attempts to summarize the major stages of a long ML pipeline, including data preparation and labeling, feature engineering, model engineering, training engineering, deployment optimization, and the deployment platform itself. Within each major stage lie more detailed methods, *e.g.*, data labeling can incorporate basic labeling, accurate labeling, noise filtering, and potentially many others.

To evaluate to what extent this longer ML pipeline is applied to ML-for-storage research, we summarized recent research papers in **Table 3.1** (on page 2). The five-row groups in the table represent papers in popular storage domains, such as admission [116, 142, 332], caching [281, 324, 351], indexing [94, 215, 305], prefetching [49, 293], and miscellaneous categories [69, 229, 232, 267]. We can conclude that the ML-for-storage literature has *gaps* in exploring various stages of this pipeline.

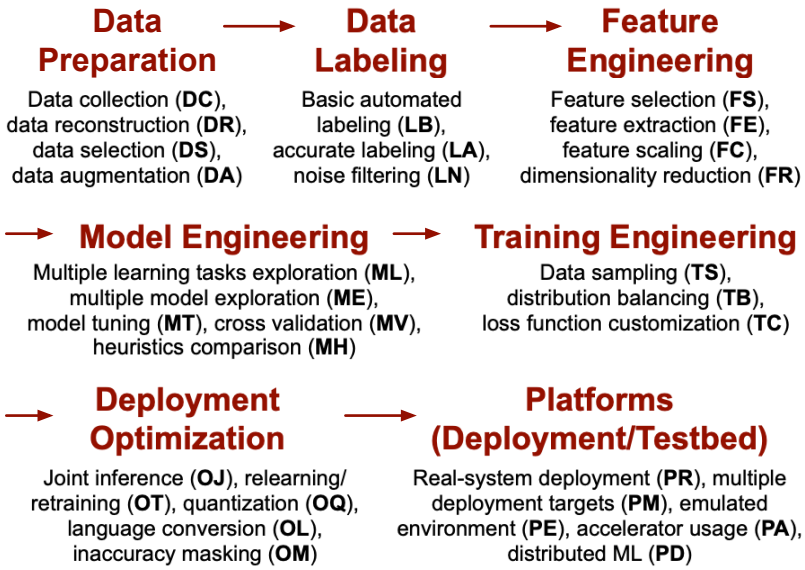


Figure 3.1: **Long machine learning pipeline.**

For example, in the data preparation stage, "data augmentation (DA)" is rarely used; hence, the model is likely not trained with pattern variations outside the available workload [324]. In the data labeling stage, prior works only employ one (most likely simple) labeling method, but did not explore more "accurate labeling (LA)" methods that also include "noise filtering (LN)" to anticipate unclean traces and reduce "garbage in, garbage out" [81]. In the feature engineering stage, "feature scaling (FC)" is not fully studied, potentially causing the model to assign disproportionate importance to certain features over others [260]. Finally, in the deployment and platform stages, most papers do not provide various models that balance overhead/accuracy via "joint/group inferences (OJ)." Many also did not consider the need for "retraining/relearning (OT)" for long-term deployment, potentially in "multiple layers/targets (PM)."

In this paper, we explore opportunities to fill in the gaps mentioned above and evaluate whether filling the gaps will improve decision-making in storage systems. In the last 2+ years, we have built HEIMDALL, an extensive machine learning pipeline designed for an important storage subdomain: the I/O admission policy for flash storage. Here, the storage system needs to decide for every I/O whether to admit it to the underlying storage device or reroute it to other devices (§3.2). As highlighted in the last row of Table 3.1, HEIMDALL's pipeline covers more machine learning

ML Methods →	DDDD LLL FFFF MMMMM TTT OOOOO PPPPP CRSA BAN SECR LETVH SBC JTQLM RMEAD
LinnOS [142]	xxxx x.. xx.. ..x.x ..x ..xxx x....
LAKE [116]	xxxx x.. xx.. ..x.x ..x ..xxx x..x.
Baleen [332]	xx.. x.. xx.. xxx.x xx. ..... x.xx.
Cacheus [281]	xxx. x.. .x.. ..x.x ... .x..
GLCache [351]	xx.. x.. xx.. ..x.x x.. xx.x. x....
XStore [324]	xx.x x.. .x.. .x..x x.. .x.xx x...x
Bourbon [94]	xxx. x.. .xx. xx..x x.. ...xx x....
LeaFTL [305]	xx.. x.. .x.. ..x.x ... ...xx x.x..
Rolex [215]	xxx. x.. .x.. ....x ... .x.x. x...x
KML [49]	xxx. x.. xxx. .xxxx x.. ..xx. x....
Voyager [293]	xx.. xx. xx.. ..x.x ... ..xxx ..x..
DeepSketch [267]	xx.. x.. .x.x xxxx. .x. ...x. ..xx.
OSML [229]	xx.. x.. .xx. xxxxx ..x ...x. xx.x.
Llama [232]	xx.. x.. xx.. x.x.x x.. ...xx x....
TraceRNN [69]	xx.. x.. xx.. x.x.. x.x ..... x....
HEIMDALL	xxxx xxx xxx. xxxxx .x. xxxx. xxx..

Table 3.1: **Usage of ML methods in ML-for-storage literature (§3.1).** *The table shows the usage of ML methods in ML-for-storage papers. Each column has a two-letter acronym that represents an ML method shown earlier in Figure 3.1. For example, "$\frac{D}{C}$" in the first column denotes "Data Collection." The major stages are separated with empty columns. "***X***" implies **use** of the method and "**.**" implies **absence/no-use**.*

approaches compared to the state of the art. For each method, we must apply it in a careful, meticulous, and domain-specific way that improves the accuracy and performance of the I/O admission policy. Every decision must be justifiable and understandable, in terms of how they improve the model's accuracy and deployment performance.

On accuracy (§3.3), we make domain-specific innovations related to I/O admission policy in various ML stages. We justify how supervised learning, such as *a neural network,* suits well for fast admission decisions. To help our supervised learning, we show the limitation of simple labeling methods that are based on latency cutoffs and introduce a *more accurate period-based labeling method*. Further, we introduce a *3-stage noise filtering* using a domain-specific understanding of how device-specific features and characteristics, such as device-level caches, retries, and error check-and-correction (ECC), can lead to noisy data. With cleaner data, we perform *in-depth feature*

engineering where we show that even in its sub-steps like feature scaling, there are many scaling options, such as normalization and standardization, that need to be evaluated. Finally, we perform *fine-grained tuning* of our neural network parameters to determine the optimal number of layers and neurons, as well as the appropriate activation and output functions to choose among various options to balance between accuracy and inference overhead.

On deployment performance (§3.4), we perform various levels of optimization, from Python-to-C++ conversion, `gcc`-flag usage, to quantization, in order to reduce inference time from a naive 45,000µs to *sub-µs latency*. This is very important as we target real-world storage systems such as the Linux block layer and Ceph [326]. We also provide *joint/group inference* with various efficient models capable of making a single inference for multiple I/Os. This lean design leads to negligible memory and CPU overhead.

We built HEIMDALL in ≈21 KLOC (§3.5), providing three levels of integration: user-level storage (for fast and large-scale evaluation), Linux kernel (for mimicking real deployments), and Ceph-Rados (for distributed storage settings). Our comprehensive evaluation (§3.6) uses 2 TB of real-world I/O traces and generates 11 TB of intermediate data for all the experiments. We evaluate our model *unbiasedly* with 500 random experiments, demonstrating that HEIMDALL delivers 15-35% lower average latency compared to popular algorithms, such as hedging and advanced admission heuristic and ML models [142, 307], and up to 2× faster to a baseline.

Behind this optimal performance is HEIMDALL's impressive accuracy improvement, from a raw average accuracy of 67% up to 93%. We also compare HEIMDALL with AutoML and show that the 16 models generated by AutoML have 22% lower accuracy than HEIMDALL, showing that meticulous design and fine-tuning still win for our problem domain. We also show that AutoML models are too heavyweight and impractical for our deployment scenarios.

Finally, we close by discussing our *retraining policy* for long-term deployment (§3.7), which further highlights more topics to explore in this long ML pipeline for storage, including efficient retraining, continual learning models, and model zoo/management. We conclude (§3.8) by demon-

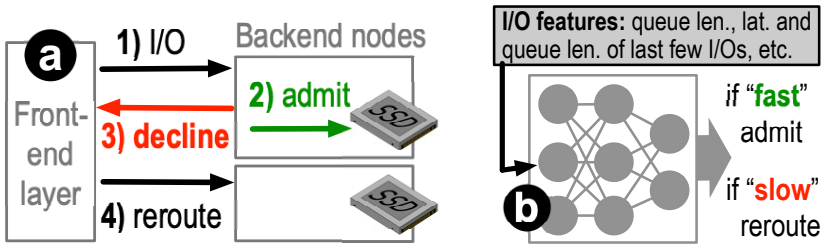


Figure 3.2: **I/O admission (§3.2).** ***(a)*** *I/O admission decision and* ***(b)*** *neural-network-based decision in every backend node.*

strating how our extensive HEIMDALL pipeline can foster "design competitions" where students and researchers can make new innovations within the pipeline or extend it.

3.2 Background and Motivation

Admission problem: The admission problem is fundamental for operations such as job submission [52, 87, 164], VM placement [284], cached data [332], RPCs [368], and I/O requests [135, 142, 178]. The admission policy needs to decide whether to admit the operation to an underlying resource (*e.g.*, machine, memory, storage device), delay, or reroute it to another resource. Such a policy is useful for resources that exhibit tail-latency behavior where most of the time the operations are fast but sometimes (*e.g.*, 1-10% of the time) are slow because of resource contention.

I/O admission: This paper focuses on I/O admission at the block level for flash storage arrays with data replication. As illustrated in **Figure 3.2a**: (1) The front-end layer sends an I/O request to a backend SSD node that has the data. Each backend node makes admission decision to (2) *admit* every request to the underlying SSD or to (3) decline the request and ask the front-end layer to (4) *reroute* it to another node that has the replica.

Flash storage: Admission decision is useful to reroute I/Os from flash devices that are experiencing heavy resource contention from garbage collection (GC), internal buffer flush, wear leveling, and not to mention, bursty workloads. Without proper admission/rerouting, all of these can induce read tail latencies and increase the average latency. We focus on optimizing read latencies as many prior works already showed that write tail latencies are very rare due to device-internal

write buffers [141, 214].

ML-based policies: I/O admission can be based on ML models such as LinnOS [116, 142], which uses a light neural network to predict contention inside black-box SSDs. As shown in **Figure 3.2b**, it makes *per-page* (4KB I/O) admission decision and uses historical features (*e.g.*, the latencies of the last few I/Os and the I/O queue lengths when those I/Os arrived) to *predict* if the incoming I/O will be "*fast*" (hence, admit the I/O) or "*slow*" (hence, reroute the I/O). The model is deployed at the kernel block layer on top of every flash device.

Training: To make accurate *binary* "fast/slow" predictions, an ML model like LinnOS must be trained first. For instance, before enabling admission decision, a storage operator can log the characteristics of the last 15 minutes I/Os, recording their static/runtime features and the I/O latencies. The operator then labels every logged I/O as "fast" or "slow" based on some latency cutoff algorithm (more in Section 3.3.1). During training, the model learns which I/O patterns result in slow I/Os for the workload-device pair. The neuron weights from training are then applied to the in-kernel model. The admission decision is now running, potentially for hours before requiring retraining due to workload drifts.

Accuracy: The ML model can make two inaccurate decisions: *false admits*, when the I/O is predicted to be "fast," but apparently experiences slowness, or *false reroutes*, when it predicts "slow," but there is no busyness. Our recent evaluation found the accuracy of LinnOS' 4-year-old model degraded to 67% as it failed to keep up with modern workloads and faster SSDs. The work also did not explore other important ML stages (Table 3.1). Next, we present HEIMDALL. For space, unfortunately, HEIMDALL's data preparation stages are not described but will be publicly available with the *artifact release* which includes the entire pipeline.

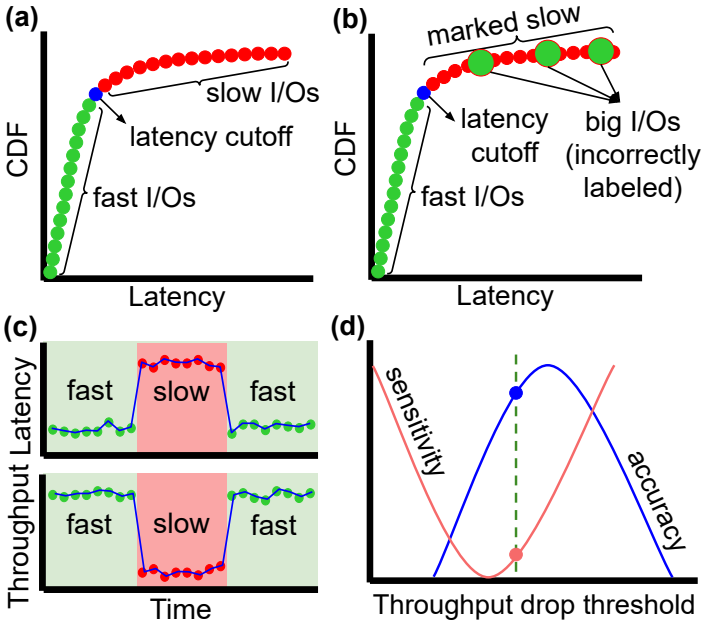


Figure 3.3: **Accurate labeling (§3.3.1).** *In all figures, a dot represent an I/O.* ***(a)*** *Fast / slow cutoff, latency based.* ***(b)*** *Inaccurate labeling, latency CDF with annotated big I/Os.* ***(c)*** *Timeline figure, period-based labeling.* ***(d)*** *Gradient descent.*

3.3 HEIMDALL's Pipeline

We begin by showing how leveraging various stages in the ML pipeline in a disciplined manner can significantly improve the model's accuracy.[1] In particular, we make domain-specific innovations in the data analysis stage with accurate labeling (§3.3.1) and noise filtering (§3.3.2); the modeling stage with thorough feature engineering (§3.3.3), model exploration (§3.3.4), and hyperparameter tuning (§3.3.5); and the training stage with data distribution balancing (§3.3.6).

3.3.1 Accurate Labeling

In the labeling step, automated labeling is necessary for labeling a large dataset. However, we found that prior works did not delve deeper into the complexities of the labeling process. In reality, "inaccurate labeling can lead to unreliable results" [300]. In this context, we look into the

1. Throughout this section, we use ROC-AUC [98] for accuracy, but later in Section 3.6.4, we report various accuracy metrics.

concept of *accurate labeling*, which is "the cornerstone of effective machine learning." Without it, machine learning models "cannot perform optimally" [381].

We first evaluated the automated labeling that prior works performed [82, 122, 123, 142, 271] in the context of latency-based modeling. We found that several methods use *latency-based* algorithms to decide the *latency cutoff*, as illustrated in **Figure 3.3a**, where the algorithm labels the I/Os in the training data set with "fast" or "slow" based on the inflection point (the cutoff) generated by the algorithm.

While latency-cutoff algorithms work well in certain domains—for example, in networking [271] where per-packet/per-request size remains stable, or in storage domain where the per-page latency model can only make inferences on per-4KB I/O [142]—this method does not work for ML models that make decision at the whole I/O level. This is because an I/O can range from a one-page (4KB) to a big request (2MB), and hence the latency-cutoff methods can lead to incorrect labeling. For example, in **Figure 3.3b**, a *large* I/O (the red dot) is labeled as "slow" here because its measured latency in the dataset is larger than the cutoff. However, this is *inaccurate* because even if the I/O is rerouted to another device, the I/O will *still* be "slow" due to its large size.

To rectify the problem, we devised several other algorithms and found that *period-based* algorithms give the best result for our problem domain. That is, instead of deciding which specific I/Os should be marked slow or fast, we label based on periods (in the window of time) where our algorithm guesses whether the device is in fast period (*e.g.*, no internal contention) or slow period (*e.g.*, experience GC contention, etc.). Although we cannot exactly predict when those events are going to happen, we can still discover some patterns in the data to narrow down the plausible tail latency segments in the dataset. For example, in **Figure 3.3c**, all the I/Os in the slow period (*e.g.*, where latency spikes and throughput drops happen) will be labeled as "slow."

Our algorithm is composed of 3 stages, as summarized in **Figure 3.4**. **(a)** First, in line 9, we categorize the relationship between latency and throughput. We should only be suspicious of device busyness when latency is high and throughput is low at the same time. We observed that

```
1. Function AccurateLabeling ()
2.    Input  : Data D {size, throughput, latency}
3.    Output: Data D {size, throughput, latency, label}
4.    high_lat, low_thpt = CalcThreshold(D)
5.    MAX_DROP = CalcThptDropThreshold(D)
6.    thpt_median = CalcMedian(throughputs)
7.    for io in D do:             // Initialization
8.       io.label = 0; io.mark_start = 0
9.       if IsBusy(io, high_lat, low_thpt, MAX_DROP) do:
10.         io.label = 1          // Start of the TailZone
11.   for io in D do:
12.      if io.label == 1 do:     // Label the TailZone
13.         while io.next.throughput < thpt_median do:
14.            io.label = 1      // 1 = decline; 0 = admit
15.            io = io.next
```

Figure 3.4: **Accurate labeling algorithm (§3.3.1).**

internal contention causes throughput drops and latency spikes. Throughput is more sensitive for detecting the start and the end of such events since throughput also takes I/O size into account. **(b)** Thus, we use latency and throughput thresholds (declared in line 4) to decide when latency looks high or throughput looks low.

Setting up the threshold values is *not trivial*. These variables are data/workload-dependent and we need to *balance sensitivity and accuracy*. We use a gradient descent-based method to pick the proper thresholds. For example, for the throughput threshold, as shown in **Figure 3.3d**, there is a maximum point (in blue line) for accuracy and a minimum point (in red line) for sensitivity. The gradient descent attempts to find a threshold value (in the x-axis) that balances these two. Finally, **(c)** based on these values, we decide when the busy period starts and ends (lines 12 to 15). In the evaluation (§3.6.4), we demonstrate that accurate labeling improves HEIMDALL's accuracy by 5.5%, resulting in accuracy as high as 93%.

3.3.2 3-Stage Noise Filtering

Still in the context of accurate labeling, there is the possibility that the training data is noisy and can mislead the model during training, leading it to recognize *insignificant features* ("garbage in, garbage out" [81]). Hence, we need to perform domain-specific noise filtering [81]. After several

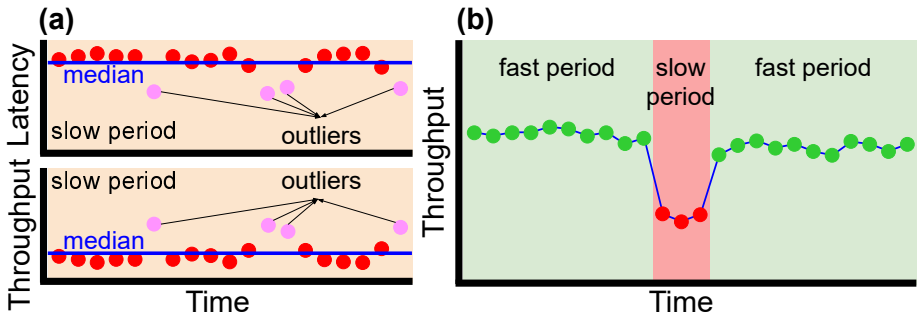


Figure 3.5: **Noise filtering (§3.3.2).** ***(a)*** *Outliers within slow period and* ***(b)*** *short noises*

tries, we introduce a robust 3-stage noise filtering.

In the first stage, we remove *outliers within the slow period*, as illustrated in **Figure 3.5a**. Here, long sequences of slow I/Os may indicate the device's internal busyness. However, sometimes a few I/Os can be "lucky" and hit the internal device cache even though the device is busy with other activities that only affect NAND-level reads and writes. Thus, as shown in the figure, we remove these "fast" outliers, specifically I/Os that have lower latency and higher throughput than the respective median values within the period.

In the second stage, we remove *outliers within the fast period*, the opposite of the first case above. These slow I/Os could happen due to some other rare device idiosyncrasies such as read retries due to voltage mismatch [76, 243, 383], error check and correction (ECC) [243], and many others. Since these rare cases are transient errors in nature, removing them from the dataset will increase the model's accuracy.

The data now becomes "cleaner," but we still find a slight irregularity as a result of our labeling process. We observed a *short burst of a slow period* (*e.g.*, only 3 consecutive I/Os), which is unlikely caused by internal device contention, as illustrated in **Figure 3.5b**. Because supplying such short bursts could confuse the model, in the third stage of the filtering, we employ the same gradient-descent method as in Figure 3.3d in Section 3.3.1 to find a reasonable threshold that will provide high accuracy but low sensitivity. In most datasets, we find a quick burst of 3 (or less) consecutive "`slow`" I/Os should be removed. Overall, our 3-stage noise filtering improves accuracy further by 16% on average (§3.6.4).

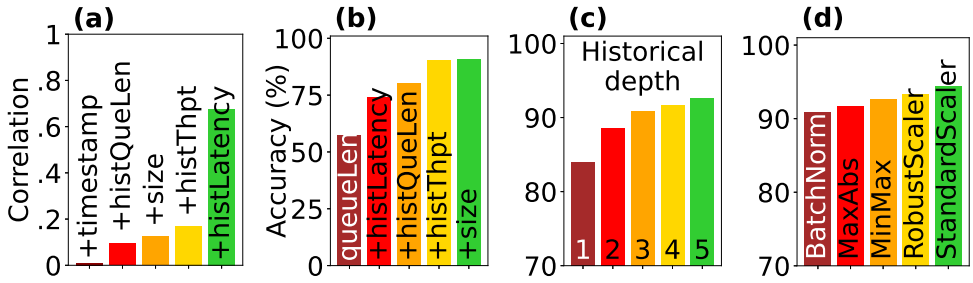


Figure 3.6: **Feature engineering (§3.3.3).** ***(a)*** *Correlation values of each feature,* ***(b)*** *accuracy improvements attributed to each feature,* ***(c)*** *model accuracy on various historical depths, and* ***(d)*** *model accuracy on different normalization methods.*

3.3.3 In-Depth Feature Engineering

The quality and relevance of features can significantly impact the model performance [106]. Typically, request-based traces already contain basic features such as request arrival time, size, and I/O type (read/write). On top of this, prior works have already performed basic **feature extraction** to extract additional features such as *runtime* information which includes latency (on the target device) and the I/O queue length observed when every I/O is submitted. Beyond basic feature extraction, in HEIMDALL, we show the importance of other feature engineering steps to increase accuracy.

