Buoy: Efficient and Effective Cache Replacement for Prefix Caching

LIANG WANG, Huazhong University of Science and Technology, China
RANJUN JIA, Huazhong University of Science and Technology, China
KAI WANG, Huazhong University of Science and Technology, China
KAI LU*, Huazhong University of Science and Technology, China
JIGUANG WAN, Huazhong University of Science and Technology, China

Modern large language model (LLM) systems widely employ prefix caching to enable key-value (KV) cache reuse across different queries to minimize inference costs. At the heart of prefix caching is the replacement algorithm, which is crucial for managing limited cache space across the GPU–CPU memory hierarchy. However, the prevalent least-recently-used (LRU) algorithm is not prefix-aware or tier-aware, and remains far from optimal. In this paper, we present Buoy, a novel and general cache replacement policy for prefix caching. Buoy achieves quick demotion and considers prefix-matched access characteristics to improve cache efficiency (hit rate). Meanwhile, Buoy enables effective cache tiering to reduce access latency upon cache hits. We evaluate Buoy on real-world traces, and the experimental results demonstrate its superiority in improving hit rates and prefill throughput over state-of-the-art algorithms. Compared to LRU, Buoy achieves 6–93% higher cache hit rates and increases LLM prefill throughput by 8–16%.

CCS Concepts: • **Information systems → Information storage systems**; • **Computer systems organization**;

Additional Key Words and Phrases: Large Language Models, Prefix Caching, Cache Replacement

ACM Reference Format:
Liang Wang, Ranjun Jia, Kai Wang, Kai Lu, and Jiguang Wan. 2026. Buoy: Efficient and Effective Cache Replacement for Prefix Caching. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 263 (September 2026), 27 pages. https://doi.org/10.1145/3837101

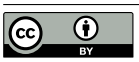
1 Introduction

Large language models (LLMs) are ubiquitous in modern AI services, such as conversational assistants [10, 38, 54] and retrieval-augmented generation (RAG) systems [13, 16, 24]. A major performance bottleneck in LLM inference lies in the prefill stage, where the model processes the input prompt and constructs key–value (KV) caches for subsequent token generation. The KV caches computed for one prompt can be reused by other prompts that share the same prefix, eliminating the need for redundant computation.

Realizing the significant reuse potential of prefix KV caches, recent LLM serving systems [20, 60] aggressively retain active prefix KV caches across GPU memory (HBM) and CPU memory (DRAM)

*Corresponding author.

Authors' Contact Information: Liang Wang, Huazhong University of Science and Technology, Wuhan, China, liangw@hust.edu.cn; Ranjun Jia, Huazhong University of Science and Technology, Wuhan, China, m202474017@hust.edu.cn; Kai Wang, Huazhong University of Science and Technology, Wuhan, China, m202474156@hust.edu.cn; Kai Lu, Huazhong University of Science and Technology, Wuhan, China, kailu@hust.edu.cn; Jiguang Wan, Huazhong University of Science and Technology, Wuhan, China, jgwan@hust.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/9-ART263
https://doi.org/10.1145/3837101

to enable cache reuse across different queries. However, the capacity of GPU and CPU memory remains insufficient for storing all prefix KV caches over time [27]. Thus, cache replacement becomes essential for determining which prefix KV caches to retain and which to evict under limited memory budgets. The classic least-recently-used (LRU) is one of the most prevalent cache replacement policies and has been widely adopted. In LLM serving systems, prefix KV caches are typically managed at chunk granularity [20, 27, 38, 45], and LRU evicts chunks based on their access recency, prioritizing the most recently accessed ones for retention.

Prefix KV caching raises new challenges for cache replacement. For prefix caching, a cache should be (1) *efficient*, maximizing the hit rate to avoid redundant recomputation; and (2) *effective*, minimizing the LLM prefill latency. However, traditional cache replacement policies are not designed to meet the requirements of prefix caching:

- Efficiency. Prefix KV cache accesses are not independent, unlike accesses to objects, in-memory key–value pairs, or blocks. Accessing a prefix KV chunk requires first accessing all of its ancestor prefixes. Traditional cache replacement policies do not account for such prefix-matched access characteristics and miss this opportunity to further improve cache efficiency.
- Effectiveness. In traditional caching systems, cache efficiency and effectiveness are closely aligned, as a high hit rate allows most requests to be served by the cache with short latencies. However, due to the relatively large size of KV caches in hierarchical caching, efficiency and effectiveness are no longer fully aligned. A cache hit in DRAM can still introduce additional data transfer delay, which lies on the critical path of LLM prefilling.

Meanwhile, some design principles from traditional cache replacement remain applicable to prefix KV caching. In particular, prior work [38, 45] reveals that prefix KV cache hits exhibit highly skewed reuse patterns, driven by two factors. First, a small fraction of users account for the majority of requests over a period of time, and since prefix KV cache hits primarily occur across queries from the same user, this imbalance naturally leads to skewed cache reuse. Second, even with a balanced request distribution across users, individual users tend to issue similar queries with overlapping prefixes in their prompts or contexts. Quick demotion [50, 51, 57] is a key idea for improving cache efficiency under skewed workloads, as it rapidly evicts unpopular prefixes to make space for prefixes with strong reuse potential.