First, **feature selection** can reduce the feature space and increase the efficiency of the model's training. There are many metrics to measure feature importance such as mutual information, chi-square, and correlation coefficient scores. **Figure 3.6a** sorts the features by their correlation scores with respect to the decision. For example, we can confidently remove features such as I/O arrival time (timestamp) due to its low correlation score.

Next, we conducted **feature EDA** (exploratory data analysis) on the important features to understand their impacts on the model's accuracy. We modify the input layer and add one or more features at a time. **Figure 3.6b** shows how the features affect the overall model accuracy. This result confirms the earlier finding that the five main features (the queue length, latency, historical queue length, throughput, and I/O size) all matter in improving the accuracy.

Third, we also varied the historical depth (N) of certain input features to determine the amount of past information needed by the model to obtain reasonable accuracy. By historical depth, we meant that the model should not only see information about the current I/O being submitted but should also be provided with historical information about the last N I/Os (such as the most recent queue lengths and I/O latencies). For example, if most recently, we observe an I/O with high latency but a short queue length, this could imply that internally the device is busy. **Figure 3.6c** shows that N==3 is sufficient to improve accuracy across various datasets. This choice is important for striking a balance between inference performance and accuracy since an excessive number of features will increase the training and inference overhead.

Finally, we also explored **feature scaling**, a technique to reduce model bias to a specific range of values on data features. There are two general scaling techniques, normalization and standardization, each has various algorithms [375]. With normalization, we tried various methods, as summarized in the first three bars of **Figure 3.6d**, such as batch-norm, max-abs, and min-max normalizations [45, 375], and found that min-max gives the best accuracy on average. For standardization, we found that robust and standard scalers methods (the last two bars in the figure) deliver higher accuracy but are not feasible for our domain because of their high memory overhead for keeping all the historical latency values for standard deviation and quantile calculations. In summary, min-max normalization is the most suitable, not only for its high accuracy but also because it fits the non-Gaussian nature of our dataset's distribution. This is a major departure from prior works, such as LinnOS [142], which uses digitization for scaling the input features.

3.3.4 Model Exploration

With the chosen feature set, we perform **multiple model explorations** to find the most suitable model for our problem domain. **Figure 3.7** summarizes our findings, where the x-axis is the accuracy variation and the y-axis is the normalized accuracy; the upper left of the figure is a more suitable model. Overall, we found that the neural network ("**NN**") achieves good accuracy with

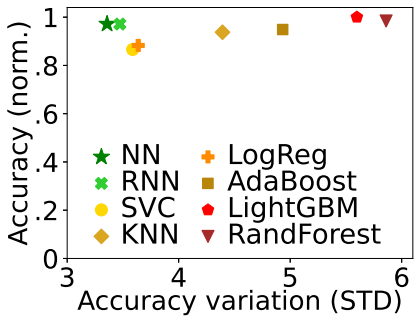


Figure 3.7: **Model exploration.** *NN model achieves high and stable accuracy.*

the highest stability compared to other models. We provide two conjectures.

First, NN learns smooth decision boundaries, especially in higher dimensions, which can generalize well to unseen data. In contrast, tree-based ensemble, support vector classifier (SVC), and k-nearest neighbors (KNN) have discrete decision boundaries, leading to higher accuracy variation.

Second, in tree-based ensemble algorithms, the tree depths restrict the models from learning the full intricate relationships in complex datasets. Although heavier and more complex than NN, recurrent neural network (RNN) also faces limitation due to their limited memory capacity which causes vanishing gradients issue. In contrast, the NN does not suffer from these limitations and can learn hierarchical features at different levels of abstraction across all layers [279].

In addition, we also consider reinforcement learning (RL), but implementing RL in our problem domain presents several challenges. RL relies on knowing the consequences of decisions for updating the punishment/reward scores in the RL's Q-table [59]. In our case, this means the model must know the consequences of declined/admitted I/Os, which are essentially the resulting latencies. However, such information is only available when we deploy the model in the target systems, which are *not written* in the same language as the model engineering platform (more in Section 3.4.1). For example, the Linux Kernel is in C, while the PyTorch model simulation is in Python. For this reason, RL is not a practical solution in our domain. Furthermore, RL is known for its slowness in convergence time (due to many trials and errors) [59] and its non-scalability in observing all combinations of the input state (which, in our context, denotes the latency of previous I/Os, which could vary widely).

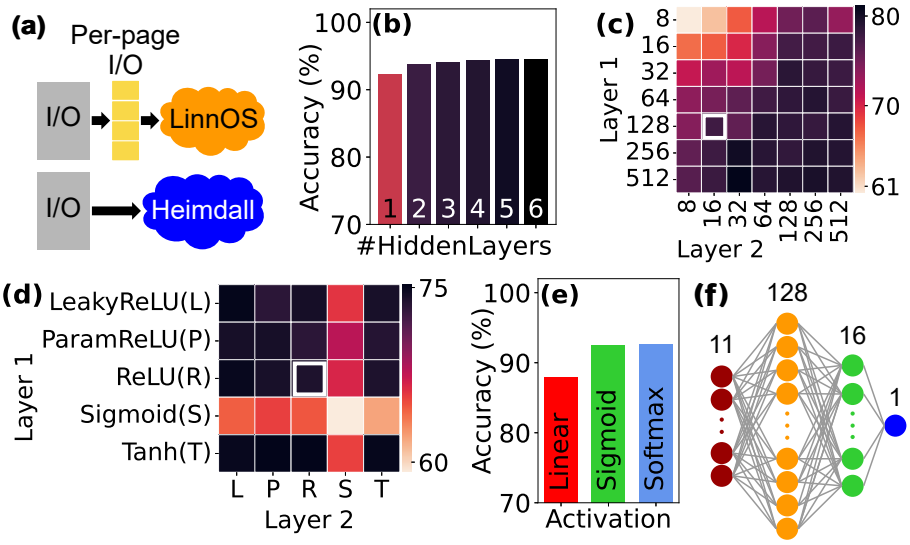


Figure 3.8: **Hyperparameter tuning (§3.3.5).** ***(a)*** *Per-page vs. per-I/O,* ***(b)*** *hidden layers,* ***(c)*** *number of layers,* ***(d)*** *activation functions,* ***(e)*** *output layer, and* ***(f)*** *final NN design.*

3.3.5 Neural Network Hyperparameter Tuning

After selecting the neural network (NN), we conduct **hyperparameter tuning** to determine the optimal number of layers and neurons, as well as the appropriate activation functions to achieve a balance between training and inference. As a result of disciplined feature and model engineering, our new NN significantly departs from LinnOS' NN model.

(a) First, as depicted in **Figure 3.8a**, LinnOS' model can only make inferences on every 4KB request, thus a big I/O must be split into small uniform-sized block I/Os, increasing the inference computation overhead. Again, this is because of LinnOS' simple per-page labeling approach (§3.3.1). However, as we fixed the issue earlier in the pipeline, our new model can make inferences for I/Os of any size.

(b) Second, we use 2 hidden layers, while LinnOS only uses one because **Figure 3.8b** shows that the most impactful accuracy growth comes from adding the 2nd hidden layer. Adding more layers can increase accuracy, but the inference time (overhead) will also increase.

(c) Third, we use 128 and 16 neurons for the first and second hidden layers, respectively (while LinnOS uses 256 neurons in just one layer). We design the model to use fewer neurons than

LinnOS to minimize its computation overhead during training and inference. This decision is backed up by our accuracy heatmap in **Figure 3.8c**. The x- and y-axis represent the number of layers in the 1st and 2nd hidden layers, respectively, and the cell color represents the accuracy achieved. We select the lightest model design which gives relatively high accuracy (darker cell color).

(d) Fourth, we kept ReLU for the activation function of the hidden layers. As a confirmation, we permuted various activation functions such as ReLU, PReLU, LeakyReLU, Sigmoid, and Tanh [289, 375]. The x- and y-axis in **Figure 3.8d** represent the permutation of the 1st and 2nd layers, respectively. We kept ReLU due to its high accuracy (darker cell color) and light overhead, unlike LeakyReLU and PReLU's higher overhead and more complicated computation.

(e) Finally, for the output layer (which makes the fast/slow prediction), we experimented with softmax, linear, and sigmoid [289, 375]. Based on the results shown in **Figure 3.8e**, we opted for a single-neuron sigmoid. This differs from LinnOS' 2-neuron output layer which bears the consequence of doubling the computation when propagating the gradient from its neighboring hidden layer.

Our model's final design is depicted in **Figure 3.8f**. While the final design appears simple, it is the outcome of a lengthy and disciplined exercise in hyperparameter tuning.

3.3.6 Training

To improve accuracy, various training methods are available. Briefly, we use **data distribution balancing** to remove prediction bias over one or more classes [81, 358]. This is important to our domain because of the heavy population imbalance between slow and fast I/Os, which often reaches a ratio of 5%:95%. However, instead of oversampling/undersampling, which might expose some risk [313], we make sure our data selection process (§3.3.3) includes some periods with heavy write I/Os (to trigger device background activities).

We also tried **biased training** by customizing the weighted loss function [314] to penalize the

model when admitting the slow I/Os. However, we see insignificant improvement or even *worse* result, highlighting that *not* all methods bring benefits as we already tuned the models with various other methods. Customizing the loss function also involves extensive tuning to determine the loss value of each class. Different datasets examined might result in different optimum loss values which require more complicated tuning.

3.4 Deployment Optimizations

From modeling and training, we now move to deployment matters such as inference latency optimizations (via joint inference and Python-to-C conversion and optimization) and retraining optimizations.

3.4.1 Negligible Inference Latency

Many storage systems, such as Linux block layer and Ceph [326], are written in C, not Python. This means that we must perform **Python-to-C** model conversion to significantly improve the real-deployment inference latency. To do so, we perform three steps (Python-to-C++ conversion, `gcc` optimization, and quantization) to reduce inference time from the naive 45,000µs to highly optimized *sub-µs* latency.

First, with a careful and manual *Python-to-C++ conversion*, we reduce the inference time from 45,000µs to 20µs. Inference in Python is longer than C/C++ for many reasons: being an interpreted language, Python executes code line by line at runtime; Python's dynamic typing, memory management, and garbage collection introduce runtime overhead; and Python's Global Interpreter Lock (GIL) can limit the parallel execution of threads in multi-core systems.

Second, we use additional `gcc` *optimization* to reduce the execution time, since without using them, the `gcc` compiler will prioritize compiling quickly which sacrifices the performance of the code itself. For this, we tested different optimization flags `-O`, `-O2`, `-O3`, `-Os`, and `-Ofast`. From here, we discovered that `-Ofast` enables the most aggressive optimizations that may disregard

precision, which can lead to unexpected behavior. Therefore, we use `-O3` since it gives the highest optimizations while still obeying the strict compliance to language standards and retaining the computation precision, which speeds up the inference to 0.08µs.

Finally, we perform *quantization* to reduce the computational complexity of our model and minimize the memory footprint. We multiply the weights by 1,024 for all layers and quantize the bias of each layer correspondingly to match the scale. We use 1,024 because we can capture the non-zero digits from most of the weights within 4 decimal points. The final latency drops to 0.05µs per inference.

We also ran tests on various CPUs and found that interestingly the inference latency can vary by an order of magnitude. For example, we get 0.12µs latency on AMD Ryzen 9 5900HS 3.30GHz and 0.08µs latency on AMD EPYC 7352 2.30GHz. On Apple M1 Pro 3.20GHz, it is even faster, at 0.05µs. We leave further investigation for future work.

3.4.2 Joint/Group Inference

In some deployments, making admission decision on every I/O might be too fine-grained and lead to high CPU overhead under intensive I/O workload and limited resources. Some recent works like LAKE [116] propose batching which only works well if the inference runs in a large batch on the GPU to exploit the parallelism (*e.g.*, sending dozens of inference jobs to the GPU at a time and returning *all* the results to the host). While this improves throughput, the host-to-GPU latency overhead incurs additional delays.

In contrast, we are inspired by **joint/group inference** [75, 132, 239], where we modify the model to be able to take features of up to P parallel I/Os and make one inference on behalf of all of them. Joint inference is more efficient than batching. Joint inference makes *one* inference on behalf of all of the P I/Os (imagine a green traffic light for P cars to pass through), while batching still requires the model to be run P times and make P decisions.

We have prepared a set of models that can take between 1 to P I/Os at a time. Storage operators

Components		Integration + Eval	
Dataset preparation	2.5 K	In User level	3.7 K
Design pipeline	3.6 K	In Linux kernel	2.1 K
Optimizations	1.2 K	In Ceph-Rados	2.3 K
Retraining	0.2 K	Evaluation module	5.3 K
Total: 20.9 K			

Table 3.2: **Implementation scale (§3.5).** HEIMDALL *is written in 20.9K lines of code (LOC) mainly in Python and C/C++.*

can decide which model they want to deploy. The challenge of developing joint inference models lies in the feature selection phase since we do not want to increase the model complexity by aggregating all input features from the *P* I/Os. Instead, we believe that the most up-to-date device behavior is reflected in the most recent I/Os. Thus, we need to prioritize the features from the most recent I/Os and ignore most of the features from the rest of the I/Os. For instance, for $P==5$, we still only supply the historical information of the last three I/Os before this group of 5 I/Os (§3.3.3), hence keeping the model light from the reduced redundancy. Later in Section 3.6.5, we will evaluate the tradeoffs, *e.g.*, higher *P* leads to higher throughput/performance but only reduces accuracy slightly.

3.5 Implementation Scale

Table 3.2 breaks down our 20.9 KLOC implementation of HEIMDALL pipeline. The left column shows the main components, which includes dataset preparation scripts, all the design stages (§3.3), deployment optimizations (§3.4), and retraining (later in §3.7), and the right column shows our three levels of integration in user-level storage (§3.6.1), Linux kernel (§3.6.2), and Ceph-Rados (§3.6.3), including our evaluation module that contains re-implementation of other policies.

3.6 Evaluation

Our comprehensive evaluation is set up as follows: • **Data size:** We use 2 TB of raw I/O block traces from Alibaba, Microsoft, and Tencent [2, 253, 370] and generate 11 TB of intermediate

data for all the experiments. • **Target deployments:** We integrate HEIMDALL into *user-level storage* (for fast, large-scale evaluation), *Linux kernel* (to mimic real deployments), and *Ceph* (for distributed evaluation). • **Machine and SSDs:** By default, we use Chameleon's Storage-NVMe node which has AMD EPYC 7352 2.30GHz 24-Core CPU with 256 GB DRAM. We use 10 different SSD models from various manufacturers[2] • **Train/test:** All the experiments use 50:50 train-test methodology which provides a more balanced representation compared to the 80:20 split.

3.6.1 Large-Scale Evaluation

To evaluate HEIMDALL comprehensively and **unbiasedly**, we conducted a large-scale evaluation with **hundreds of random time windows ("traces")** from various real-world traces from Alibaba, Microsoft, and Tencent [2, 253, 370]. To ensure that these hundreds of traces represent various workload characteristics, we picked the traces based on five criteria which are read/write ratio, size, IOPS, randomness, and overall ranking. For each criterion, we picked time windows with p10, p25, p50, p75, p90, and p100 values, with respect to all the time windows in the long, multi-day traces. To further increase the variability of our datasets, we also apply 5 different data augmentation functions (0.1× rerate, 0.5× rerate, 2× rerate, 2× resize, and 4× resize). From this large dataset pool, we **randomly** picked **500 traces**. Each trace is then capped at 3 minutes long, which contains between 100k to 10 millions of I/Os. Based on our experience, a 3-minute trace is long enough for the underlying devices to exhibit some tail latencies due to GC, write amplification, and other contentions, but at the same time, it is short enough to allow us to perform a large-scale experiment.

For faster experiment setups (just for this subsection), we deploy HEIMDALL in a user-level I/O replayer to simply mimic application-level storage with direct I/Os. For each experiment, we simulate a 2-way replicated storage environment using a machine equipped with 2 Samsung SSD

2. Intel (DC-S3610 and DC-P4600), Samsung (850-PRO, 970-PRO, PM961, PM1733, PM1725a, MZV-PV128HDGM, and MZH-PV128HDGM), and Hitachi SN260.

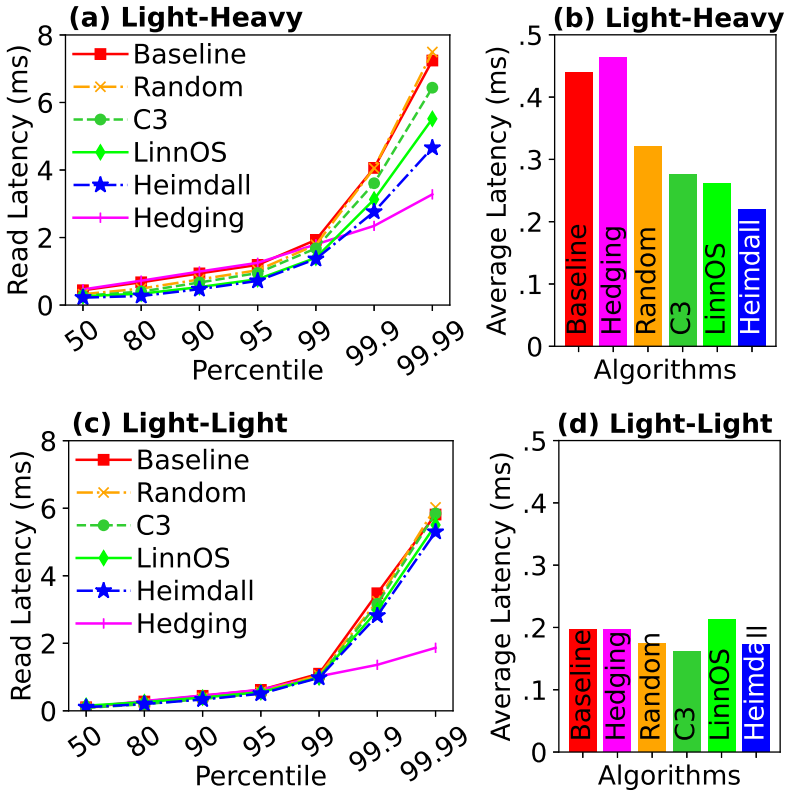


Figure 3.9: **Large-scale evaluation (§3.6.1).** *Subfigures **(a)** and **(b)** depict the read latency at percentiles ranging from p50 to p99.99 and the average latency measured under light-heavy workload combinations, while subfigures **(c)** and **(d)** present same measurements taken under light-light workload conditions.*

970 PRO 1TB SSDs. During each experiment, we run a random trace where the I/Os will pass through a HEIMDALL model specific for the device, which determines the I/O admission decision. If an I/O is declined, it is redirected to the other device, which will be admitted by default.

In our first set of experiments, we focus on "heavy-vs-light" traces, where one device sees a heavier workload than the other one. We categorize a trace as light if the I/O count is less than 300k. In this combination, it's essential to avoid blindly rerouting I/Os from the heavy trace to the light one. Doing so could inadvertently overload the other device, resulting in reduced overall performance. An efficient model should only decline and reroute I/Os necessarily.

We compared HEIMDALL performance against **baseline** (*i.e.*, always admit, no rerouting), **random** (*i.e.*, an I/O sent to a random target out of the two devices), **C3** [307], **LinnOS** [142], and

hedging [100], which are state-of-the-art admission/rerouting algorithms. **Figure 3.9a** shows the tail latencies (in the y-axis) at various percentiles (x-axis), concluding that HEIMDALL *outperforms state-of-the-art algorithms*. These latencies are the *average* percentile latencies from the 500 experiments, hence showing HEIMDALL *wins in large-scale, unbiased experiments*. Furthermore, **Figure 3.9b** shows another impressive outcome of HEIMDALL in terms of delivering the *lowest average latency*.

We can also see that while hedging delivers shorter tail latencies above p99, its average latency is far worse than HEIMDALL. For instance, hedging at p98 (between 0.75ms–1.5ms), after a 2ms timeout, a backup I/O is sent, causing too much overload (instability) which leads to *higher* average latency than the baseline. Therefore, hedging appears ineffective for low-latency requests.

Finally, to show a fairer experiment, we must show the performance of all these algorithms under "light-light" workload combinations where perhaps not much rerouting is needed (*i.e.*, more sensitive to over-rejection). **Figure 3.9c** shows that, even in light-light scenarios, HEIMDALL outperforms other methods slightly. **Figure 3.9d** presents a significant finding: HEIMDALL maintains a slightly faster average latency than baseline, but heavier models like LinnOS add more overhead to the baseline latency (more in Section 3.6.6).

3.6.2 Kernel-Level Evaluation

While the previous section shows a user-level deployment, this section briefly shows HEIMDALL results when deployed inside the Linux kernel block layer. As we already did a large-set experiment previously with homogenous datacenter Samsung SSD 970 PRO 1TB SSDs, here we provide a different setup by running a portion of Microsoft trace on a machine with two consumer-grade SSDs, Intel DC-S3610 and Samsung PM961. As demonstrated in **Figure 3.10a**, HEIMDALL also works effectively for in-kernel deployment and on heterogenous SSDs, outperforming other methods by delivering faster latencies (in the y-axis) at various percentiles (in the x-axis). **Figure 3.10b** further shows a *successful in-kernel deployment of* HEIMDALL, delivering the lowest average la-

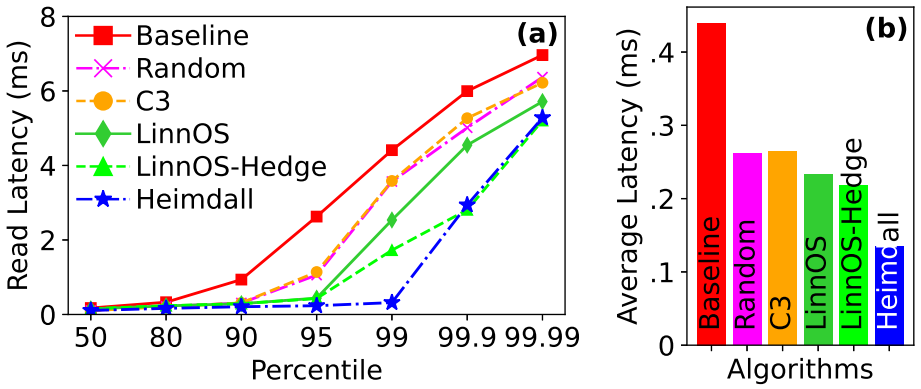


Figure 3.10: **Kernel-level evaluation (§3.6.2).** HEIMDALL *achieves* ***(a)*** *most stable latencies at p50 to p99.99 percentiles and* ***(b)*** *lowest average latency.*

tency, 38-48% faster compared to the non-baseline methods.

3.6.3 Wide-Scale Evaluation

So far, we have analyzed HEIMDALL on a single machine with multiple drives. This section shows HEIMDALL's performance with a "wide-scale" setup by deploying HEIMDALL in Ceph distributed storage system [326]. For this, we use 10 Chameleon Ice Lake machines, each with two 2.30 GHz 40-core Intel(R) Xeon(R) Platinum 8380 with 256 GB DRAM. On each machine, we deploy two Ceph Object Storage Daemons (OSDs) to set up a replicated storage with primary and secondary OSDs. Due to limited availability of dual SSD machines in the Chameleon cluster, we set up the OSDs on top of FEMU-emulated SSDs (100 GB each) [210]. To send requests to these 20 OSDs, we create 20 client nodes. Based on the results from the prior section, we now only compare three methods: baseline, random, and HEIMDALL. In the baseline Ceph setup, each request is initially directed to the primary OSD, whereas in the random setup, requests are randomly load-balanced. We also run noise injectors to see how these policies react to noisy neighbors. The latency CDF in **Figure 3.11a** shows that *in a wide-scale evaluation,* HEIMDALL *also delivers the best result.*

On top of this evaluation, we now change the "scaling factor" (*SF*). According to Google's seminal "Tail at Scale" paper [100], an end-user request can consist of multiple parallel "sub-requests" to different destination servers, but the end-user request is only considered complete

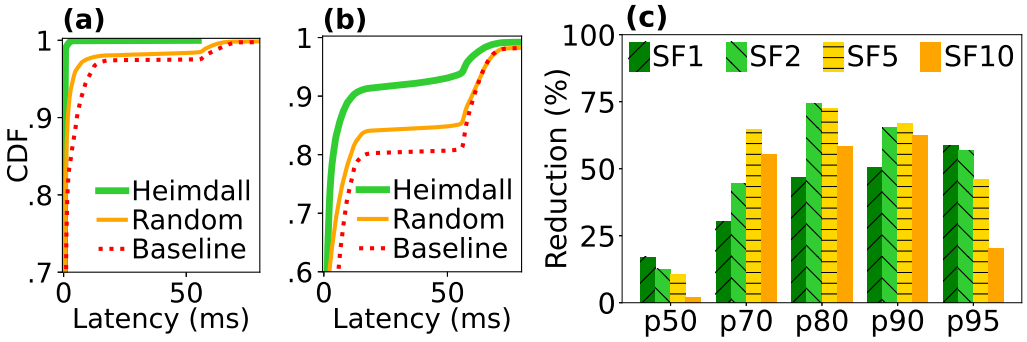


Figure 3.11: **Wide-scale evaluation (§3.6.3).** *CDF latency on Ceph cluster for* ***(a)*** $SF = 1$*,* ***(b)*** $SF = 10$*;* ***(c)*** *latency reduction at percentiles ranging from p50 to p95 across various SFs.*

when all the sub-requests are complete. To show HEIMDALL's benefits when the tail is amplified by scale, we vary the *SF* factor (*e.g.*, $SF = 10$ means an end-user request has 10 parallel sub-requests. **Figure 3.11b** shows the latency CDF when $SF = 10$, showing that tail behavior starts appearing in baseline at p75 and HEIMDALL can efficiently cut the large portion of the tail area. Furthermore, **Figure 3.11c** shows the tail latency reduction of HEIMDALL vs. random (in the y-axis) at various percentiles (x-axis) across a variety of scaling factors (as shown in the legend). HEIMDALL wins in all the scenarios (positive percentage of latency reduction).

3.6.4 Accuracy

Behind HEIMDALL's strong performance lies the high accuracy it achieves. *This section dissects how every design decision contributes to increasing* HEIMDALL*'s overall accuracy.* There are five main metrics of accuracy that we use: ROC-AUC, PR-AUC, F1-Score, FNR, and FPR, based on the number of true/false positives and negatives (TP, FP, TN, and FN, respectively). Their equations can be found here [98]. In our case, *true positive (TP)* implies that the model correctly identifies the I/O as "slow" and *false positive (FP)* implies the opposite (marked as "slow" but the I/O will actually be fast if submitted to the device).

In our domain, ROC-AUC is the preferred metric for imbalanced datasets as it balances sensitivity and specificity [72]. More specifically, the tail latencies are the minority compared to the

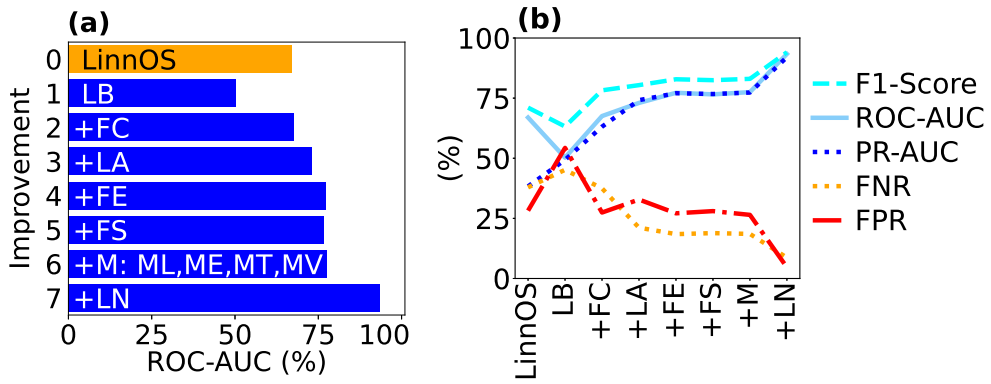


Figure 3.12: **Accuracy evaluation (§3.6.4).** *Contribution of each step; comparing **(a)** ROC-AUC only and **(b)** all metrics of LinnOS towards Heimdall design steps: Basic Labeling (LB), Feature Scaling (FC), Accurate Labeling (LA), Feature Extraction (FE), Feature Selection (FS), Model Engineering (M), and Noise Filtering (LN).*

fast latencies. Unlike `accuracy` or `precision`, ROC-AUC provides a comprehensive evaluation that considers the trade-off between true positive and false positive rates at various classification thresholds.

As shown in **Figure 3.12a**, we measure the resulting ROC-AUC score (x-axis) of every step-by-step optimization (y-axis). These optimizations contribute to enhancing both the dataset and the model which increases HEIMDALL's overall accuracy at every step of the way. The numbering, (*n*), below corresponds to the y-axis values in Figure 3.12a.

0. We first measured LinnOS' accuracy as a baseline, which stands around 67% on average, across over 100 random datasets. LinnOS' accuracy dropped significantly compared to what the authors reported in the paper four years ago. The model is outdated given the different behavior of modern SSDs (faster latencies) and more recent workloads released by the community, which highlights the necessity to re-design the model with a more extensive ML pipeline.
1. Here, we started HEIMDALL with LinnOS' original feature set, model, and, *basic cutoff-based labeling (LB)* without implementing the digitization because we wanted to establish a baseline performance, resulting in a drop of accuracy by 16.8%, to 50.2%.
2. We then applied *min-max scaling (FC)* (§3.3.3) to normalize the input features instead of

using digitization (as in LinnOS), resulting in a slightly better accuracy (67.5%) compared to LinnOS.

3. Afterward, we crafted our more *accurate period-based labeling (LA)* method (§3.3.1) and used it to replace LinnOS' cutoff-based labeling. As a result, we increase the accuracy by 5.5%, bringing it to 73%.
4. Then, with additional *feature extraction (FE)* (§3.3.3), including I/O size and historical throughput, the accuracy gains another 4% improvement (now reaching 77%) as HEIMDALL now can discern patterns related to the volume of data being transferred and the transfer rates.
5. To reduce the model's inference overhead, we applied a *feature selection (FS)* method (§3.3.3) that maintains accuracy without inducing degradation.
6. Likewise, we employed *model engineering (M)* (§3.3.4 and §3.3.5) steps consisting of learning task exploration (ML), model exploration (ME), hyperparameter tuning (MT), and validation (MV), to find a balance between the model's complexity and accuracy. As a result, we obtained a minimalist model capable of maintaining the same level of accuracy achieved so far.
7. Finally, our *noise filtering (LN)* (§3.3.2) proves to be one of the most important contributions, which increases the accuracy by 16%, leading to 93% model accuracy.

Later in Section 3.6.7, we demonstrate how AutoML models can also benefit from these optimizations. Furthermore, to show that we are not biased over one accuracy metric, **Figure 3.12b** shows the resulting accuracy (in the y-axis) across the five accuracy metrics (the five lines) as we add the step-by-step contributions (in the x-axis). Overall, as more optimizations are introduced, ROC-AUC, PR-AUC, and F1-Score continue to increase. Furthermore, the false-negative and false-positive rates (FNR and FPR) also continue to decrease as desired.

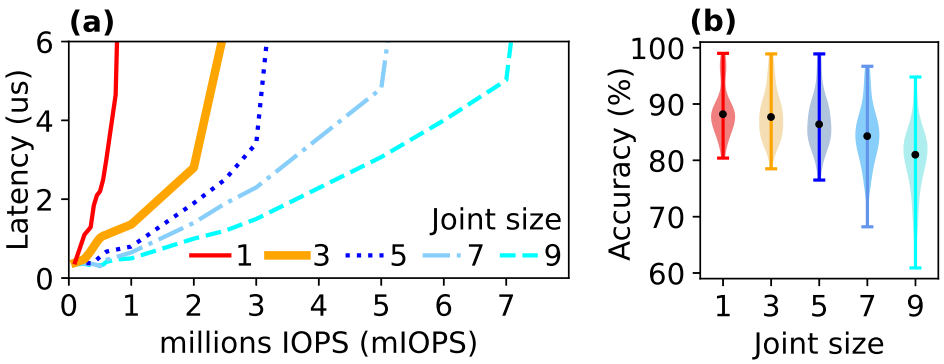


Figure 3.13: **Joint inference (§3.6.5).** ***(a)*** *Throughput stability on various joint sizes and* ***(b)*** *model's accuracy.*

3.6.5 Joint/Group Inference

We now discuss HEIMDALL's joint inference performance (§3.4.2). As shown in **Figure 3.13a**, with the default setting of HEIMDALL (without joint inference, shown by joint size = 1), it can only receive 0.5 mIOPS workload before its latency spikes to 2µs. However, with a joint size = 9, HEIMDALL maintains latency under 2µs even with a 4 mIOPS workload on 1 CPU core, an 8× heavier workload than the default HEIMDALL. Note that this latency includes queueing delay, rendering it slower than our fastest inference latency recorded at §3.4.1.

Joint inference reduces accuracy, as quantified in **Figure 3.13b**. For example, transitioning from the default HEIMDALL to 9 I/O joint inference, the accuracy drops from 88% to 81% in the median value. The figure shows the resulting accuracy distribution across 50 random datasets. Given the results above, we believe that joint size = 3 is appropriate for balancing out the throughput/accuracy tradeoff. When deploying HEIMDALL, storage administrators can also adjust the joint size dynamically.

3.6.6 CPU and Memory Overhead

We now assess HEIMDALL memory and CPU overhead. **Figure 3.14** shows that, compared to LinnOS, our model achieves **(a)** 2.4× less memory overhead, 68KB vs. 28KB, and **(b)** 2.5×

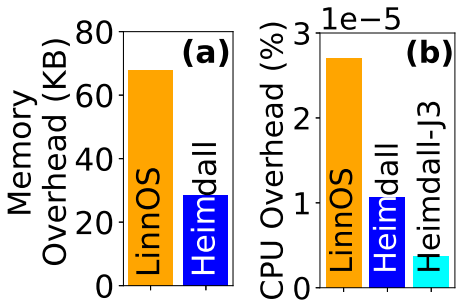


Figure 3.14: **Overhead.** ***(a)*** *Memory overhead and****(b)*** *CPU overhead.*

less CPU overhead. HEIMDALL has a total of 3472 weights and biases, which is smaller than LinnOS' total of 8706 weights and biases. Furthermore, our model has 2.4× fewer multiplication operations compared to LinnOS, with 3472 vs. 8448 multiplications. Moreover, HEIMDALL makes significantly fewer inference decisions since it operates on a per-I/O basis instead of per-page decisions like in LinnOS. To further reduce the overhead, HEIMDALL can be deployed with joint size = 3 (represented by HEIMDALL-J3, light-blue bar), resulting in 85% less CPU overhead compared to LinnOS.

3.6.7 HEIMDALL *vs. AutoML*

HEIMDALL is the result of applying various machine learning methods in a disciplined, careful, and meticulous way with deep domain knowledge. However, one might wonder *if Automated Machine Learning ("AutoML" for short) can outperform our manual approach*. We use `auto-Sklearn` [115], a state-of-the-art framework that automates ML algorithm selection and its hyperparameter tuning.

In the first experiment, we randomly picked 50 datasets and supplied the *raw* dataset to the AutoML framework without the manual feature engineering steps that we did (§3.3.3). We use *16 AutoML classifiers* which consist of various algorithms (as detailed in the figure caption above) such as generalized-linear, support vector machine, neighbors-based, Naive-Bayes, tree-based, discriminant-based, tree-based ensemble, and layer-based algorithms. This is deliberate to see how much AutoML methods can help with the user exerting the least possible action.

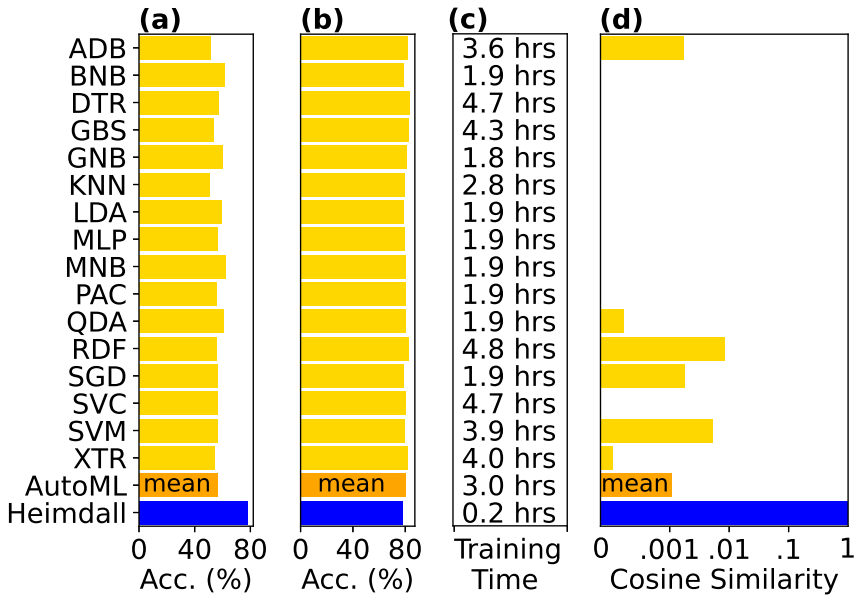


Figure 3.15: **HEIMDALL vs AutoML (§3.6.7).** *Generalized-linear algorithms such as Linear Discriminant Analysis (LDA) and Passive Aggressive Classifier (PAC); support vector machine algorithms such as LibSVM Support Vector Machine (SVM) and Linear Support Vector Classifier (SVC); neighbors-based algorithms such as K-Nearest Neighbors (KNN); Naive-Bayes algorithms such as Bernoulli Naive-Bayes (BNB), Gaussian Naive-Bayes (GNB), and Multinomial Naive-Bayes (MNB); tree-based algorithms such as Decision Tree (DTR); discriminant-based algorithms such as Quadratic Discriminant Analysis (QDA) and Linear Discriminant Analysis (LDA); tree-based ensemble algorithms such as Adaboost (ADB), Gradient Boosting (GBS), Random Forest (RDF), and Extra Trees (XTR); and layer-based algorithms such as Multi-Layer Perceptron (MLP).*

Using `auto-Sklearn`, AutoML autonomously conduct hyperparameter tuning to craft the best model configuration. **Figure 3.15a** compares the ROC-AUC accuracy (x-axis) of AutoML models (top rows) and HEIMDALL (last row). *AutoML models exhibit suboptimal performance, with an average accuracy 22% lower than* HEIMDALL. This outcome aligns with our expectation: given that AutoML exclusively utilizes the raw feature set (size, offset, type, etc.) and does not leverage domain-specific derived features (*e.g.*, queue length), AutoML models are suboptimum in identifying meaningful correlations between the feature sets and the resulting label. Although one could argue that we can configure the AutoML framework to explore derived features, the process will blow up the execution time (*e.g.*, to determine the number of historical I/O features as discussed in §3.3.3).

In the second experiment, *we obtained 24% accuracy improvement on AutoML models by supplying* HEIMDALL*'s fully engineered dataset from our accurate labeling and feature engineering steps* (§3.3.1 and §3.3.3). The improved AutoML results shown in **Figure 3.15b** solidify the goals of our work, showing the importance of exploring other stages of a long machine learning pipeline in a disciplined way. Here, the AutoML models exhibit a slightly higher accuracy (2% on average) than HEIMDALL, which is expected because the size of the AutoML models is not bounded, while in our case, we need to balance the complexity of the model (*e.g.*, #layers, #neurons) to maintain low inference overhead.

In **Figure 3.15c**, we also demonstrate that *AutoML models are impractical for our problem domain.* Specifically, *the training time (in the x-axis) for AutoML models is on average 15× longer than HEIMDALL*, ranging from 1.8 to 4.8 hours compared to just 0.2 hours. This is due to the explorative nature and unbounded complexity of the AutoML models, resulting in significantly longer training time compared to HEIMDALL. Note that this experiment is conducted on CPU instead of GPU because `auto-Sklearn` does not support GPU acceleration by default. Furthermore, HEIMDALL's training time on the CPU can be further improved (out of the scope of this paper) by porting the training code into C/C++ and using CPU-optimized training libraries. AutoML models, on the other hand, are hard to optimize due to their complex exploration strategies and heavy reliance on various Python-based ML plugins and dependencies.

Additionally, *the models generated from AutoML have widely different architectures* (*e.g.*, #layers, #neurons) across the datasets, while HEIMDALL, on the other hand, is agnostic to the dataset and doesn't need to perform hyperparameter tuning when being trained on a new dataset. **Figure 3.15d** quantifies this problem. Here, for each approach (in the y-axis), we compute the cosine similarity (x-axis, in log-scale) between each approach's configurations across the datasets. While HEIMDALL*'s similarity stays at 1, AutoML models exhibit poor generalization as many of them have cosine similarity of lower than 0.01*, which makes them unsuitable for production systems. Specifically, they pose a major challenge when deployed in a continuous training setup as their

specificity to particular workloads requires them to re-explore and re-discover the best hyperparameter tuning with substantial retraining costs (more in Section 3.7).

3.6.8 Training Time

Finally, we report the average training time of HEIMDALL, which depends on the number of I/Os. For every 1 million I/Os we use for training, it takes us 16.8 seconds of preprocessing time on CPU and 3.7 seconds of training time on GPU. Preprocessing includes labeling, extracting features, normalizing, and shuffling the data. The next question to answer is how much data to train on and how should we retrain the model in a long deployment scenario. To us, this represents another significant research question that falls outside the scope of the paper. Answering it would necessitate further extensive exploration of the ML pipeline, as we will delve into in the next section.

3.7 Retraining for Longer Deployment

In reality, ML models are deployed for the long term. In this context, we conduct a *preliminary long-term evaluation of* HEIMDALL *by testing it on an 8-hour real-world trace with one of the most "challenging" traces where accuracy fluctuates in the long run*. Here, we used a Tencent trace where the write IOPS is 2× more than the read IOPS, triggering more GC activities. Furthermore, this trace exhibits an almost constant I/O interarrival time, causing all devices to experience similar workloads and heavy utilization simultaneously.

Figure 3.16a shows HEIMDALL's accuracy after a single training session using the initial 1, 5, and 15 minutes of the trace. We measure the accuracy within a 10-minute window (a dot is the model's average accuracy in the last 10 minutes). We can reach two simple conclusions. First, a longer training trace (*e.g.*, a 15-minute trace) results in better long-term overall accuracy, but requires longer training time (§3.6.8). Second, accuracy fluctuates over time, with a min-max accuracy of 63%–82%. This is also known as *model's performance drift*, which could stem from factors like shifts in workload behavior (input drift), device/environment changes (concept drift),

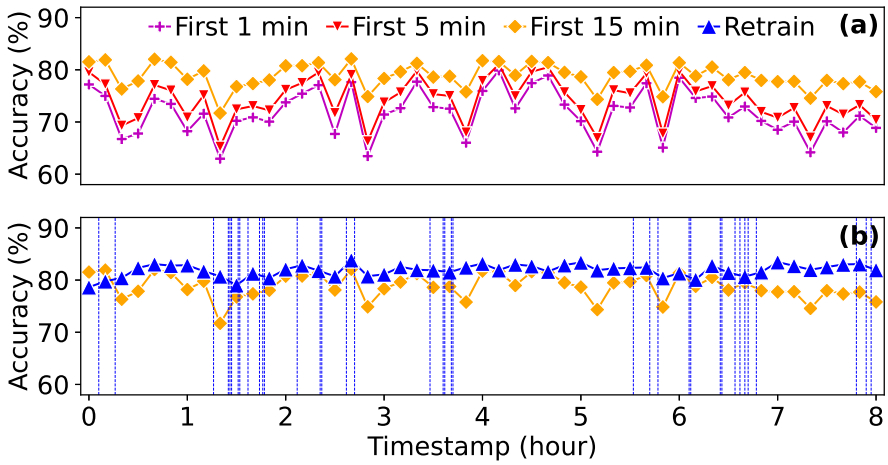


Figure 3.16: **Long-term evaluation (§3.7).** HEIMDALL *performance with different training methods: "First N min" trains the model once using only the first N minutes of the workload, while "Retrain" uses a simple retraining strategy described in §3.7. Vertical blue lines mark the time when retraining was triggered.*

and others [329].