Motivated by these observations, we rethink cache replacement from the perspective of prefix caching. First, we analyze the prefix-matched access characteristics of prefix KV caches and show how prefix-aware cache replacement can better preserve shared prefixes. Second, we demonstrate quick demotion as a key mechanism for improving cache efficiency under skewed prefix reuse. Third, we reveal that achieving high cache hit rates does not necessarily maximize cache hits in HBM, and can instead introduce non-trivial KV cache transfer overhead along the critical path of LLM prefilling.

Guided by these insights, we design Buoy, an efficient and effective cache replacement algorithm for prefix KV caching. Buoy incorporates quick demotion as a core replacement idea to filter out most prefixes after they are inserted, reserving cache space for popular prefixes. To this end, newly generated KV caches are first placed in the DRAM queue and are promptly evicted if not reused; lightweight second-chance reuse feedback also accelerates the removal of unpopular prefixes, reducing cache pollution under skewed workloads. Buoy further incorporates depth-aware cache admission to control which prefixes are allowed to enter the cache hierarchy. By leveraging prefix depth and reuse signals, this mechanism selectively admits prefix chunks with high reuse potential and prevents deep, low-value prefixes from occupying cache space. In addition, Buoy retains recently evicted prefixes in a lightweight ghost cache, enabling efficient reuse validation when prefixes are requested again. Finally, Buoy explicitly aligns cache replacement with hierarchical cache effectiveness by biasing cache hits toward HBM, increasing the fraction of cache hits served

directly from HBM and thereby reducing host-to-GPU transfer overhead and improving LLM prefill performance.

We evaluate Buoy through a comprehensive experimental study. Our evaluation covers two representative models (Llama-8B and Mistral-24B) on GPU servers with hierarchical HBM–DRAM memory using large-scale real-world request traces. We compare Buoy against both traditional cache replacement policies and recent cache replacement studies for prefix caching [17, 25, 45], and report key metrics including cache hit rate, prefill throughput, and time-to-first-token (TTFT). We first quantify cache efficiency with a prefix KV cache simulator, analyzing hit-rate breakdowns across HBM and DRAM and demonstrating Buoy 's advantage in overall hit rate as well as its ability to concentrate hits in HBM. We then measure LLM serving performance with different cache replacement policies, showing that Buoy delivers the highest prefill throughput and the lowest TTFT across models and traces. We further analyze the robustness of Buoy along multiple dimensions, including temporal behavior, chunk granularity, and cache size. In addition, we extend the cache hierarchy to include SSD in order to study three-tier cache settings with external storage and their impact on prefix caching performance. Finally, we conduct ablation studies to isolate the impact of key design choices in Buoy.

Overall, our contributions can be summarized as follows:

• We identify key insights about cache replacement for prefix caching by revealing prefix-matched access characteristics, the efficiency benefits of quick demotion, and the gap between cache efficiency and LLM prefill performance in hierarchical caches (§3).

• We design Buoy for efficient and effective cache replacement by enabling quick demotion and depth-aware cache admission while reducing host-to-GPU KV cache transfers on cache hits (§4).

• We evaluate Buoy on real-world traces, demonstrating that it achieves higher cache efficiency and LLM prefill throughput than state-of-the-art algorithms (§5).

2 Background

2.1 LLM Inference and Prefix KV Caching

We first introduce the standard LLM inference process and the role of KV caching for efficient LLM serving.

Prefilling and decoding. LLM inference typically consists of two stages: prefill (prompt computation) and decode (token generation) [6, 16, 61]. During the prefill stage, the LLM consumes the input prompt, processing it through all LLM transformer layers to produce the first output token. The subsequent decode stage iteratively generates one new token per step by forwarding the current sequence (including the prompt and previously generated tokens) through the LLM, repeating this procedure until the output sequence is completed.

KV caches. To accelerate the decoding stage, key-value (KV) caches are widely adopted in LLM inference. In transformer-based LLMs, each token processed by the self-attention layers generates the corresponding key and value representations. During the prefill phase, the key and value tensors for all input tokens are computed and cached on the GPU. In the decoding stage, instead of recomputing the attention for the entire token sequence at every step, the model reuses the cached KV tensors from previous tokens. This caching mechanism allows the model to efficiently attend to the entire sequence history, improving decoding throughput.