To address this, we built a preliminary *retraining policy* that monitors the model's accuracy every minute and triggers retraining when the accuracy drops below 80%. To keep retraining light, we only retrain using the last 1 minute before the trigger. **Figure 3.16b** shows the result where the vertical blue lines represent the times when retraining is triggered. More specifically, within this 8-hour window, retraining occurs 37×, each utilizing an average of 816k I/Os which can be completed in a couple of seconds (§3.6.8). *This initial result also points to more research questions.* For example, the presence of consecutive retraining instances (vertical blue lines that are close to each other) suggests that some retraining decisions may not be useful. Second, we cannot expect the last 1-minute trace before the retraining session is available because per-request logging is turned off by default due to the significant overhead [79].

Overall, these findings suggest that *there are more topics to explore in this long machine learning pipeline for storage* such as efficient retraining, continual learning models [196, 206, 234], and model zoo/management [160, 173, 280, 290]. There are many research questions to ask. How often do we need to retrain the model? When retraining, how much recent data should be used? How to avoid catastrophic forgetting during retraining? To speed up retraining, will data sampling

work? Which sampling methods can generate highly representative data? After retraining, can the current model adapt to the new workload behaviors while still remembering past behaviors? How do we detect performance drifts? What are the I/O characteristics that can provide hints of workload drifts? How to know what type of drift (input/environment/etc.) is happening? How often and how much data to use to check for such drifts? Will continual learning model require heavy computational resources to continuously fine-tune its knowledge? If so, are model zoo/management methods more efficient and what sets of parameters can be used to activate different models?

3.8 Conclusion and Competitions

This paper shows that extensively exploring various stages of ML pipeline brings many accuracy and performance benefits. HEIMDALL can also serve as a platform to train future "data scientists for storage systems." We have used it as a testbed for "mini competitions," motivated by popular competitions such as ImageNet and Kaggle [15, 22]. We had 15 students participate, with each focusing on different challenges (*e.g.*, "find/create new datasets where the current ML models perform poorly", "build a better model or a labeling method", etc.). They explored 20 forms of data augmentation, built 35 classification- and 20 regression-based models, and tried other training strategies (sampling, quantization, etc.). We will release HEIMDALL publicly and hope it can also serve the community and be extended to cover other important storage policies.

CHAPTER 4

TINFETCH: A TINY NEURAL NETWORK FOR BLOCK PREFETCHING VIA PRECISE ADDRESS SEPARATION AND SUFFIX PREDICTION

4.1 Overview

In the contemporary landscape of high-performance servers tailored for data centers and the demanding field of AI/ML training, the dependence on solid-state drives (SSDs) and spinning disks (HDDs) as secondary storage devices take center stage. The integral role played by these storage devices in dictating the overall system performance accentuates the need to address and minimize their I/O latency, especially in a hybrid storage system [176]. Caching and prefetching emerge as prevalent strategies to mitigate the high access latency of storage devices. Caching involves using faster but less dense memory to store frequently accessed data [189, 298]. Prefetching [77], whether implemented in software within the operating system [213, 249, 378] or directly within the SSD/HDD firmware [343], is introduced as a crucial technique for reducing latency by fetching data from their original storage in slower memory to cache before they are needed.

Typical block-level cache prefetchers operate by receiving input in the form of logical block address (LBA) sequences, represented as integer numbers. These prefetchers predict the LBA of the data that is likely to be accessed shortly and decide on whether to prefetch it or not. Efficient prefetcher design encounters two primary challenges. First, real-world applications demonstrate complex LBA access sequences due to random accesses from diverse users or applications, which are common in modern large-scale storage systems [63, 217]. Second, the precision of prefetchers is vital for their effectiveness. Inaccurate prefetchers result in inefficient use of I/O bandwidth and cache space [144]. Consequently, the development of effective prefetchers holds significant importance for storage systems.

Prefetchers are mostly heuristics going back 1-2 decades ago [91, 104, 152, 167, 197, 223,

258, 259, 266, 282, 299, 338, 382] where they heavily rely on predefined rules to prefetch data based on Logical Block Address (LBA) access sequences. However, they struggle with adapting to complex real-world scenarios due to their rigid pattern detection technique. For example, the read-ahead prefetcher [114, 193] is restricted to prefetch the next data item within a file for faster sequential accesses. To address this limitation, various learning-based methods are developed [83, 325, 335], including recent Long Short-Term Memory (LSTM) techniques like DeepPrefetcher [124] and Delta LSTM [80], which model the LBA delta (difference between successive access requests) and cover a broader range of LBAs. However, these methods ignore internal temporal correlation during concurrent accesses, leading to limited performance. More advanced prefetchers [140, 357] capable of learning complex I/O access patterns are often hard to deploy due to their computational cost. Accurately predicting future I/O accesses remains challenging, especially in real-world applications where sequential I/Os are mixed with random requests from multiple users performing independent tasks simultaneously. The transformation of straightforward accesses on the application side into seemingly random accesses on the SSD/HDD level adds complexity to the prediction challenge [71].

In this paper, we take the following position: *an ideal solution should be the combination of an efficient heuristic approach and an accurate learning-based method that can dynamically adjust its aggressiveness to minimize cache pollution*. Moreover, an efficient prefetcher can be deployed into the production system with negligible computation and space overhead. Unfortunately, existing prefetching algorithms fall short for several reasons. First, they are designed to only utilize either the heuristic (pattern-based) approach or the complex learning-based method. Second, the deployment of the learning-based methods is complicated due to their intricate environment set up for enabling the continuous training pipeline. Finally, they cannot quickly adapt to temporal changes in page access patterns which results in over-polluting the cache when being optimistic and under-utilizing the cache as a result of being pessimistic.

In this work, we propose TINFETCH, a **Ti**ny **N**eural Network for block pre**fetch**ing via pre-

cise address separation and suffix prediction. Unlike existing prefetching algorithms that rely on detecting and learning specific LBA delta patterns, TINFETCH works by employing an adaptive heuristic method to disentangle complex interleaved I/O streams and using a pretrained Neural Network model to predict future LBA access based on the suffixes. Specifically, TINFETCH divides the prefetching algorithm into two modules to form a cohesive hybrid solution, the adaptive (heuristic) module and the predictive (learning-based) module. First, the adaptive module focuses on disentangling the random accesses and tuning the prefetching aggressiveness. While the disentanglement makes TINFETCH resilient to short-term irregularities in access patterns (e.g., due to multi-threading), the aggressiveness tuning helps it to minimize cache pollution. Second, the predictive module focuses on predicting the future LBA access with the assistance of a small pretrained model. This design enables TINFETCH to be deployed in any storage system without the necessity of a pre-installed ML/AI pipeline, often requiring GPU or TPU (Tensor Processing Unit).

We evaluate TINFETCH against practical real-time prefetching algorithms (Stride, Read-ahead, Markov Chain) and various state-of-the-art algorithms (SGDP [357], Leap [235], Pythia [67], Delta LSTM [80]) on real production traces from three major companies (Alibaba [40], Microsoft [25], and Tencent [33]) and FIO-generated [12] traces. Evaluated on various production and synthetic traces, TINFETCH achieves a 3% to 27% improvement in hit rate compared to state-of-the-art heuristics and ML-based prefetchers. Additionally, it has the highest hit rate per storage bandwidth load of 0.2, surpassing the best state-of-the-art learning-based and heuristic-based prefetchers by 2.4×and 1.6×respectively. Furthermore, our multi-dimensional evaluation shows that TINFETCH strikes a remarkable balance between a high hit rate and minimal storage bandwidth load, outperforming the second-best by 10% in hit rate. We provide a low-overhead, practical implementation of TINFETCH on a C/C++ language that is optimized for production deployment which achieves inference latency in sub-μs, tested on 2.6 GHz Intel Core i9.

We make the following contributions in this paper:

- We observe three key shortcomings in prior prefetchers that significantly limit their perfor-

mance benefits: (1) the use of only LBA delta for pattern detection without a dedicated disentanglement method, (2) lack of integration between the heuristic and learning-based method, and (3) lack of generalization capability on the existing learning-based methods since they always require retraining to adapt to the ever-changing workloads.

- We present TINFETCH, a hybrid prefetcher that combines the effectiveness of heuristic methods and the learning capability of a Neural Network model. To the best of our knowledge, TINFETCH is the first prefetching solution that pioneers the use of a pretrained model and a precise address separation method for disentangling interleaved I/O streams.
- We introduce *storage bandwidth load* as an important metric for effectively evaluating prefetching algorithms, given its agnostic nature towards cache size and caching algorithms. This metric, when combined with the hit rate, provides a comprehensive assessment of both cache pollution and storage bandwidth utilization.
- By extensive evaluation, we show that TINFETCH outperforms prior state-of-the-art prefetchers over a wide variety of workloads in a wide range of metrics analysis.

4.2 Background

We first briefly review the basics of prefetching and its metrics. We then describe various prefetching approach, from the traditional to the learning based prefetching. Finally, we explain why Neural Network is a good framework for designing a prefetcher that fits our goals.

4.2.1 Prefetching

In storage systems, prefetching plays a crucial role in enhancing performance by proactively fetching data anticipated to be accessed in the near future. These data will be preloaded from a slow storage device (SSD/HDD) into faster memory, generally DRAM, to decrease the overall read latency. An accurate and efficient prefetcher is important in reducing the performance gap between different tiers of storage systems [224]. There are three common metrics to evaluate the perfor-

mance of prefetching algorithms which are **hit rate**, **accuracy**, and **timeliness**. While hit rate evaluates the success of prefetching by assessing how much of the prefetched data is found in the cache upon subsequent access, accuracy measures the ratio of the number of prefetched data items that were subsequently accessed (correct predictions) to the total number of data items that were prefetched. A high hit rate suggests that the prefetched data is effectively utilized, reducing the need to fetch the same data from slower storage locations and improving overall system efficiency. A high accuracy emphasizes the precision of predictions made by the prefetching algorithm. Furthermore, the measurement of these two metrics must obey the timeliness meaning that the hit rate and the accuracy must be calculated based on the data blocks that were prefetched sufficiently ahead of time so that the data is present in DRAM in advance of the actual demand.

Traditionally, the prefetching performance is always tied to the cache size and caching algorithm that is being used during the evaluation. A prefetcher can get a very low hit rate when the cache size is under a certain threshold depending on the traces, and vice versa. The common solution is to provide the same cache size and same cache eviction policy when evaluating them, but it means that all algorithms must be implemented and integrated into the same systems for fairness which increases the complexity dimension of the evaluation. To answer this challenge, we introduce a **storage bandwidth load** (or *bandwidth-load* for short) as another important metric for measuring the performance of a prefetcher. This metric can effectively evaluate the prefetching algorithms by isolating their performance measurement from the cache size and the caching algorithms. Specifically, storage bandwidth load measures the bandwidth load on the secondary storage (SSD/HDD) during the prefetching simulation with unlimited cache. The load is affected by the I/O workload and the prefetched I/Os. For example, when we don't utilize the cache/memory, the bandwidth-load will be 100% because all of the requested I/Os will go directly to the storage (SSD/HDD). When prefetching is enabled, the prefetched I/O will also be counted towards the total storage bandwidth load. This prefetched I/O could end up reducing the storage bandwidth load if it is frequently requested. However, it can also significantly increase the storage bandwidth load

if the prefetching prediction is not accurate causing the storage (SSD/HDD) to be busy fetching the data instead of serving the actual I/O from the user workload.

In general, the higher the storage bandwidth load, the more likely that the prefetcher is over-polluting the cache. While a low storage bandwidth load signifies that the load to the secondary storage device is reduced because most of the requested data blocks can be found in the memory. Hence, the ideal prefetcher has high hit-rate, high accuracy, and low storage bandwidth load. These metrics collectively provide insights into the effectiveness, efficiency, and impact of prefetching on the overall system performance, enabling a comprehensive assessment of prefetching algorithms in diverse computer system environments.

4.2.2 Basic Heuristic Prefetcher

This class of prefetching algorithms relies on straightforward rules or heuristics to anticipate and fetch data in advance of explicit requests. These heuristics are designed to recognize predictable patterns within data block access sequences, and two common variants of this approach are the Stride Prefetcher [120] and the Read-Ahead Prefetcher [114]. The Stride Prefetcher operates on the concept of "stride", which denotes the fixed interval between consecutive accesses. When a regular pattern is discerned, the Stride Prefetcher predicts the next access by adding a constant stride value to the current address. It issues prefetch requests for future addresses based on this observed stride, anticipating sequential access. The Read-Ahead Prefetcher adopts a different strategy by focusing on fetching contiguous blocks of data beyond the currently requested block. It operates on the assumption that if one block is accessed, subsequent blocks are likely to be accessed soon. The Read-Ahead Prefetcher, therefore, issues prefetch requests for these subsequent blocks, aiming to bring them into the cache before they are explicitly requested. Despite their simplicity and low computational overhead, these prefetchers are designed based on specific assumptions and may underperform when these assumptions are violated. One significant limitation is their lack of adaptability to dynamic changes in access patterns.

Another line of work utilized Markov chains [89] for prefetching data from SSDs [193, 343]. We compare our approach with these prior works in Section 4.6, confirming prior observations that Markov chain-based prefetchers have poor accuracy resulting in a high storage bandwidth load on real-world traces where the I/O streams are more complex [285]. In addition, some variants of heuristic prefetchers learn irregular access patterns by memorizing pairs of correlated LBAs [218, 328, 333, 334]. However, that method won't be effective in dynamic access pattern scenarios with diverse and inconsistent correlation address pairs. As a result, these traditional methods cannot achieve good performance in practice which fueled the need to develop more advanced prefetching techniques, including learning-based approaches.

4.2.3 Learning-based Prefetcher

A Learning-Based Prefetcher represents an advanced class of prefetching algorithms that leverages machine learning techniques to dynamically adapt to and learn from the complex and evolving access patterns. Unlike Basic Heuristic Prefetchers, which rely on predefined rules or simple heuristics, Learning-Based Prefetchers can analyze historical access patterns, identify correlations, and make predictions based on learned models. One of the primary benefits of Learning-Based Prefetchers is their adaptability. These prefetchers can dynamically adjust their prefetching strategies based on changing access patterns, making them more resilient in scenarios where the assumptions of heuristic-based approaches may fall short. By utilizing advanced machine learning models, such as neural networks or reinforcement learning, these prefetchers can capture dependencies and correlations that may be challenging for rule-based prefetchers to discern.

In general, the prefetching algorithm can be treated as a prediction problem which can be done using regression-based or classification-based models. Of those two approaches, classification-based methods are more popular than regression because they can cover specific/targeted LBA accesses, but not practical as it is hard to cover all possible LBAs. To reduce the prediction complexity, many learning-based methods are using the LBA delta as the input feature [144], instead of

the raw LBA. DeepPrefetcher [124] transforms the LBA sequence into LBA deltas, then employs the word2vec model and LSTM architecture to capture the hidden feature in the input sequence. Hashemi [144] used neural network based sequence models for prefetching DRAM accesses. To improve the adaptability where the I/O sizes are fluctuating, Delta LSTM is proposed [80] and includes I/O size as part of the prediction target. However, relying on the LBA delta makes them vulnerable to interleaved I/O access pattern because the delta calculation will take the randomly accessed I/O sequence as it is without applying any disentanglement method which will result in a hard-to-predict delta pattern. Moreover, due to the vast amount of computation they require for inference, these models' inference latency is much higher than the acceptable latency of a prefetcher at any cache level making them impractical and hard to deploy.

Another advanced method is by utilizing graph. Complex patterns in LBA access streams can be represented as graphs [107, 220]. Nexus uses metadata relationship graphs to assist prefetching decision-making [133]. Ainsworth et al. design a prefetcher for breadth-first searches on graphs [47]. SGDP models the LBA delta streams using a weighted directed graph structure to represent interactive relations among LBA deltas [357]. Unfortunately, the inherent sparsity of LBA accesses leads to the generation of expansive graphs in sequence-based approaches, rendering these prefetchers less effective in practical applications. The challenge lies in the limited density of LBAs, hindering the ability of these methods to generate compact and actionable graphs that truly capture the intricacies of block access patterns in real-world scenarios. To overcome the model complexity and the overhead of Neural-Network or Graph-based models, Pythia implements a Reinforcement Learning (RL) model that can predict the top-N delta efficiently since it can continuously learn online by iteratively optimizing its policy using the rewards received from the environment [67]. However, this model tends to cause excessive storage bandwidth load as we evaluated in Section 4.6.

Overall, while Learning-Based Prefetchers offer significant advantages in adaptability and accuracy, they also face specific challenges that warrant consideration. One of the primary challenges

lies in the deployment complexity associated with these advanced models and the continuous training pipeline (CT) to update their models based on real-time (online) observations. This CT pipeline is complicated to deploy due to its intricate setup of the re-training framework, drift detection, and data collection methods. These shortcomings are not apparent in small-scale testing, but they result in limited performance when deployed in a wide-scale production setup. Therefore, the development of a tiny pretrained model offers a promising avenue to address the challenges associated with Learning-Based Prefetchers.

4.3 TINFETCH: Tiny NN Prefetcher

In this section, we first highlight the key idea of our hybrid prefetcher along with its different components and the design principles behind them. Finally, we discuss how we develop TINFETCH' pretrained model. The overall design of the TINFETCH systems is illustrated by **Figure 4.1**.

4.3.1 Key Idea

We believe that an ideal solution to prefetching challenges lies in a synergistic combination of an efficient heuristic approach and an accurate learning-based method. Such a hybrid model, characterized by its dynamic adjustability, ensures optimal aggressiveness to minimize cache pollution effectively. Moreover, the success of an efficient prefetcher is contingent upon its seamless integration into production systems. Achieving this integration necessitates minimal computation and space overhead, emphasizing the importance of a solution that not only enhances prefetching efficacy but also aligns with the practical constraints and efficiency requirements of real-world computing environments.

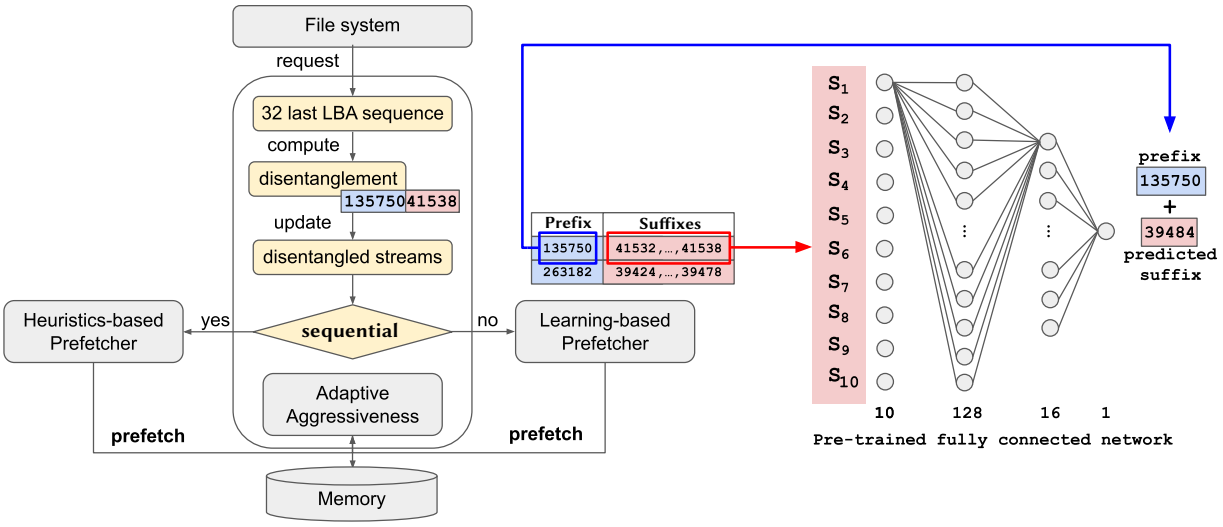


Figure 4.1: **TINFETCH System Design.** TINFETCH *is design as a hybrid-prefetcher which combines the efficiency of a heuristic method and the accuracy of a learning-based method. As part of the heuristic method,* TINFETCH *utilizes a precise address-separation method to disentangle the past LBAs.* TINFETCH *then utilizes its hybrid approach to directly prefetch upon a clear continuous pattern; otherwise, it uses a pre-trained neural network model to predict the suffix of the memory address to be prefetched.*

4.3.2 Main Components

To bring that idea into realization, we developed a modular prefetching solution that consists of two main components:

1. **Adaptive Module**

   This component refers to a specialized heuristic technique employed within TINFETCH to disentangle or separate complex interleaved Input/Output (I/O) streams. Its primary objective is to unravel intricate patterns within concurrent and random accesses. TINFETCH disentanglement process shares similarities with the stride prefetcher's stream detection, yet it boasts superior accuracy owing to its narrower range of LBA suffix. Furthermore, this module is also responsible for the adaptive aggressiveness behavior of TINFETCH. The aggressiveness is tuned based on the effectiveness of the prefetcher. For example, if there are very few hits in the last-N accesses, TINFETCH will pause the prefetching action for a while. When the current hit rate is high, TINFETCH will increase the prefetch size which makes

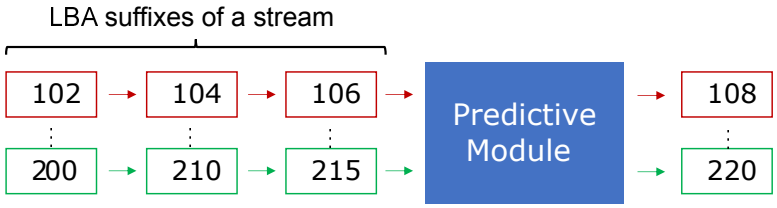


Figure 4.2: **Predicting the next suffix.** TINFETCH *uses LBA suffixes to predict the future LBA. The suffixes are provided by our effective heuristic method that disentangles interleaved I/Os into sets of streams based on their LBA prefixes. By having a modular design that separates the disentanglement process from the predictive module, we can optimize the accuracy of each module and reduce the overall complexity of our prefetching systems.*

it more aggressive. These adaptive behaviors are managed efficiently by a smart heuristic method, without the need for complex learning techniques.

2. **Predictive Module**

   Unlike the previous module that relies on heuristic methods, this module is designed to be a hybrid that combines the best of both heuristic and learning-based approaches. It's all about predicting the next number in a sequence of LBA suffixes as depicted at **Figure 4.2**. The heuristic prediction is made by an efficient read-ahead prefetcher that will prefetch the next LBA if it detects a consecutive pattern. To handle complex sequence where there is no obvious consecutive pattern within the sequence of suffixes, we use a learning-based prefetcher that utilizes a pretrained Neural-Network model.

4.3.3 Disentangled Streams

We disentangle the addresses by using modulo operation which can be done efficiently with a bit-wise operation. The granularity of the prefix is 8MB. In other words, a single prefix will represents any accesses within 8MB of contiguous data block. TINFETCH maintains 32 last LBA sequences organized in the form of disentangled streams. Each of the streams stores the list of LBA suffixes in the same order as it arrives. To achieve an O(1) lookup speed, we use a dictionary data structure where the *prefix* is the key and the array of *suffixes* is the value. Given this prefix-suffix dictionary,

the predictive module will check how many suffixes are recorded under the current address prefix. If there are more than three suffixes, the inference process will be triggered utilizing the given suffixes as the input feature. Otherwise, the model will not prefetch. The details of our ML model will be explained in the next section.

4.3.4 Pretrained Neural Network

The decision to use a pretrained model is due to the deployment consideration. Without the need to re-train, we can easily deploy TINFETCH in any storage system, including in the storage devices' firmware. Using a Neural Network to predict the next suffix in the sequence of LBA suffixes is effective due to its ability to handle non-rigid patterns. Unlike static heuristic-based prefetchers, Neural Networks adapt and learn from data, making them well-suited for sequences with varying complexities. However, efficiently training the Neural Network for sequence prediction is challenging because it requires striking a balance between model complexity and avoiding overfitting. Overfitting occurs when the model memorizes the training data instead of learning general patterns, leading to poor performance on new, unseen data. Therefore, optimizing the network architecture, selecting appropriate hyperparameters, and employing regularization techniques are crucial tasks in overcoming this challenge. Additionally, obtaining diverse and representative training data is essential to ensure the model can effectively generalize to different patterns and variations within sequences.

To overcome those challenges, we design TINFETCH's Neural Network model as compact as possible while still maintaining the accuracy. This model does a regression to predict the future LBA suffix that ranges between 0 to 8192 (2^{13}). We strategically limit this range so that the predicted block remains within 8MB of the HDD's head-seek from its current position. This approach enables TINFETCH to be deployed in HDD firmware without introducing seek overhead when fulfilling prefetch requests. In SSD deployment, TINFETCH can be seamlessly integrated without any design consequences, as SSDs can efficiently fetch any random data without head-seek

overhead. In terms of the model, this limited range of suffixes not only enhances precision but also contributes to a decrease in the overall complexity of TINFETCH's predictive module. Once trained, the weights can be used to perform inference in any platform and language with a forward-propagation formula. This *pretrained* Neural Network brings significant advantages to TINFETCH for improving adaptability and reducing the complexity burden during deployment. As depicted in Figure 4.1, the model architecture comprises ten input features, followed by two fully connected hidden layers consisting of 128 and 16 neurons respectively. A single output neuron, activated by a linear activation function, generates the predicted suffix. Both hidden layers utilize the rectified linear unit (ReLU) activation function and are initialized using a normal distribution. Additionally, the model is compiled using the mean absolute error as the loss function and the Adam optimizer. To train the model effectively, we combine real-world traces with synthetic datasets. These synthetic datasets are engineered to mimic various sequencing patterns and incorporate random noises, simulating challenging access patterns.

4.3.5 Hybrid Prefetching Decision

TINFETCH employs a read-ahead prefetcher in tandem with an ML-based approach, synergistically enhancing prefetching accuracy. The read-ahead prefetcher operates by proactively fetching the next block when the last access is contiguous to the currently accessed block. Conversely, the ML-based prefetcher addresses scenarios lacking an apparent sequential pattern. This combination ensures effective prefetching, catering to both sequential and non-sequential access patterns. By leveraging the historical accesses, the ML-based model prefixes and suffixes, it predicts future blocks likely to be accessed.

4.4 Implementation

We efficiently implement TINFETCH main modules within 200 LOC (Line of Code) in Python for simulation and C/C++ for low-level system integration. The model is trained on PyTorch

and the weights are quantized into integers for faster computation during inference. A dedicated prefetching simulator, comprising 500 LOC in Python, was developed to evaluate performance. Furthermore, our simulator incorporates four state-of-the-art prefetching solutions (Pythia, Leap, Delta LSTM, and SGDP) and three fundamental prefetching methods (Stride, Read-ahead, and Markov Chain), totaling 5000 LOC. This integration allows for a comprehensive analysis of diverse prefetching strategies within our simulation environment.

4.5 Experiment Setup

This section details how we setup the experiments to evaluate TINFETCH against the other prefetching solutions.

Workloads: We use FIO-generated traces and production traces from Microsoft, Tencent, and Alibaba as the workload. In total, billions of I/Os can be simulated from these traces. However, given our limited computing power, we only pick 50 random samples from these traces. Each sample trace contains 10k to 100k I/Os which is enough to help the model learn about the workload patterns within those time window. Furthermore, since the workload is always evolving and changing given different user and data usage scenarios, we believe that an ML-based prefetcher should focus on learning specific/short workload patterns. When the workload changes, we can update the model accordingly and enable model management to maximize the model reusability which is out of the scope of our current work. That being said, these randomly selected collections of traces represent different workload patterns that could happen at any given time. Testing various prefetching solutions on this workloads will ensure its capability to deal with different patterns similar to the situation in the real deployment scenario. We will make the FIO-generated traces publicly available in the future, while the Microsoft, Tencent, and Alibaba traces can be downloaded at the SNIA website.[25, 33, 40].

Machine Specification: The machine that we use has 125GB RAM and Intel i9-7980XE CPU @2.60GHz. It supports 2 threads per core and 18 cores per socket. The machine has 4 SSDs from

various manufacturers: Samsung SM951, WD Ultrastar SN200, Samsung PM1725A, and Intel DC P4600. For implementation, we use Python 3.8.15 with Keras v2.7.0 and Scikit-Learn v0.24.1.

Code Integration: We implement the basic prefetching algorithms (Stride, Read Ahead and Markov Chain) based on the description that we can find in a textbook and other papers [89, 120, 235]. Additionally, we acquire the source code of the state-of-the-art prefetchers (SGDP, Pythia, Leap, and Delta LSTM) from their GitHub repositories [34, 36, 39, 44]. We integrate these algorithms into our simulator and apply the same default setup for fairness.

Default values: We apply 50:50 for splitting the dataset into training and testing sets. The prefetchers that require this setup are SGDP, Pythia, and Delta LSTM. As a result, the metrics measurement for those algorithms will only be done on the 50% of the dataset. Furthermore, since TINFETCH's prefetch size is 128KB, we apply the same prefetch size to other prefetching methods unless it comes with a default mechanism to predict or adjust the prefetch size. The timeliness is implemented by simulating the fetching/seeking time delay from the HDD which is about 20 ms. When the prefetcher decides to fetch a data block, we will put that data block into a pending queue and apply a 20 ms delay before making it available in the cache. We believe that this delay is sufficient for storing the prefetched data in memory, especially considering that TINFETCH piggybacks the prefetch target onto the current I/O. This approach ensures that the HDD head only needs to move within 8MB of its current position to fulfill the prefetching request (as explained in §4.3.4) which can be done easily within 20 ms.

Unlimited memory: In real-world production environments, prefetching algorithms are typically combined with caching algorithms to manage the heap efficiently. However, for the purpose of evaluating the true performance of the prefetcher algorithm, all experiments are conducted with an unlimited cache size. This approach allows us to assess the prefetcher's ability to accurately predict future data without being influenced by the performance of different cache eviction policies. With this unlimited memory setup, the storage bandwidth load (§4.2.1) measurement is critical to evaluate the aggressiveness and the accuracy of different algorithms.

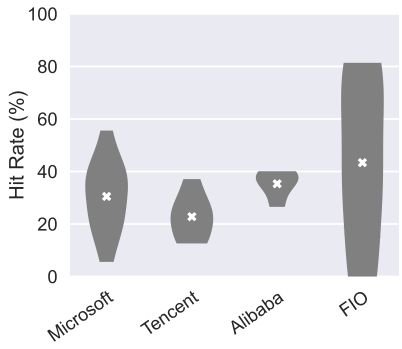


Figure 4.3: **Hit Rate across various traces.** TINFETCH *achieves average hit rate at around 30% with Microsoft and Tencent being the most challenging traces.*

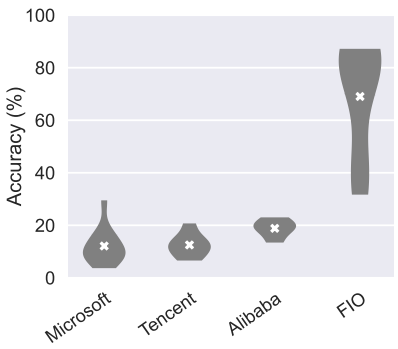


Figure 4.4: **Accuracy across various traces.** TINFETCH *gets 4.5x higher accuracy on FIO-generated traces compared to the Microsoft, Tencent, and Alibaba traces.*

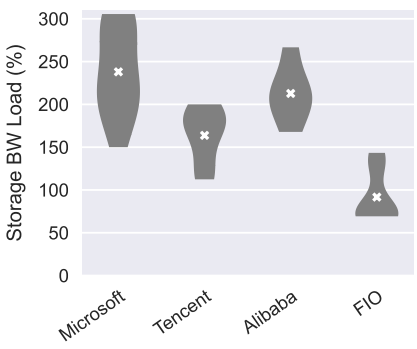


Figure 4.5: **Storage Bandwidth load.** *The FIO-generated traces being the most predictable workload compared to Microsoft, Tencent, and Alibaba traces.*

4.6 Evaluation

We start our TINFETCH evaluation by detailing its performance on various trace sources. Then, we compare its performance against the other prefetching techniques based on the hit rate, accuracy, and storage bandwidth load (or *bandwidth-load* for short). Finally, we analyze the performance and calculate the overall score.

4.6.1 TINFETCH *on Various Workloads*

Illustrated in **Figure 4.3, Figure 4.4, and Figure 4.5**, we want to evaluate TINFETCH's performance on each workload. In Figure 4.3, TINFETCH maintains a stable average hit rate of approximately 30%, with FIO-generated traces emerging as the most unpredictable workload which results in hit rate ranging from 1% to 80%. Moving to Figure 4.4, the accuracy across various traces is quite stable, especially at Microsoft, Tencent, and Alibaba averaging around 15%. Additionally, TINFETCH's exceptional performance can be seen from the accuracy on FIO-generated traces which is 4.5x higher than the other traces. Finally, Figure 4.5 reveals that while TINFETCH introduces a high bandwidth-load on most traces, it exhibits a very low bandwidth-load on the majority

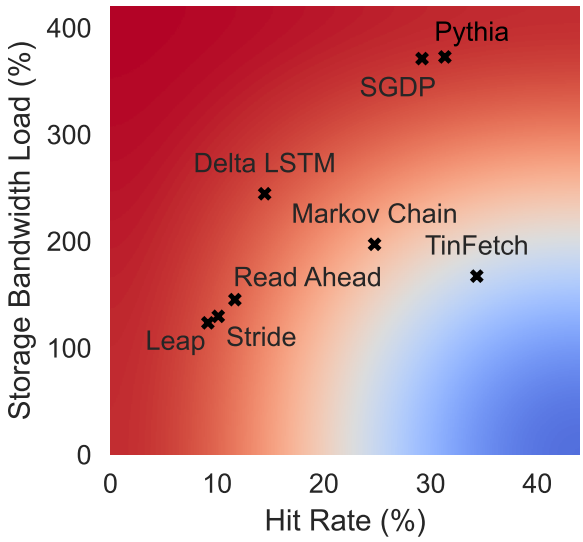


Figure 4.6: **Hit Rate vs Storage Bandwidth Load.** TINFETCH *performed best, having the highest high hit-rate with considerably low storage bandwidth load (bottom right))*

of FIO-generated traces. The bandwidth-load means that adding a prefetching system can alleviate significant load from the underlying storage devices which usually happens when the workload pattern is easily predictable which can be observed from the accuracy score that reach 80% as shown in the previous figure. Overall, this set of experiments shows that TINFETCH performs reasonably well across these diverse collections of workloads. Going forward, the experiments will be run on all of these workloads without showing detailed performance numbers on each trace. This will simplify our evaluation metrics when comparing TINFETCH vs other solutions.

4.6.2 Hit Rate vs Storage Bandwidth Load

In this experiment, we want to compare TINFETCH against other prefetching techniques by showing how each algorithm handle the tradeoff between performance gain and the bandwidth-load. As shown by **Figure 4.6**, the optimal performance of TINFETCH is evident as it is the closest one to the bottom-right quadrant marked by the blue area in the graph. In this region, TINFETCH demonstrates its superiority by achieving the highest hit rate while maintaining significantly low storage bandwidth-load. This outcome underscores TINFETCH's efficiency in prefetching, strik-

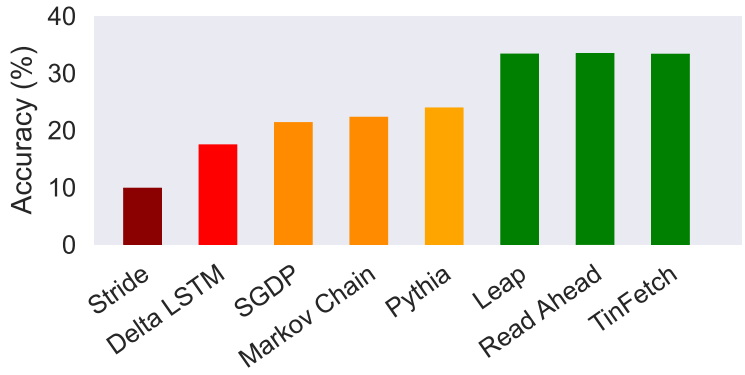


Figure 4.7: **Prefetching Accuracy.** TINFETCH *achieves high accuracy at around 33%, followed closely by Delta LSTM and Read Ahead prefetcher. A high accuracy emphasizes the precision of predictions made by the prefetching algorithm.*

ing a balance between a remarkable hit rate and minimal storage bandwidth-load. The second best algorithm is Markov Chain as it stays within the border between blue and red zones. Pythia and SGDP perform well on the hit rate metric, but they fail badly in the storage bandwidth-load category. Conversely, Leap and Stride exhibit simultaneous low bandwidth-load and minimal hit rates. This aligns with expectations for prefetchers struggling to discern any distinct LBA pattern within the trace. Consequently, this difficulty leads to a lower prefetch frequency and an overall diminished hit rate. In the next experiment, we will compare TINFETCH to other prefetchers based on different evaluation metrics.

4.6.3 Prefetching Accuracy Analysis

Accuracy is one of the most important metrics in the case of prefetching. It measures how precise the prediction made by the prefetching algorithm is. Based on **Figure 4.7**, TINFETCH achieves the highest accuracy rate followed by Read Ahead and Leap prefetcher as the second and third-best with all of them having nearly the same accuracy scores at around 33%. Next, there is Pythia Markov Chain, and SGDP which get around 25% accuracy. Finally, Delta LSTM and Stride sit in the last two positions where Stride gets the lowest score at 10% which is also reflected in the previous figure. Nevertheless, a closer examination of Read Ahead and Leap reveals that rely-

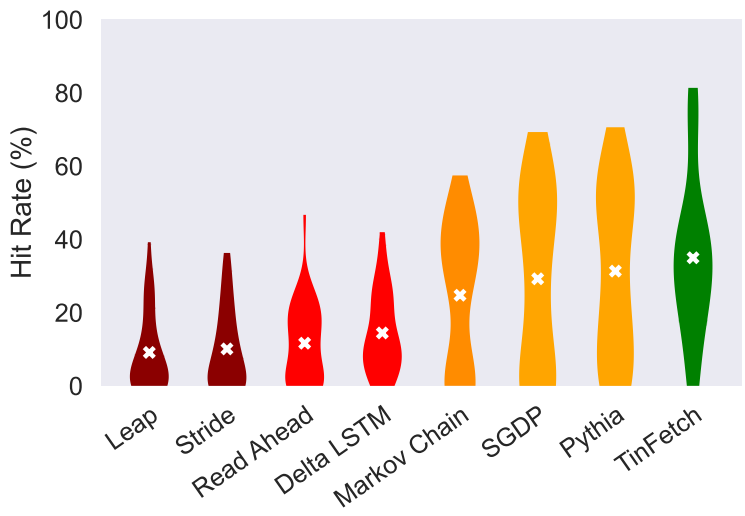


Figure 4.8: **Hit rate distribution.** TINFETCH *achieves highest hit rate value and performs consistently around the median result.*

ing solely on the accuracy data might not convey the complete narrative. As depicted in Figure 4.6 these two prefetchers find themselves in the red zones, indicating a challenge in achieving an optimal balance between maximizing performance gain and minimizing storage bandwidth load. Intriguingly, Markov Chain presents a positive outcome in Figure 4.6, despite having a moderate accuracy score. This observation implies that a lower accuracy score does not always correlate with a reduced hit rate. Such instances occur when a prefetcher, despite its lower accuracy, successfully retrieves popular data blocks in a timely manner, consequently boosting its hit rate. This underscores the nuanced nature of prefetching dynamics. Lastly, since TINFETCH consistently dominates the metrics in accuracy, hit rate, and bandwidth-load as shown in the last two figures, it reinforces TINFETCH's prowess and reliability in achieving superior performance in diverse aspects of prefetching.

4.6.4 TINFETCH's Performance Score

Examining **Figure 4.8** reveals that TINFETCH boasts the highest average hit rate, closely trailed by Pythia and Markov Chain. relying solely on this data doesn't provide a comprehensive under-

Algorithm	Hit (%)	BW-load (%)	Perf. Score
TINFETCH	34.3	167.2	**0.205**
Markov Chain	24.7	197.0	0.125
Pythia	31.3	372.7	0.084
Read Ahead	11.6	145.2	0.080
SGDP	29.2	371.1	0.078
Stride	10.1	129.6	0.077
Leap	9.1	123.5	0.074
Delta LSTM	14.4	244.4	0.058

Table 4.1: **Performance Score.** TINFETCH *acquires the best performance score.*

standing of algorithm performance. A closer look at **Table 4.1** unveils the storage bandwidth load (SBL) for each algorithm. Notably, despite Pythia securing the second-highest hit rate, its storage bandwidth load score is the worst among all algorithms. This discrepancy suggests that a high hit rate can lead to excessive storage bandwidth load, particularly when the prefetcher aggressively fetches the data blocks which causes cache pollution.

$$\texttt{Performance Score} = \frac{\texttt{Hit Rate}}{\texttt{Storage BW Load}} \tag{4.1}$$

To fully evaluate these multi-dimensional metrics, we formulated a scoring method named *Performance Score*, outlined in **Equation (4.1)**. As detailed in Table 4.1, TINFETCH attains the highest performance score of 0.2, surpassing the best state-of-the-art learning-based and heuristic-based prefetchers by 2.4x and 1.6x respectively. This comprehensive evaluation underscores TINFETCH's outstanding performance across various dimensions.

4.7 Conclusion

We have introduced TINFETCH, a novel data prefetching method that combines heuristic and learning-based approaches. It not only achieves a high hit rate but also maintains low storage bandwidth load, enabling its practical use in high-performance data centers. TINFETCH also opens doors for future work, such as exploring combinations of traditional algorithms with various machine learning models. In addition, there are many components inside TINFETCH that can be further enhanced, such as improving the stream-disentanglement algorithm and compressing the neural network model.

CHAPTER 5

OTHER WORKS

Here we briefly describe some of our other storage-related works, including PBSE (Section 5.1), FlyMC (Section 5.2), E2E (Section 5.3), and LibrOS (Section 5.4).

5.1 PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel

Data-parallel frameworks have become a necessity good in large-scale computing. To finish jobs on time, such parallel frameworks must address the "*tail latency*" problem. One popular solution is *speculative execution (SE)*; with SE, if a task runs slower than other tasks in the same job (a "straggler"), the straggling task will be speculated (via a "backup task"). With a rich literature of SE algorithms, existing SE implementations such as in Hadoop and Spark are considered quite robust. Hadoop's SE for example has architecturally remained the same for the last six years, based on the LATE algorithm [361]; it can effectively handle stragglers caused by common sources of tail latencies such as resource contentions and heterogeneous resources.

However, we found an important source of tail latencies that current SE implementations cannot handle graciously: *node-level network throughput degradation* (*e.g.*, a 1Gbps NIC bandwidth drops to tens of Mbps or even below 1 Mbps). Such a fault model can be caused by a severe network contention (*e.g.*, from VM over-packing) or due to degraded network devices (*e.g.*, NICs and network switches whose bandwidth drops by orders of magnitude to Mbps or Kbps level in production systems. The uniqueness of this fault model is that only the network device performs poorly, but other hardware components such as CPU and storage are working normally; it is significantly different than typical fault models for heterogeneous resources or CPU/storage contentions.

To understand the impact of this fault model, we tested Hadoop [4] as well as other systems

including Spark [360], and Flume [3], and Apache S4 [5], on a cluster of machines with one slow-NIC node. We discovered that many tasks transfer data through the slow-NIC node but *cannot escape* from it, resulting in long tail latencies. Our analysis then uncovered many surprising *loopholes* in existing SE implementations, which we bucket into two categories: the *"no" straggler* problem, where *all* tasks of a job involve the slow-NIC node (since all tasks are slow, there is "no" straggler detected) and the *straggling backup* problem, where the backup task involves the slow-NIC node again (hence, *both* the original and backup tasks are straggling). Overall, we found that a network-degraded node is worse than a dead node; it leads to a worse performance degradation.

Given the maturity of SE implementations (*e.g.*, Hadoop is 10 years old), we investigate further the underlying design issues that lead to the loopholes. We believe there are two root causes. First, node-level network degradation (without CPU/storage contentions) is not considered a fault-model. Yet, this real fault model affects data-transfer *paths*, not just tasks per se. This leads us to the second root cause: existing SE approaches only report task progress but do not expose *path* progress. Sub-task progress*es* such as data transfer rates are simply lumped into *one* progress score, hiding slow paths from being detected by the SE algorithm.

In PBSE [304], we present a robust solution to the problem, *path-based speculative execution*, which contains three important ingredients: path progress, path diversity, and path-straggler detection and speculation.

First, in PBSE, *path progresses* are exposed by tasks to their job/application manager (AM). More specifically, tasks report the data transfer progresses of their Input→Map, Map→Reduce (shuffling), and Reduce→Output paths, allowing the AM to detect straggling paths. Unlike the simplified task progress score, our approach does *not* hide the complex dataflows and their progresses, which are needed for a more accurate SE. Paths were not typically exposed due to limited transparency between the computing (*e.g.*, Hadoop) and storage (*e.g.*, HDFS) layers; previously, only data locality is exposed. PBSE advocates the need for a more cross-layer collaboration, in a non-intrusive way.

Second, before straggling paths can be observed, PBSE must enforce *path diversity*. Let's consider an initial job (data-transfer) topology X→B and X→C, where node X can potentially experiences a degraded NIC. In such a topology, X is a *single point of tail-latency failure* (*"tail-SPOF"* for short). Path diversity ensures no potential tail-SPOF will happen in the initial job topology. Each MapReduce stage will now involve multiple distinct paths, enough for revealing straggling paths.

Finally, we develop *path-straggler detection and speculation*, which can accurately pinpoint slow-NIC nodes and create speculative backup tasks that avoid the problematic nodes. When a path A→B is straggling, the culprit can be A or B. For a more effective speculation, our techniques employ the concept of failure groups and a combination of heuristics such as greedy, deduction, and dynamic-retry approaches, which we uniquely personalize for MapReduce stages.

We have implemented PBSE in the Hadoop/HDFS stack in 6003 LOC (which we will release publicly). PBSE runs side by side with the base SE; it does not replace but rather enhances the current SE algorithm. Beyond Hadoop/HDFS, we also show that other data-parallel frameworks suffer from the same problem. To show PBSE generality, we also have successfully performed initial integrations to Hadoop/QFS stack [265], Spark [360], and Flume [3].

We performed an extensive evaluation of PBSE with a variety of NIC bandwidth degradations (60 to 0.1 Mbps), real-world production traces (Facebook and Cloudera), cluster sizes (15 to 60 nodes), scheduling policies (Capacity, FIFO, and Fair), and other tail-tolerance strategies (aggressive SE, cloning, and hedge read). Overall, we show that under our fault model, PBSE is superior to other strategies (between 2–70×speed-ups above the 90^{th}-percentile under various severities of NIC degradation), which is possible because PBSE is able to escape from the network-degraded nodes (as it fixes the limitation of existing SE strategies).

PBSE is a major continuation of our prior work in the "limplock" paper [105]. In this PBSE paper, we deliver many new contributions. First and most significantly, we introduce PBSE as a robust solution to the aforementioned problem, while the limplock paper only highlights the

problem. Second, in terms of Hadoop, our prior work only used four simple microbenchmarks, while for PBSE, we ran over 850 of hours of real production workloads. As a result, we uncovered more loopholes than the three limplock topologies we reported before [105]). The more uncovered loopholes allowed us to dissect the root causes such as the hidden path progresses. Finally, we have collected more stories of degraded NIC from datacenter engineers and operators, who also highlight that monitoring tools are not a sufficient solution to address the problem.

5.2 FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems

Datacenter systems such as distributed key-value stores, scalable file systems, data-parallel computing frameworks, and distributed synchronization services, are the backbone engines of modern clouds, but their complexities and intricacies make them hard to get right. Among all types of issues in such systems, complex interleavings of messages, crashes, and reboots are among the most troublesome [119, 145, 226, 227, 330]. Such a non-deterministic order of events across multiple nodes cause "*distributed concurrency*" bugs to surface (or "**DC** bugs" for short). Developers deal with DC issues on a monthly basis [130, 136], or worse on a weekly basis for newly developed protocols [7]. They are hard to reproduce and diagnose (take weeks to months to fix the majority) and lead to harmful consequences such as whole-cluster unavailability, data loss/inconsistency, and failed operations [203].

Ideally, bugs should be unearthed in testing, not in deployment [101]. One systematic testing technique that fits the bill is stateless/software model checking that runs directly on implementation-level distributed systems [134, 137, 174, 202, 296, 347, 353]. These software model *checkers*[1] attempt to exercise many possible interleavings of non-deterministic events such as messages and fault timings, hereby pushing the target system into unexplored states and potentially revealing

1. In this paper, **"checkers"** specifically represent the distributed system software model checkers, as cited above, *not* including "local" thread-scheduling checkers [14, 251] or the classical model checkers [113, 129].

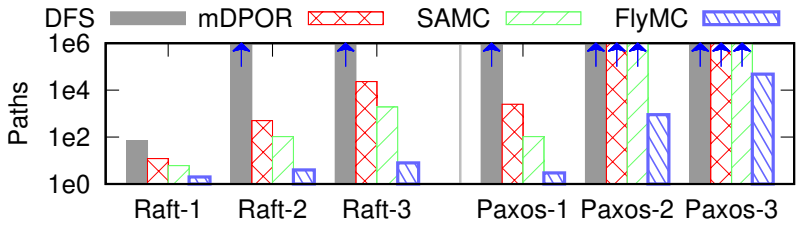


Figure 5.1: **Checkers[1] scalability.** *The x-axis represents the tested protocol (Raft or Paxos) with 1 to 3 concurrent updates. The log-scaled y-axis represents the number of paths to exhaust the search space (i.e., the path explosion). Compared to our checker,* FLYMC*, current checkers do not scale well under more complex workloads. "↑" indicates incomplete path exploration.*

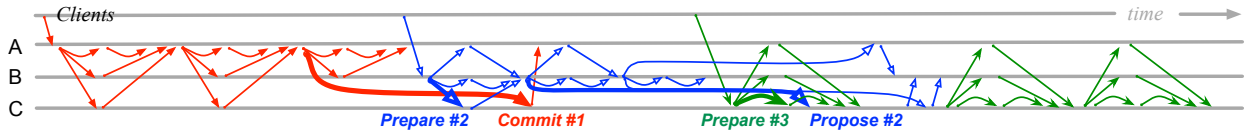


Figure 5.2: **A complex DC bug in Cassandra Paxos (`CASS-1`).** *This bug which we label as "`CASS-1`" [6] requires three Paxos updates and only surfaces with the two flips (the prepare message with ballot 2 must be enabled before the commit with ballot 1 and the prepare with ballot 3 before the propose with ballot 2) happening within all the possible flips of the 54 events, resulting in data inconsistency.*

hard-to-find bugs.

One nemesis of checkers is the *path explosion problem*. As an illustration, suppose there are 10 concurrent messages (*events*) $\{a, b, .., j\}$, a naive checker such as depth-first search (DFS) has to exercise 10! (factorial) unique execution paths ($ab..ij$, $ab..ji$, and so on). Figure 5.1 illustrates further this explosion problem. The gray "DFS" bar shows almost 100 paths to explore (in y-axis) under a *simple* workload such as an instantiation of a Raft update protocol ("`Raft-1`") [263].

To tame this problem, checkers employ path *reduction algorithms*. For example, MODIST [353] and some others [296, 347] adapted the popular concept of Dynamic Partial Order Reduction (DPOR) [117, 128], for example "a message to be processed by a given node is *independent* of other concurrent messages destined to other nodes [hence, need *not* to be interleaved]." SAMC [202] also extended DPOR further. As a result, reductions significantly improve upon a naive DFS method, as shown by the "mDPOR" and "SAMC" bars on `Raft-1` in Figure 5.1.

Despite these early successes, we found that the path explosion problem remains untamed

under more *complex* workloads. For example, under two or three concurrent Raft updates (`Raft-2` and `-3` workloads in Figure 5.1), the number of paths to explore still increases significantly in MoDist and SAMC. Not to mention a much more complex workload such as Paxos [195] where the path explosion is larger (*e.g.*, `Paxos-1` to `-3` workloads in Figure 5.1).

To sum up, existing checkers fail to scale under more complex distributed workloads. Yet in reality, some real-world bugs are still hidden behind complex interleavings. For example, the Paxos bug in Cassandra in Figure 5.2 can only surface under a workload with three concurrent updates with 54 events in total. These kinds of bugs will take weeks to surface with existing checkers, wasting testing compute resources and delaying bug finding and fixes. For all the reasons above, to find DC bugs, some checkers mix their algorithms with *random* walks [353] or *manual* checkpoints [137], hoping to faster reach "interesting" interleavings that would lead to DC bugs. However, this approach becomes unsystematic – the random and manual approaches lead to poorer coverage than a systematic coverage of all states relevant to observable events.

We present FlyMC, a fast, scalable, and systematic software/stateless model checker that covers all states relevant to observable events for testing distributed systems implementations. FlyMC achieves scalability by leveraging the internal properties of distributed systems as we illustrate below with three FlyMC's algorithms.

(1) Communication and state symmetry: Common in cloud systems, many nodes have the same role (*e.g.*, follower nodes, data nodes). The state transitions of such symmetrical nodes usually depend solely on the order and content of messages, irrespective of the node IDs/addresses. Thus, FlyMC reduces different paths that represent the same symmetrical communication or state transition into a single path.

(2) Event independence: While state symmetry significantly omits symmetrical paths, many events must still be permuted within the non-symmetrical paths. FlyMC is able to identify a large number of event independencies that can be leveraged to alleviate a wasteful reordering. For example, FlyMC automatically marks concurrent messages that update disjoint sets of variables

as independent. FLYMC can also find independence among crash-related events.

(3) Parallel flips: While the prior methods reduce message interleavings to every node, in aggregate many flips (reordering of events) must still be done across all the nodes. The problem is that in existing checkers, only one pair of events is flipped (reordered) at a time. To speed this up, parallel flips perform simultaneous reorderings of concurrent messages across different nodes to quickly reach hard-to-reach corner cases.

Finally, not only path reduction but wall-clock speed also matters. Existing checkers must wait a non-negligible amount of time in between every pair of enabled events for some correctness and functionality purposes. The wait time is reasonable under simple workloads, but it significantly hurts the aggregate testing time of complex workloads. FLYMC optimizes this design with local ordering enforcement and state transition caching.

Collectively, the algorithms make FLYMC on average 16× (up to 78×) faster than other state-of-the-art systematic and random-based approaches, and the design optimizations improve it to 28× (up to 158×). FLYMC is integrated with 8 widely-used systems, the largest number of integration that we are aware of. We model checked 10 protocol implementations (Paxos, Raft, etc.), successfully reproduced 12 old bugs, and found 10 new DC bugs, all confirmed by the developers and all were done in a systematic way *without* random walks or manual checkpoints. Some of these bugs cannot be reached by prior checkers within a reasonable time. We have released our FLYMC publicly [13].

5.3 E2E: Embracing User Heterogeneity to Improve Quality of Experience on the Web

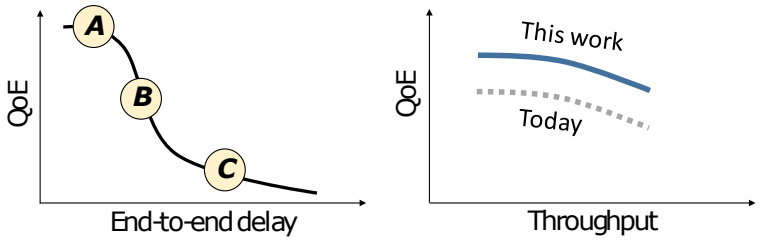
Improving end-to-end performance is critical for web service providers such as Microsoft, Amazon, and Facebook, whose revenues depend crucially on high quality of experience (QoE). More than ten years have passed since Amazon famously reported every 100ms of latency cost them 1% in sales, and Google found 0.5s additional load time for search results led to 20% drop in

traffic [24]. Today, latency remains critical but the consequences have gotten steeper: an Akamai study in 2017 showed every 100ms delay in website load time hurt conversion rates by 7% [29], and Google reported higher mobile webpage load times more than doubling the probability of a bounce [31]. Naturally, web service providers strive to cut server-side delays—the only delays they can control—to improve the end-to-end performance of each web request. Following this conventional wisdom, a rich literature has developed around reducing web service delays (*e.g.* [86, 141, 154, 188, 307, 312, 339]).

Our work is driven by a simple observation: although reducing server-side delay generally improves QoE, the exact amount of QoE improvement varies greatly depending on the *external delay* of each web request, *i.e.*the total delay caused by ISP routing, last-mile connectivity, and so forth. In other words, if we define *QoE sensitivity* as the amount QoE would improve if the server-side delay were reduced to zero, there is substantial heterogeneity in QoE sensitivity across users. This heterogeneity results from two empirical findings. First, as illustrated in Figure 5.3(a), QoE typically decreases along a sigmoid-like curve as delay increases. When the external delay is very short or very long (*e.g.A* or *C* on the curve), QoE tends to be less sensitive to the server-side delay than when the external delay is in the middle (*e.g.B* on the curve). We verified this trend using traces from Microsoft, a cloud-scale production web framework, as well as a user study we ran on Amazon MTurk to derive QoE curves for several popular websites.

Second, external delays are inherently diverse across user requests due to factors that are beyond the control of web service providers: *e.g.*ISP routing, last-mile connectivity, DNS lookups, and client-side (browser) rendering and processing. Our analysis of Microsoft traces reveals substantial variability in external delays even among requests received by the same frontend web server, for the same web content.

The heterogeneity in QoE sensitivity implies that following the conventional wisdom of minimizing server-side delays uniformly across all requests can be inefficient, because resources may be used to optimize requests that are not sensitive to this delay. Instead, we should devote these



(a) User heterogeneity (b) Better QoE & throughput

Figure 5.3: (a) An illustrative example of three requests with different QoE sensitivities to server-side delays, and (b) the potential QoE/throughput improvement if we leverage user heterogeneity.

resources to requests whose QoE *is* sensitive to server-side delay.

At a high level, user heterogeneity is inherent to the Internet's loosely federated architecture, where different systems are connected together functionally (client devices, ISPs, cloud providers, etc.), but delay optimization is handled separately by each system. Our work does not advocate against this federated architecture; on the contrary, we argue that web service providers should *embrace the heterogeneity of QoE sensitivity across users* to better allocate server-side resources to optimize user QoE. Using traces from Microsoft, we show that by reshuffling server-side delays among concurrent requests so that requests with more sensitive QoE get lower server-side delays, we could increase average duration of user engagement (a measure of QoE) by 28% .

To fully explore the opportunities of leveraging user heterogeneity, we present *E2E* [367], a resource allocation system for web services that optimizes user QoE by allocating resources based on each user's sensitivity to server-side delay. E2E can be used by any shared-resource service; for example it can be used for replica selection in a distributed database such that sensitive requests are routed to more lightly-loaded replicas.

The key conceptual challenge behind E2E is that, unlike static properties of a request (*e.g.*basic vs. premium subscription, or wireless vs. wired connectivity), one cannot determine the QoE sensitivity of a request upon arrival based solely on its external delay. Instead, QoE sensitivity depends on the server-side delay as well. The same server-side delay might cause more QoE degradation to a seemingly less sensitive request (*A*) than to a seemingly more sensitive request (*B*), depending on the magnitude of the delay. Thus, one cannot prioritize the allocation of resources without tak-

ing into account the external delay distribution as well as the server-side delay distribution, which in turn is affected by the by the resource allocation itself. This makes the problem circular and computationally expensive to solve at the timescale of a web serving system.

E2E addresses this challenge from *both* the algorithmic perspective and the systems perspective. From the algorithmic perspective, E2E decouples the resource allocation problem into two subproblems, each of which can be solved efficiently: (1) a workload allocation process, which determines the server-side delay distribution without considering QoE sensitivity; and (2) a delay assignment process, which uses graph matching to "assign" the server-side delays to individual requests in proportion to their QoE sensitivity. E2E solves the two subproblems iteratively until it finds the best workload allocation and delay assignment.

From the systems perspective, E2E further reduces the decision-making cost of each request by coarsening the timescale and the granularity of resource allocation decisions. Observing that the optimal allocations are insensitive to small perturbations in the distributions of external delays and server-side delays, we allow the system to cache the allocation decisions in a lookup table that can be disseminated as needed, and only update it when a significant change is detected in either of the delay distributions.

We demonstrate the practical benefit of E2E by integrating it into two open-source systems to make them QoE-aware: replica selection in a distributed database (Cassandra) and message scheduling in a message broker (RabbitMQ). We use a trace-driven evaluation based on Microsoft traces and our testbed deployments to show that (1) E2E can improve QoE (duration of user engagement) by 28% or serve 40% more concurrent requests without any drop in QoE; and (2) E2E incurs negligible (4.2%) system overhead and less than 100μs delay.

5.4 Layered Contention Mitigation for Cloud Storage

In the storage world, many varieties of products including hyperconverged storage, key-value stores and databases, are advertised not only with traditional metrics such as throughput and av-

erage latency, but tail latency as well (*e.g.*Xms latency guaranteed at the Y^{th} percentile) [1, 16, 17, 30]. For many storage deployments, especially in the cloud, resource sharing is a de-facto configuration; many different users share the same storage, different applications run on the same machine, and foreground and background tasks can run at the same time. While reports say unpredictable latency can be caused by many factors [212], in this context of sharing, the dominant factor of unpredictable latency is resource contention.

The major challenge is that contention appears in many different resource layers, all directly impacting software systems that use multiple resources, including storage systems. Let's take distributed cloud storage such as key-value stores as an example. They require CPUs to process user requests, memory to cache the data, lock resources to implement concurrent data sharing correctly, and storage devices to fairly and promptly serve their I/Os. However, CPUs might not be instantly available due to process/VM contention, load imbalance or task rebalancing across cores [102, 143, 165, 207, 264, 270, 345]; memory access can be halted by the language runtime for heap reorganization [73, 125, 126, 233, 256, 306]; foreground locks might be used by background management operations such as compaction, flushing and migration [60, 186, 204, 274]; and I/Os can be delayed under bursty workloads [68, 169, 216, 245, 275]. Another challenge on top of all of these is that unlike compute jobs that run for seconds, the small latency tail that storage users expect is at the millisecond granularity (*e.g.*5ms at the 99^{th}-percentile latency).

Guaranteeing highly stable latencies in multi-resource systems including distributed storage is still an open-ended challenge. In this context, we studied and reviewed popular contention mitigation scenarios, from application modification, speculation, replica selection and resource-level optimization, and evaluate them on multiple dimensions such as simplicity, efficiency, reactivity and coverage. We found that while each of these methods has advantages in multiple dimensions, they have inherent limitations that cannot be fixed within its own category.

Our finding poses the following question: Is there a strategy that can combine the best of all the worlds, *e.g.*a strategy that can keep the simplicity of speculation, the efficiency of application

modification, the reactivity of resource optimization, and the coverage of replica selection? We found that this question is fundamentally hard to answer when existing methods attempt to solve the problem either entirely in the applications or in the individual resource managers (*e.g.*in OS, runtime or library).

We introduce LIBROS [319], an ecosystem for contention mitigation with supports from library, runtime, and operating system layers. The principle behind our ecosystem is that while distributed applications are responsible for the retry mechanisms (as they know where data replicas are), resource managers should help notify applications when resources are highly contended. In this "app-OS" co-design, neither the application nor the resource managers attempt to solve the problem entirely by itself.

The key ingredient in LIBROS is enabling a uniform type of support that can be adopted across multiple resource layers. For this, LIBROS first introduces an end-to-end "`request`" abstraction that flows through multiple resource layers. The concept of `request`, including its corresponding deadline and cancellability, becomes a first-class citizen. In mitigating tail latency, resource managers now operate on `request` level as opposed to opaque stream of bytes. Around this abstraction, resource managers can build a stackable support, namely request cancellation and delay prediction, for the individual resources that they manage.

To adhere on simplicity, our approach does not modify resource-level policies such as scheduling and allocation decisions. Instead, given a particular QoS policy that a resource manager employs, we build a delay predictor that estimates how long an incoming request will be delayed in that layer. Hence, all the resource layers in the LIBROS ecosystem can provide a new, uniform capability: when a request arrives at a resource layer, the resource manager predicts the delay in that layer, and if the delay violates the request deadline, the layer will cancel this (cancellable) request, knowing that the application has another replica to go to.

To achieve fast reactivity, resource managers send cancellation notifications that inform the application to quickly react to the delay by sending speculative retries to other replicas. At the same

time, efficiency (no extra load) is achieved because the original request has been automatically cancelled by the contended resource layer. Finally, for coverage, we build the capability above for three major often-contended resources. Specifically, we present ETOS, an operating system with CPU contention prediction, ETR, a Java runtime that notifies when requests will be stalled due to heap garbage collection, and ETLIB, a library that throws an exception when a lock cannot be required within the deadline. ("ET" implies end-to-end tail mitigation support.)

In building LIBROS, the main challenge is creating two new capabilities in resource layers: prompt cancellation notification and accurate delay prediction. In our implementation, these capabilities are written in 2300, 1000 and 250 LOC in ETOS, ETR and ETLIB, respectively (will be open sourced). To make applications benefit from these capabilities, LIBROS exposes simple APIs, *e.g.*, we modified Cassandra and MongoDB only in 120 and 50 lines, respectively. Our evaluation shows that LIBROS is faster by 5-70% starting at 90P (the 90^{th}percentile) than popular practices such as speculative execution and is only 3% slower on average compared to the "best" (no contention) scenario.

CHAPTER 6

FUTURE WORKS

In this section, we outline several promising avenues for future research and development in the field of storage systems. These areas represent diverse yet interconnected aspects of leveraging machine learning (ML) techniques to enhance the efficiency, scalability, and adaptability of storage infrastructure. As storage systems continue to evolve in complexity and diversity, the need for more adaptable systems becomes increasingly imperative. Utilizing machine learning enables systems to learn abstract patterns that may elude traditional heuristic-based algorithms, thereby unlocking new levels of optimization and performance enhancement.

6.1 Groupability-aware Computing

Groupability-aware computing introduces a novel paradigm in data processing by leveraging the inherent grouping structures present in data access to enhance computational efficiency for an all-or-nothing scenario. In EVSTORE, we have shown that groupability pattern can be mined efficiently with negligible overhead while being integrated into various caching algorithms. This fundamental principle of groupability can be generally applied to develop a groupability-aware computing, emphasizing its ability to exploit group-based patterns [84], correlations, and dependencies to optimize algorithmic performance across diverse domains such as machine learning, data mining, and data access optimization in various topics such as graph analytics [354], graph learning [131], protein discovery [354], big data framework [185], memory offloading system [327], ML/AI serving systems [359], microservices [231], large language model serving [190], serverless computing [108], semi-microkernel file system [228], and disagregated memory system [200].

6.2 Hybrid Caching Algorithm

Heuristic-based caching systems often rely on predefined rules or policies, which may not adapt well to dynamic and evolving workloads. On the other hand, ML-based approaches have shown promise in learning complex patterns and making predictions based on historical data. By combining the strengths of both approaches, hybrid caching systems aim to achieve improved adaptability, efficiency, and performance optimization. ML algorithms can leverage large datasets to identify patterns and trends in workload behavior, while heuristic algorithms can provide stability and reliability in decision-making processes. This integration allows caching systems to dynamically adjust their caching policies based on real-time workload characteristics, leading to enhanced cache hit rates, reduced latency, and improved overall system performance. For example, our EVSTORE can be improved to continuously learn about the workload and fine tune the caching configuration dynamically based on the workload pattern. Moreover, we can also build a general ML agent that can learn the behaviour of any given heuristic based caching algorithm such as FIFO [352]. Then, the agent will dynamically fine-tune any algorithm independently by relying on feedback mechanism from the environment similar to the Q-learning strategy in Reinforcement Learning. Given the popularity and the scale of in-memory database to store ML-related data structure, we can combine the hybrid caching algorithm with efficient concurrency protocol [241] and novel indexing method [362] to improve the asynchronous update performance and its consistency.

6.3 Machine Learning Framework for ML-for-Storage

The ML Framework for ML-for-Storage represents a structured approach to implementing machine learning techniques in storage systems to enhance their efficiency, reliability, and performance. By extensively exploring various stages in ML pipeline, such as data preprocessing, feature engineering, model training, and evaluation, this framework enables the development of intelligent storage solutions tailored to specific use cases and workloads. Through the integration of ma-

chine learning algorithms, such as classification, regression, clustering, and anomaly detection, into storage management tasks, the framework aims to optimize resource allocation, improve data access patterns, and mitigate performance bottlenecks. The DS framework in our HEIMDALL can be extended to explore various ML-for-Storage topics such as scheduling [155, 292, 323], data placement [244, 287, 301], automatic hardware optimization [363], model management [295], and autonomous database [322].

6.4 ML-based I/O admission/placement for Zoned Namespace SSD

Traditional admission control mechanisms often struggle to adapt to the unique characteristics of Zoned Namespace SSDs, which organize storage into zones with different performance characteristics. ML techniques offer a promising solution by leveraging historical data and workload patterns to make intelligent decisions about which I/O requests to admit to the SSD. By analyzing various factors such as access patterns, data locality, and resource availability, ML models can dynamically adjust admission policies to optimize performance and resource utilization. This approach enables Zoned Namespace SSDs to better handle diverse and dynamic workloads, leading to improved throughput, reduced latency, and enhanced overall system efficiency [244]. For example, ML-based I/O admission similar to our HEIMDALL can learn about the workload behavior and adjust the admission strategy dynamically to minimize the tail latency [103, 153, 211, 303].

6.5 Data-stall aware DNN Training/Scheduling

Incorporating machine learning to develop a data-stall aware DNN scheduler is motivated by the growing complexity and scale of DNN training workloads [155, 355], particularly in multi-tenant cluster environments [246]. Traditional heuristic-based methods often struggle to effectively schedule tasks in such environments due to the intricate and interleaved data access patterns inherent in DNN training. We can extend TINFETCH to learn the pattern of the data usage and

memory access [292] during the DNN training and perform prefetching to mitigate the data stall during DNN training. Since the scale of distributed DNN training keeps growing (e.g., involving tens of thousands of GPUs) and the length of training time (in months), the solution to tackle this challenge must also consider the data-stall induced by node failure [323], the data placement within the disaggregated storage [46], and the gradient synchronization bottleneck that accounts for more than 60% of the total training time [58].

6.6 ML-powered Tiered Memory/Storage Systems

Tiered storage and memory systems play a crucial role in modern computing environments by optimizing data access and storage efficiency. These systems leverage a hierarchy of storage tiers, ranging from high-speed [180, 236], expensive memory to slower, more cost-effective storage options [54], to accommodate varying performance and capacity requirements. By dynamically migrating data between tiers locally or through network [74] based on access patterns and usage frequency, tiered storage systems aim to minimize latency and maximize throughput while minimizing costs. For example, in EVSTORE, we exploited groupability pattern and embedding compression methods to optimize our three layers caching system. In this case, similar caching methods can be implemented on multiple memory tiers instead of relying on the main memory (DRAM) [109, 276]. Moreover, instead of prefetching the data to a the main memory, we can build similar systems like TINFETCH to prefetch data into multiple tiers based on their degree of importance [208]. Furthermore, our HEIMDALL and EVSTORE can further be tuned to predict the access latency on multi-tiers storage systems to better handle the I/O rerouting [147, 294], load balancing [307, 309], data placement [236], and memory management [48, 118, 162, 185, 292, 355].

6.7 ML-based Memory Harvesting VMs

By employing predictive models and adaptive algorithms, we can identify underutilized memory resources, predict future memory demands, and efficiently reallocate memory to VMs with higher priority or increased workload demands. This approach is very effective in lowering the data center operation cost considering that the price of AWS, Azure, and GCP's Spots instances are cheaper than the regular server node. This will also over a better resource utilization and an enhanced memory performance to meet the evolving needs of modern cloud workloads, especially during the peak loads. Our caching algorithm in EVSTORE and prefetching algorithm in TINFETCH can be extended to better learn about the VMs' behavior to continuously analyze VM memory usage patterns, application behavior, and workload dynamics to intelligently reclaim and redistribute memory resources [121]. Furthermore, this solution can also be applied to RDMA-supported VMs [248] by dynamically allocating/polling from the available RDMA to increase its utilization. This remote memory [291] can be utilized to help serve giant ML models (e.g., GPT-3 requires 325 GB) during peak loads [219].

6.8 Continuously Learning Storage Systems

Continuously learning (CL) storage systems represent a novel approach in the field of storage management, characterized by their remarkable ability to adapt and optimize storage resources dynamically in response to evolving workload patterns and user demands. Our ML-based solution for I/O admission (HEIMDALL) and prefetching (TINFETCH) can further be improved to continuously learn new workload pattern using Dynamic Neural Network [93] or Mixture-of-Expert (MoE) which needs an efficient model management strategy for large-scale deployment. The continuously learning pardigm can applied to dynamically adjust storage configurations such as data placement strategies [118, 236, 322] and scheduling [155, 355]. With the growing demand of adaptive deep learning parallelism, this CL solution can be integrated into the training framework

[194, 377], model serving systems [255, 356], embedding vector DB [344], etc., to ensure that the system remains responsive and efficient, even in the face of fluctuating workloads, diverse storage backends, and various software level objective.

6.9 ML-based Software-Defined Storage

Software-Defined Storage (SDS) represents a transformative paradigm in storage infrastructure, where storage management and provisioning are decoupled from physical hardware and abstracted into software layers. This abstract outlines the core principles, architecture, and benefits of SDS, emphasizing its flexibility, scalability, and cost-effectiveness. By virtualizing storage resources and automating data management tasks, SDS enables dynamic allocation of storage resources based on application demands, simplifies storage administration, and facilitates integration with cloud and hybrid environments. The abstract explores various SDS features, including policy-driven automation, centralized management interfaces, and support for diverse storage technologies, highlighting its role in modernizing storage infrastructure and enabling agile, software-defined data centers [322]. With this explosion of tunable and learnable features, ML can be utilized to better exploit the pattern that is invisible to normal heuristic-based methods. Our HEIMDALL and TINFETCH can be improved and integrated into a higher-level ML-based agent that controls the storage system stack [111, 147, 272, 277, 278].

CHAPTER 7

CONCLUSION

This dissertation addresses the pressing need to enhance storage supports for deep recommendation systems in the face of escalating demands for low-latency, scalability, and cost-efficient deployment. The exponential growth of deep recommendation system usage underscores the importance of optimizing storage solutions to ensure seamless user experiences. We embarked on a comprehensive exploration, developing solutions at various layers of the storage stack to tackle the challenges posed by large embedding vector tables (EV tables) and unpredictable SSD latency.

The first contribution, EVSTORE, offers a caching system designed to exploit groupability patterns, enabling scalable and cost-efficient deployment of deep recommendation systems. By integrating EVSTORE into existing infrastructure, we achieved significant reductions in latency and memory footprint, translating into substantial cost savings for cloud providers.

However, reliance on SSDs as backend storage introduced performance instability due to internal activities like garbage collection and wear leveling, leading to tail latencies. To address this, we introduced HEIMDALL, an efficient ML-based I/O admission method. HEIMDALL demonstrated remarkable improvements in tail latency reduction and inference throughput, ensuring stable performance even in the face of unpredictable SSD activities.

Furthermore, our solution, TINFETCH, focuses on optimizing the performance of secondary storage (SSD/HDD) read latency through advanced block prefetching techniques. By leveraging both heuristic and machine learning methods, TINFETCH outperformed existing prefetching solutions, achieving higher hit rates and reducing inference latency to sub-μs levels.

In summary, this dissertation highlights the critical role of storage supports in meeting the evolving demands of deep recommendation systems. Our solutions provide practical and efficient approaches to enhance performance predictability and cost-effectiveness, paving the way for future advancements in storage systems tailored for AI/ML workloads.

REFERENCES

[1] 99th Percentile Latency at Scale with Apache Kafka. https://www.confluent.io/blog/configure-kafka-to-minimize-latency/.

[2] Alibaba Block Traces. https://github.com/alibaba/block-traces.

[3] Apache Flume. http://flume.apache.org.

[4] Apache Hadoop. https://hadoop.apache.org/.

[5] Apache S4. http://incubator.apache.org/s4/.

[6] BUG: CASSANDRA-6023: CAS should distinguish promised and accepted ballots. https://issues.apache.org/jira/browse/CASSANDRA-6023.

[7] BUG: HBASE-4397: -ROOT-, .META. tables stay offline for too long in recovery phase after all RSs are shutdown at the same time. https://issues.apache.org/jira/browse/HBASE-4397.

[8] Cassandra - Speculative Execution for Reads / Eager Retries. https://issues.apache.org/jira/browse/CASSANDRA-4705.

[9] Chameleon Cloud Testbed. https://www.chameleoncloud.org/.

[10] Data Science vs. Machine Learning: What's the Difference? https://www.coursera.org/articles/data-science-vs-machine-learning.

[11] EVStore GitHub. https://github.com/ucare-uchicago/ev-store-dlrm.

[12] fio - Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[13] FlyMC Open-Sourced Code. http://ucare.cs.uchicago.edu/projects/FlyMC/.

[14] Java Path Finder. https://babelfish.arc.nasa.gov/trac/jpf.

[15] Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/.

[16] Keynote: Storage for HyperScalers - Presented by Mark Carlson at SNIA Storage Developer Conference 2016. https://www.snia.org/sites/default/files/SDC/2016/presentations/keynote_general/MarkCarlson_HyperscaleStorage.pdf.

[17] Micron 9300 MAX NVMe SSDs + Red Hat Ceph Storage. https://www.micron.com/-/media/client/global/documents/products/other-documents/micron_9300_and_red_hat_ceph_reference_architecture.pdf.

[18] ML Engineer vs Data Scientist. https://neptune.ai/blog/ml-engineer-vs-data-scientist.

[19] MongoDB - Basic Support for Operation Hedging in NetworkInterfaceTL. https://jira.mongodb.org/browse/SERVER-45432.

[20] RocksDB. http://rocksdb.org/.

[21] Storage Latency in Flash Arrays. https://www.violinsystems.com/wp-content/uploads/Storage-Mojo-WP-storage-latency.pdf.

[22] The Evolution of Image Classification Explained. https://stanford.edu/~shervine/blog/evolution-image-classification-explained.

[23] Why Deterministic Storage Performance is Important. https://www.architecting.it/blog/deterministic-storage-performance/.

[24] Marissa mayer at web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006.

[25] MSR Cambridge Traces. http://iotta.snia.org/traces/block-io/388, 2007.

[26] Download Terabyte Click Logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs/, 2013.

[27] Click-Through Rate Prediction: Predict whether a mobile ad will be clicked. https://www.kaggle.com/c/avazu-ctr-prediction, 2014.

[28] Display Advertising Challenge. https://www.kaggle.com/c/criteo-display-ad-challenge, 2014.

[29] Akamai online retail performance report: Milliseconds are critical. https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp, 2017.

[30] A Real-Time, Low Latency, Key-Value Solution Combining Samsung Z-SSDTM and Levyx's HeliumTM Data Store. https://www.samsung.com/semiconductor/global.semi.static/Whitepaper_Samsung_Low_Latency_Z-SSD_Levyx_Helium_Data_Store_Jan2018.pdf, 2018.

[31] Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, 2018.

[32] Notes from the ai frontier insights from hundreds of use cases. https://www.mckinsey.com/featured-insights/artificial-intelligence/, 2018.

[33] Tencent Block Storage. http://iotta.snia.org/traces/parallel/27917?n=10&page=1, 2018.

[34] Leap GitHub repository. https://github.com/SymbioticLab/Leap, 2019.

[35] Use cases of recommendation systems in business current applications and methods. https://emerj.com/ai-sector-overviews/use-cases-recommendation-systems/, 2019.

[36] Delta LSTM GitHub repository. https://github.com/Chandranil2606/, 2020.

[37] SQLite. https://www.sqlite.org/index.html, 2020.

[38] How machine learning powers Facebook's News Feed ranking algorithm. https://engineering.fb.com/2021/01/26/ml-applications/news-feed-ranking/, 2021.

[39] Pythia GitHub repository. https://github.com/CMU-SAFARI/Pythia, 2021.

[40] Alibaba Block Traces. https://github.com/alibaba/block-traces, 2022.

[41] Benchmarks for Java In Memory Caches. https://github.com/cache2k/cache2k-benchmark, 2022.

[42] CORTX-Motr. https://github.com/Seagate/cortx-motr, 2022.

[43] Memory Prices. https://memory.net/memory-prices/, 2022.

[44] SGDP GitHub repository. https://github.com/yyysjz1997/SGDP, 2023.

[45] Najmeddine Abdennour, Tarek Ouni, and Nader Ben Amor. The importance of signal preprocessing for machine learning: The influence of Data scaling in a driver identity classification. In *ACS 18th International Conference on Computer Systems and Applications (AICCSA)*, 2021.

[46] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating Deep Recommendation Model Training. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[47] Sam Ainsworth and Timothy M. Jones. Graph prefetching using data structure knowledge. In *Proceedings of the 30th International Conference on Supercomputing (ICS)*, 2016.

[48] Mohammadamin Ajdari, Pouria Peykani Sani, Amirhossein Moradi, Masoud Khanalizadeh Imani, Amir Hossein Bazkhanei, and Hossein Asadi. Re-architecting I/O Caches for Emerging Fast Storage Devices. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[49] Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. Improving Storage Systems Using Machine Learning. In *ACM Transaction on Storage*, 2023.

[50] Alaa R Alameldeen and David A Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[51] Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. SSD failures in the field: symptoms, causes, and prediction models. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[52] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[53] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[54] Patrick Anderson, Erika Blancada Aranas, Youssef Assaf, Raphael Behrendt, Richard Black, Marco Caballero, Pashmina Cameron, Burcu Canakci, Thales De Carvalho, Andromachi Chatzieleftheriou, Rebekah Storan Clarke, James Clegg, Daniel Cletheroe, Bridgette Cooper, Tim Deegan, Austin Donnelly, Rokas Drevinskas, Alexander L. Gaunt, Christos Gkantsidis, Ariel Gomez Diaz, István Haller, Freddie Hong, Teodora Ilieva, Shashidhar Joshi, Russell Joyce, Mint Kunkel, David Lara, Sergey Legtchenko, Fanglin Linda Liu, Bruno Magalhães, Alana Marzoev, Marvin McNett, Jayashree Mohan, Michael Myrah, Trong Nguyen, Sebastian Nowozin, Aaron Ogus, Hiske Overweg, Antony I. T. Rowstron, Maneesh Sah, Masaaki Sakakura, Peter Scholtz, Nina Schreiner, Omer Sella, Adam Smith, Ioan A. Stefanovici, David Sweeney, Benn Thomsen, Govert Verkes, Phil Wainman, Jonathan Westcott, Luke Weston, Charles Whittaker, Pablo Wilke Berenguer, Hugh Williams, Thomas Winkler, and Stefan Winzeck. Project Silica: Towards Sustainable Cloud Archival Storage in Glass. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[55] Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Valmiki Rampersad, Jens Axboe, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Shishir Juluri, Akshat Nanda, Manoj Wodekar, Dheevatsa Mudigere, Krishnakumar Nair, Maxim Naumov, Chris Peterson, Mikhail Smelyanskiy, and Vijay Rao. Supporting Massive DLRM Inference Through Software Defined Memory. https://arxiv.org/abs/2110.11489, 2021.

[56] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung Kyu Lim, and Hyesoon Kim. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[57] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of The 3rd International Conference on Learning Representations (ICLR)*, 2015.

[58] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient Compression Supercharged High-Performance Data Parallel DNN Training. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[59] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing Neural Network Architectures using Reinforcement Learning. In *5th International Conference on Learning Representations (ICLR)*, 2017.

[60] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[61] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *Proceedings of The FAST '04 Conference on File and Storage Technologies*, 2004.

[62] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communication of the ACM*, 60(4):48–54, 2017.

[63] Jayanta Basak, Kushal Wadhwani, and Kaladhar Voruganti. Storage workload identification. In *ACM Transaction on Storage*, 2016.

[64] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[65] Nathan Beckmann and Daniel Sanchez. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[66] Nathan Beckmann and Daniel Sanchez. Maximizing Cache Performance Under Uncertainty. In *Proceedings of the 23rd international symposium on High Performance Computer Architecture (HPCA-23)*, February 2017.

[67] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *54rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)*, 2021.

[68] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and More Harchol-Balter. RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[69] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. Generating Complex, Realistic Cloud Workloads using Recurrent Neural Networks. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[70] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. The Art and Practice of Data Science Pipelines: A Comprehensive Study of Data Science Pipelines In Theory, In-The-Small, and In-The-Large. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[71] Simona Boboila and Peter Desnoyers. Performance models of flash-based solid-state drives for real workloads. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[72] Andrew P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. In *The Journal of the Pattern Recognition Society Volume 30*, 1997.

[73] Rodrigo Bruno, Duarte Patricio, Jose Simao, Luis Veiga, and Paulo Ferreira. Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications. In *Proceedings of the 2019 EuroSys Conference (EuroSys)*, 2019.

[74] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[75] Kishan K. C., Rui Li, and Mahdi Gilany. Joint inference for neural network depth and dropout regularization. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NIPS)*, 2021.

[76] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery. In *Computing Research Repository*, 2017.

[77] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ACM Special Interest Group on Computer Architecture (SIGARCH) News*, 1991.

[78] Zhen Cao, Geoff Kuenning, and Erez Zadok. Carver: Finding Important Parameters for Storage System Tuning. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*, 2020.

[79] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*, 2020.

[80] Chandranil Chakraborttii and Heiner Litz. Learning I/O Access Patterns to Improve Prefetching in SSDs. In *European Conference on Machine Learning (ECML)*, 2020.

[81] Haihua Chen, Jiangping Chen, and Junhua Ding. Data Evaluation and Enhancement for Quality Improvement of Machine Learning. In *IEEE Transactions on Reliability*, 2021.

[82] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[83] Yu Chen, Yong Zhang, Jiacheng Wu, Jin Wang, and Chunxiao Xing. Revisiting data prefetching for database systems with machine learning techniques. In *Proceedings of the 37th International Conference on Data Engineering (ICDE)*, 2021.

[84] Audrey Cheng, David Chu, Terrance Li, Jason Chan, Natacha Crooks, Joseph M. Hellerstein, Ion Stoica, and Xiangyao Yu. Take Out the TraChe: Maximizing (Tra)nsactional Ca(che) Hit Rate. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[85] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & Deep Learning for Recommender Systems. In *Proceedings of The 1st Workshop on Deep Learning for Recommender Systems (DLRS@RecSys)*, 2016.

[86] Michael Chow, Kaushik Veeraraghavan, Michael J Cafarella, and Jason Flinn. Dqbarge: Improving data-quality tradeoffs in large-scale internet services. In *OSDI*, pages 771–786, 2016.

[87] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. Unearthing inter-job dependencies for better cluster scheduling. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[88] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[89] Kai Lai Chung. Markov chains. In *Springer-Verlag*, 1967.

[90] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[91] Robert Cooksey, Stéphan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[92] Paul Covington, Jay Adams, and Emre Sargin. Deep Neural Networks for YouTube Recommendations. In *Proceedings of The 10th ACM Conference on Recommender Systems (RecSys)*, 2016.

[93] Weihao Cui, Zhenhua Han, Lingji Ouyang, Yichuan Wang, Ningxin Zheng, Lingxiao Ma, Yuqing Yang, Fan Yang, Jilong Xue, Lili Qiu, Lidong Zhou, Quan Chen, Haisheng Tan, and Minyi Guo. Optimizing Dynamic Neural Networks with Brainstorm. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[94] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[95] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1990.

[96] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and image processing*, 1980.

[97] Anwesha Das, Frank Mueller, Paul Hargrove, Eric Roman, and Scott B. Baden. Doomsday: predicting which node will fail when on supercomputers. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[98] Jesse Davis and Mark Goadrich. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, 2006.

[99] Jesse Davis and Mark H. Goadrich. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd international conference on Machine learning*, 2006.

[100] Jeffrey Dean and Luiz Andre Barroso. The Tail at Scale. *Communications of the ACM (CACM)*, 56(2), 2013.

[101] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[102] Christina Delimitrou and Christos Kozyrakis. Bolt: I Know What You Did Last Summer... In the Cloud. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[103] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[104] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2007.

[105] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[106] Pedro Domingos. A Few Useful Things to Know about Machine Learning. In *Communications of the ACM (CACM)*, 2012.

[107] Zhu Dongjie, D. Haiwen, S. Yundong, and T. Zhaoshuo. CTDGM: a data grouping model based on cache transaction for unstructured data storage systems, 2020.

[108] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[109] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David E. Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[110] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of The 2nd Conference on Machine Learning and Systems (MLSys)*, 2019.

[111] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[112] Nima Elyasi, Changho Choi, Anand Sivasubramaniam, Jingpei Yang, and Vijay Balakrishnan. Trimming the Tail for Deterministic Read Performance in SSDs. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2019.

[113] Ernest Allen Emerson. *The Beginning of Model Checking: A Personal Perspective*. Springer-Verlag, 2008.

[114] WU Fengguang, XI Hongsheng, and XU Chenfeng. On the design of a new linux readahead framework. In *ACM SIGOPS Operating Systems Review (OSR)*, 2008.

[115] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Efficient and Robust Automated Machine Learning. In *Proceedings of the 30th Conference on Neural Information Processing Systems (NIPS)*, 2015.

[116] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. Towards a Machine Learning-Assisted Kernel with LAKE. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[117] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[118] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and Flexibility in Distribution of Hot Content. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[119] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 2017 EuroSys Conference (EuroSys)*, 2017.

[120] John WC Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *ACM SIGMICRO Newsletter*, 1992.

[121] Alexander Fuerst, Stanko Novakovic, Iñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting VMs in cloud platforms. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[122] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[123] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[124] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. DeepPrefetcher: A Deep Learning Framework for Data Prefetching in Flash Storage Devices. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2020.

[125] Lokesh Gidra, Gael Thomas, Julien Sopena, and Marc Shapiro. A study of the Scalability of Stop-the-World Garbage Collectors on Multicore. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[126] Lokesh Gidra, Gael Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[127] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Domain Adaptation for Large-Scale Sentiment Classification: A Deep Learning Approach. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, 2011.

[128] Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. volume 1032, 1996.

[129] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997.

[130] Patrice Godefroid and Nachiappan Nagappan. Concurrency At Microsoft - An Exploratory Study. Technical report, Microsoft Research, 2008.

[131] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. gSampler: General and Efficient GPU-based Graph Sampling for Graph Learning. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[132] Klaus Greff, Antti Rasmus, Mathias Berglund, Tele Hotloo Hao, Jurgen Schmidhuber, and Harri Valpola. Tagger: Deep Unsupervised Perceptual Grouping. In *Proceedings of the 30th Conference on Neural Information Processing Systems (NIPS)*, 2016.

[133] Peng Gu, Yifeng Zhu, Hong Jiang, and Jun Wang. Nexus: a novel weighted-graph-based prefetching algorithm for metadata servers in petabyte-scale storage systems. In *Proceedings of the 6th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2006.

[134] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[135] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[136] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[137] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[138] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: the who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web (WWW)*, 2013.

[139] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook's dnn-based personalized recommendation. In *Proceedings of The 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[140] Wook-Shin Han, Kyu-Young Whang, and Yang-Sae Moon. A formal framework for prefetching based on the type-level access pattern in object-relational DBMSs. In *IEEE Transactions on Knowledge and Data Engineering*, 2005.

[141] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[142] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[143] Md. E. Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[144] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

[145] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[146] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of the*

24rd International Symposium on High Performance Computer Architecture (HPCA-24), 2018.

[147] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. IOCost: block IO control for containers in datacenters. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[148] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, 2015.

[149] Gisli R. Hjaltason and Hanan Samet. Properties of embedding methods for similarity searching in metric spaces. *IEEE Transactions on Pattern Analysis and machine intelligence*, 2003.

[150] Seokin Hong, Bulent Abali, Alper Buyuktosunoglu, Michael B Healy, and Prashant J Nair. Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[151] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[152] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, 2009.

[153] Rishabh R. Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[154] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 219–230, 2013.

[155] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[156] László Jeni, Jeffrey Cohn, and Fernando De la Torre. Facing imbalanced data - recommendations for the use of performance metrics. 2013.

[157] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of The 2005 USENIX Annual Technical Conference*, 2005.

[158] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2002.

[159] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA-23)*, 2017.

[160] Michael I. Jordan and Robert A. Jacob. Hierarchical mixtures of experts and the EM algorithm. In *IEEE International Joint Conference on Neural Network (IJCNN)*, 1993.

[161] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[162] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[163] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Kandemir. Design of a Host Interface Logic for GC-Free SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 8(1), May 2019.

[164] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[165] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazieres, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for Microsecond-scale Tail Latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[166] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[167] Hui Kang and Jennifer L. Wong. To hardware prefetch or not to prefetch?: a virtualized environment study and core binding approach. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[168] Anne Kao and Steve R. Poteet. *Natural language processing and text mining*. 2007.

[169] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[170] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[171] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, July 2020.

[172] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K. Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali Raza Butt. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST)*, 2023.

[173] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. RECL: Responsive Resource-Efficient Continuous Learning for Video Analytics. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[174] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[175] Bryan S. Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an Autonomic SSD Architecture. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[176] Hyojun Kim and Umakishore Ramachandran. Flashfire: Overcoming the performance bottleneck of flash storage technology. In *Technical Report Georgia Institute of Technology*, 2010.

[177] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[178] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[179] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *Proceedings of the 17th USENIX Symposium on File and Storage Technologies (FAST)*, 2019.

[180] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[181] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. Coordinating Garbage Collection for Arrays of Solid-State Drives. *IEEE Transactions on Computers (TC)*, 63(4), April 2014.

[182] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-state Drives. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[183] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[184] Guy K Kloss. Automatic c library wrapping ctypes from the trenches. 2009.

[185] Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. TeraHeap: Reducing Memory Pressure in Managed Big Data Frameworks. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[186] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[187] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An Informed Storage Cache for Deep Learning. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*, 2020.

[188] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold'em or fold'em?: Aggregation queries under performance variations. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 7, 2016.

[189] Daniar H. Kurniawan, Ruipu Wang, Kahfi Zulkifli, Fandi Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. EVStore: Storage and Caching Capabilities for Scaling Embedding Tables in Deep Recommendation Systems. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[190] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[191] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[192] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[193] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. Lynx: A learning linux prefetching mechanism for ssd performance model. In *Proceedings of the IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, 2016.

[194] Fan Lai, Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, Zeliang Chen, Ellie Wen, Paul Rivera, Jie You, Chun cheng Jason Chen, and Mosharaf Chowdhury. AdaEmbed: Adaptive Embedding for Large-Scale Recommendation Models. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[195] Leslie Lamport. The part-time parliament (paxos). *ACM Transactions on Computer Systems*, 16(2), May 1998.

[196] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[197] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-Aware DRAM Controllers. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*, 2008.

[198] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[199] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1999.

[200] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[201] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W. Lee, and Tae Jun Ham. Merci: efficient embedding reduction on commodity hardware via sub-query memoization. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[202] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. SAMC: A Fast Model Checker for Finding Heisenbugs in Distributed Systems. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[203] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[204] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[205] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proceedings of The 2nd Conference on Machine Learning and Systems (MLSys)*, 2019.

[206] Timothée Lesort, Massimo Caccia, and Irina Rish. Understanding Continual Learning Settings with Data Distribution Drift Analysis. In *Computing Research Repository*, 2022.

[207] Jacob Leverich, Christos Kozyrakis, and Jacob Leverich. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proceedings of the 2014 EuroSys Conference (EuroSys)*, 2014.

[208] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[209] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. First-generation Memory Disaggregation for Cloud Platforms. https://arxiv.org/pdf/2203.00241, 2022.

[210] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[211] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern

Flash Storage. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[212] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[213] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. TaP: Table-based Prefetching for Storage Caches. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[214] Nanqinqin Li, Mingzhe Hao, Xing Lin, Huaicheng Li, Levent Toksoz, Tim Emami, and Haryadi S. Gunawi. Fantastic SSD Internals and How to Learn and Use Them. In *Proceedings of the 15th ACM International Systems and Storage Conference (SYSTOR)*, 2022.

[215] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST)*, 2023.

[216] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[217] Xing Li, Qiquan Shi, Gang Hu, Lei Chen, Hui Mao, Yiyuan Yang, Mingxuan Yuan, Jia Zeng, and Zhuo Cheng. Block access pattern discovery via compressed full tensor transformer. In *Proceedings of the ACM on Conference on Information and Knowledge Management*, 2021.

[218] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srivivasan, and Yuanyuan Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST)*, 2004.

[219] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[220] Jianwei Liao, François Trahay, Balazs Gerofi, and Yutaka Ishikawa. Prefetching on storage servers through mining access patterns on blocks. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2015.

[221] Leo Liberti, Carlile Lavor, Nelson Maculan, and Antonio Mucherino. Euclidean distance geometry and applications. *SIAM review*, 2014.

[222] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[223] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. CRISP: critical slice prefetching. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[224] Christianto C. Liu, Ilya Ganusov, Martin Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gap with 3D IC technology. In *IEEE Design and Test (DT)*, 2005.

[225] Chun-Yi Liu, Jagadish B. Kotra, Myoungsoo Jung, Mahmut T. Kandemir, and Chita R. Das. SOML Read: Rethinking the Read Operation Granularity of 3D NAND SSDs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[226] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[227] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[228] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[229] Lei Liu, Xinglei Dou, and Yuetao Chen. Intelligent Resource Scheduling for Co-located Latency-critical Services: A Multi-Model Collaborative Learning Approach. In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST)*, 2023.

[230] Sidi Lu, Bing Luo, Tirthak Patel, Yongtao Yao, Devesh Tiwari, and Weisong Shi. Making Disk Failure Predictions SMARTer! In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*, 2020.

[231] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[232] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[233] Martin Maas, Krste Asanovic, Tim Harris, and John Kubiatowicz. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[234] Zheda Mai, Ruiwen Li, Jihwan Jeong, David Quispe, Hyunwoo Kim, and Scott Sanner. Online continual learning in image classification: An empirical survey. In *Neurocomputing*, 2022.

[235] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[236] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[237] Tony Mauro. Adopting Microservices at Netflix: Lessons for Architectural Design. `https://www.nginx.com/blog/microservices-at-netflixarchitectural-best-practices`, 2015.

[238] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[239] Andrew McCallum. Joint inference for natural language processing. In *Proceedings of the 13th Conference on Computational Natural Language Learning (CoNLL)*, 2009.

[240] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of The FAST '03 Conference on File and Storage Technologies*, 2003.

[241] Syed Akbar Mehdi, Deukyeon Hwang, Simon Peter, and Lorenzo Alvisi. ScaleDB: A Scalable, Asynchronous In-Memory Database. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[242] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. Het: Scaling out huge embedding model training via cache-enabled distributed framework. *arXiv:2112.07221*, 2021.

[243] Rino Micheloni. Solid-State Drive (SSD): A Nonvolatile Storage System. In *2017 Proceedings of the IEEE (Proc. IEEE)*, 2017.

[244] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[245] Pulkit A. Misra, María F. Borge, Iñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the 2019 EuroSys Conference (EuroSys)*, 2019.

[246] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[247] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning Massive Graph Embeddings on a Single Machine. In *Proceedings of The 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[248] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[249] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[250] James C. Mullikin. The vector distance transform in two and three dimensions. *CVGIP: Graphical Models and Image Processing*, 1992.

[251] Madanlal Musuvathi, Shaz Qadeer, Tom Ball, Gerard Basler, Piramanayakam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[252] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[253] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace set 388). In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[254] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. arXiv:1906.00091, 2019.

[255] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[256] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[257] Victor F Nicola, Asit Dan, and Daniel M Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1992.

[258] Mais Nijim. Modelling speculative prefetching for hybrid storage systems. In *2019 IEEE 14th International Conference on Networking, Architecture, and Storage (NAS)*, 2019.

[259] Mais Nijim, Ziliang Zong, Xiao Qin, and Yousef Nijim. Multi-layer prefetching for hybrid storage systems: algorithms, models, and evaluations. In *International Conference on Parallel Processing (ICPP) Workshop*, 2010.

[260] Iratxe Nino-Adan, Eva Portillo, Itziar Landa-Torres, and Diana Manjarres. Normalization influence on ANN-based models performance: A new proposal for Features' contribution analysis. In *IEEE Access*, 2021.

[261] Even Oldridge, Julio Perez, Ben Frederickson, Nicolas Koumchatzky, Minseok Lee, Zehuan Wang, Lei Wu, Fan Yu, Rick Zamora, O Yılmaz, Alec Gunny, and Vinh Nguyen. Merlin: a gpu accelerated recommendation framework. *Proceeding s of IRS*, 2020.

[262] E. Theodore L. Omtzigt, Peter Gottschling, Mark Seligman, and William Zorn. Universal Numbers Library: design and implementation of a high-performance reproducible number systems library. *arXiv:2012.11011*, 2020.

[263] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[264] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[265] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast File System. volume 6, pages 1092–1101. VLDB Endowment, 2013.

[266] Reena Panda, Yasuko Eckert, Nuwan Jayasena, Onur Kayiran, Michael Boyer, and Lizy Kurian John. Prefetching Techniques for Near-memory Throughput Processors. In *Proceedings of the 30th International Conference on Supercomputing (ICS)*, 2016.

[267] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression. In *Proceedings of the 20th USENIX Symposium on File and Storage Technologies (FAST)*, 2022.

[268] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Shanker Khudia, James Law, Parth Malani, Andrey Malevich, Nadathur Satish, Juan Miguel Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim M. Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. https://arxiv.org/abs/1811.09886, 2018.

[269] Chris Petersen, Wei Zhang, and Alexei Naberezhnov. Enabling NVMe I/O Determinism @ Scale. `https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_INVT-102A-1_Petersen.pdf`.

[270] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[271] Mia Primorac, Katerina J. Argyraki, and Edouard Bugnion. When to Hedge in Interactive Services. In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[272] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, and Heming Cui. Bidl: A High-throughput, Low-latency Permissioned Blockchain Framework for Datacenter Networks. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[273] Gang Qian, Shamik Sural, Yuelong Gu, and Sakti Pramanik. Similarity between euclidean and cosine angle distance for nearest neighbor queries. In *Proceedings of the 2004 ACM symposium on Applied computing*, 2004.

[274] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[275] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[276] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[277] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi A. Noghabi, and Jian Huang. BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[278] Benjamin Reidys, Yuqi Xue, Daixuan Li, Bharat Sukhwani, Wen-Mei Hwu, Deming Chen, Sameh W. Asaad, and Jian Huang. RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[279] B. D. Ripley. Neural Networks and Related Methods for Classification. In *Journal of the Royal Statistical Society. Series B (Methodological) Vol. 56, No. 3 (1994)*, 1994.

[280] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling Vision with Sparse Mixture of Experts. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NIPS)*, 2021.

[281] Liana V. Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *Proceedings of the 19th USENIX Symposium on File and Storage Technologies (FAST)*, 2021.

[282] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[283] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[284] Sultan Mahmud Sajal, Luke Marshall, Beibin Li, Shandan Zhou, Abhisek Pan, Konstantina Mellou, Deepak Narayanan, Timothy Zhu, David Dion, Thomas Moscibroda, and Ishai Menache. Kerveros: Efficient and Scalable Cloud Admission Control. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[285] Jose Renato Santos, Richard R. Muntz, and Berthier Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *ACM SIGMETRICS Performance Evaluation Review*, 2000.

[286] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[287] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. RecShard: statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[288] Amit Sharma, Jake M Hofman, and Duncan J Watts. Estimating the causal impact of recommendation systems from observational data. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation*, 2015.

[289] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation Functions in Neural Networks. In *International Journal of Engineering Applied Sciences and Technology (IJEAST)*, 2020.

[290] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Computing Research Repository*, 2017.

[291] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[292] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling Deep Learning Memory Access via Tilegraph. In *Proceedings of the 17th Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[293] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A hierarchical neural model of data prefetching. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[294] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. Disaggregated RAID Storage in Modern Datacenters. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[295] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[296] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV)*, 2010.

[297] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[298] Alan J Smith. Cache Memories. In *ACM Computing Surveys*, 1982.

[299] Hui Song and Guohong Cao. Cache-Miss-Initiated Prefetch in Mobile Environments. In *IEEE International Conference on Mobile Data Management (MDM)*, 2004.

[300] Hwanjun Song, Minseok Kim, Dongmin Park, and Jae-Gil Lee. Learning from Noisy Labels with Deep Neural Networks: A Survey. In *Computing Research Repository*, 2020.

[301] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[302] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[303] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[304] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Uma Maheswara Rao G., and Haryadi S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[305] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. LeaFTL: A Learning-Based Flash Translation Layer for Solid-State Drives. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[306] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[307] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[308] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[309] Boyu Tian, Qihang Chen, and Mingyu Gao. ABNDP: Co-optimizing Data Access and Load Balance in Near-Data Processing. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[310] Dimitra Tsigkari and Thrasyvoulos Spyropoulos. An approximation algorithm for joint caching and recommendations in cache networks. *IEEE Transactions on Network and Service Management*, 2022.

[311] Uresh Vahalia. Unix internals: The new frontiers, 1996.

[312] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[313] Ruben van den Goorbergh, Maarten van Smeden, Dirk Timmerman, and Ben Van Calster. The harm of class imbalance corrections for risk prediction models: illustration and simulation using logistic regression. In *Journal of the American Medical Informatics Association*, 2022.

[314] Shiva Verma. How to Create a Custom Loss Function | Keras, 2020.

[315] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE)*, 2018.

[316] S Vijayarani and R Janani. Text mining: open source tokenization tools-an analysis. *Advanced Computational Intelligence: An International Journal (ACII)*, 2016.

[317] Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. FlashEmbedding: storing embedding tables in SSD for large-scale recommender systems. In *Proceedings of The 12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2021.

[318] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

[319] Meng Wang, Cesar A. Stuardo, Daniar H. Kurniawan, Ray A. O. Sinurat, and Haryadi S. Gunawi. Layered Contention Mitigation for Cloud Storage. In *IEEE 15th International Conference on Cloud Computing*, 2022.

[320] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *Proceedings of ADKDD*, 2017.

[321] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the Web Conference 2021*, 2021.

[322] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel S. Berger, Christos Kozyrakis, and Ricardo Bianchini. SOL: safe on-node learning in cloud platforms. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[323] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[324] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[325] Shih wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. Machine learning-based prefetch optimization for data center applications. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[326] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable and High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[327] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: transparent memory offloading in datacenters. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[328] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal streams in commercial server applications. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[329] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. In *Machine Learning (ML)*, 1996.

[330] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Tom Anderson. Verdi: A framework for formally verifying distributed system implementations. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[331] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. RecSSD: near data processing for solid state drive based recommendation inference. In *Proceedings of The 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[332] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, Gregory R. Ganger Baleen: ML Admission & Prefetching for Flash Caches. In *Proceedings of the 22nd USENIX Symposium on File and Storage Technologies (FAST)*, 2024.

[333] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal prefetching without the off-chip metadata. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[334] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Efficient metadata management for irregular data prefetching. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[335] Nan Wu and Yuan Xie. A survey of machine learning for computer architecture and systems. In *Proceedings of the ACM Computing Surveys (CSUR)*, 2022.

[336] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. Overcome the GC-induced Performance Variability in SSD-based RAIDs with Request Redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(5), May 2019.

[337] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. GC-aware Request Steering with Improved Performance and Reliability for SSD-based RAIDs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[338] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[339] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[340] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: an efficient gpu embedding cache for personalized recommendations. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

[341] Xing Xie, Jianxun Lian, Zheng Liu, Xiting Wang, Fangzhao Wu, Hongwei Wang, and Zhongxia Chen. Personalized recommendation systems: Five hot research topics you must know. *Microsoft Research Lab-Asia*, 2018.

[342] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Andy Rudoff, and Steven Swanson. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[343] Rui Xu, Xi Jin, Linfeng Tao, Shuaizhi Guo, Zikun Xiang, and Teng Tian. An efficient resource-optimized learning prefetcher for solid state drives. In *Design Automation and Test in Europe (DATE)*, 2018.

[344] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. SPFresh: Incremental In-Place

Update for Billion-Scale Vector Search. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[345] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[346] Ming Xue and Changjun Zhu. The socket programming and software design for communication based on client/server. In *Pacific-Asia Conference on Circuits, Communications and Systems*, 2009.

[347] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[348] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Transactions on Storage (TOS)*, 13(3), 2017.

[349] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[350] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. Mixed-precision embedding using a cache. *arXiv:2010.11305*, 2020.

[351] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST)*, 2023.

[352] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[353] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[354] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. Random Walks on Huge Graphs at Cache Efficiency. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[355] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[356] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[357] Yiyuan Yang, Rongshang Li, Qiquan Shi, Xijun Li, Gang Hu, Xing Li, and Mingxuan Yuan. SGDP: A Stream-Graph Neural Network Based Data Prefetcher. In *IEEE International Joint Conference on Neural Network (IJCNN)*, 2023.

[358] Xue Ying. An Overview of Overfitting and its Solutions. In *2018 1st International Conference on Computer Information Science and Application Technology (CISAT)*, 2018.

[359] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[360] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[361] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[362] Adar Zeitak and Adam Morrison. Cuckoo Trie: Exploiting Memory-Level Parallelism for Efficient DRAM Indexing. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[363] Dan Zhang, Safeen Huda, Ebrahim M. Songhori, Kartik Prabhu, Quoc V. Le, Anna Goldie, and Azalia Mirhoseini. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[364] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[365] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Name Sung Kim, Jihong Kim, and Myoungsoo Jung. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[366] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 2015.

[367] Xu Zhang, Siddhartha Sen, Daniar H. Kurniawan, Haryadi S. Gunawi, and Junchen Jiang. E2E: Embracing User Heterogeneity to Improve Quality of Experience on the Web. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[368] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: admission control for performance-critical RPCs in datacenters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[369] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[370] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent block storage traces (SNIA IOTTA trace set 27917). In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[371] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In *Proceedings of The 3rd Conference on Machine Learning and Systems (MLSys)*, 2020.

[372] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems*, 2020.

[373] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019.

[374] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed H. Chi. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems (RecSys)*, 2019.

[375] Alice Zheng and Amanda Casari. Feature engineering for machine learning: principles and techniques for data scientists., 2018.

[376] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. DGL-KE: Training Knowledge Graph Embeddings at Scale. In *Proceedings of The 43rd International ACM SIGIR conference on research and development in Information Retrieval (SIGIR)*, 2020.

[377] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[378] Randal Burns Da Zheng and Alexander S. Szalay. A parallel page cache: IOPS and caching for multicore systems. In *4th USENIX Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE)*, 2012.

[379] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep Interest Evolution Network for Click-Through Rate Prediction. In *Proceedings of The 31st Innovative Applications of Artificial Intelligence Conference*, 2019.

[380] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

[381] Xiaojin Zhu and Zhoubin Ghahramani. Learning from labeled and unlabeled data with label propagation, 2002.

[382] Xiaotong Zhuang and Hsien-Hsin S. Lee. Reducing Cache Pollution via Dynamic Data Prefetch Filtering. In *IEEE Transactions on Computers (TC)*, 2007.

[383] Lorenzo Zuolo, Cristian Zambelli, Rino Micheloni, Davide Bertozzi, and P. Olivo. Analysis of reliability/performance trade-off in Solid State Drives. In *IEEE International Symposium on Reliability Physics (IRPS)*, 2014.