Prefix caching. Besides reusing KV caches across prefill and decode stages of a single request, the KV caches can also be reused across different requests if they have the same token prefix. When multiple requests share a common prefix in their input sequences, the corresponding KV caches generated for this shared prefix remain valid for all such requests. Hence, current LLM serving systems [20, 60] will retain the KV caches of completed requests, instead of releasing them

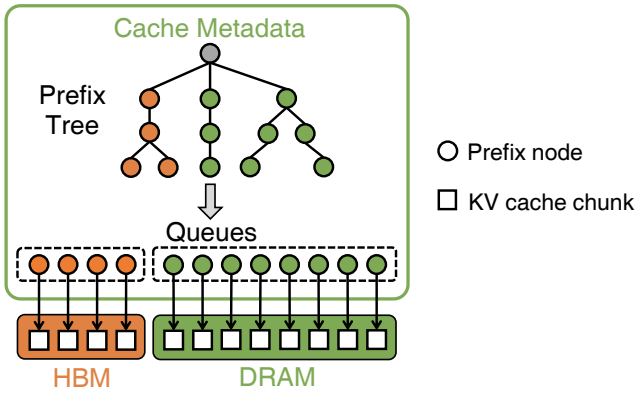


Fig. 1. An illustration of prefix caching. Prefix tree nodes store metadata and point to KV cache chunks, which hold the actual KV tensors in HBM or DRAM. Each chunk corresponds to a fixed number of tokens. Prefix nodes can be logically viewed as being organized into queues ordered by eviction preference for cache replacement.

immediately. This approach allows requests with overlapping prefixes to benefit from the cached KV tensors, reducing redundant prefill computation.

2.2 Rise of Hierarchical Prefix Caching

Reusing prefix KV caches reduces LLM inference costs, but storing KV caches incurs a substantial memory overhead. For example, with float16 precision, the KV cache footprint *per token* is approximately 0.125 MB for Llama-8B [42] and 0.156 MB for Mistral-24B [3]. Consequently, a 16 GB HBM can store about 131k tokens of KV cache for Llama-8B, while a 32 GB HBM can accommodate around 210k tokens for Mistral-24B. In practical deployments, such capacity for cache reuse across different requests is quickly exhausted. In a wide range of practical scenarios, user inputs commonly exceed 1k tokens on average, as prompts often include accumulated dialog history or retrieved documents [17, 37, 55, 56].

Existing LLM systems primarily cache prefix KV caches across HBM and DRAM to extend cache capacity beyond GPU HBM, as illustrated in Figure 1. The prefix lookup structure identifies the matched prefix chunk nodes, while the underlying replacement queues decide their promotion and demotion. However, even with DRAM as an additional cache tier, the overall cache capacity remains limited, and accessing KV caches from DRAM incurs non-trivial transfer latency. Despite these constraints, hierarchical prefix caching has proven effective for LLM inference due to two key factors. First, prefix KV cache access patterns in LLM inference are highly skewed. Second, retrieving KV caches from DRAM is still faster than recomputing the corresponding KV states during prefill.

Skewed access pattern of prefix KV caches. To characterize the access locality of prefix KV caches in practice, we analyze two representative LLM serving workloads: conversation and retrieval-augmented generation (RAG). In conversational workloads, prefix KV caches are naturally reused across multiple turns within the same session. In each conversation turn, a user inputs a new question or command q_j and awaits the response a_j from the LLM. To maintain contextual coherence, the LLM generates the next response a_{N+1} based on both the historical tokens from all previous turns $d_{[1:N]}$ and the input tokens of the current turn, forming the context sequence $q_1a_1q_2a_2...q_Na_Nq_{N+1}$. Assuming that each turn corresponds to a single inference request and that every turn except the first fully leverages all previous conversational history, for a multi-turn

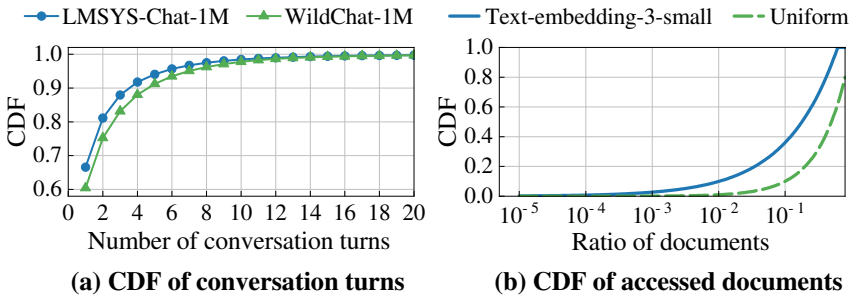


Fig. 2. (a) CDF of the number of conversation turns in LMSYS-Chat-1M and WildChat-1M datasets. (b) CDF of retrieved documents in the HotpotQA dataset.

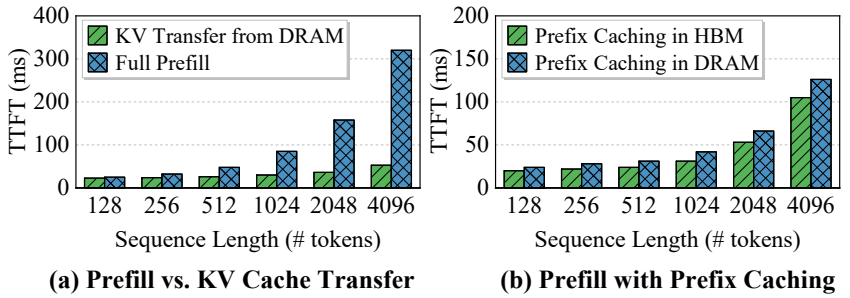


Fig. 3. Impact of hierarchical prefix KV caching on TTFT. (a) Comparison between full prefill TTFT and TTFT with the full prefix KV cache reused from CPU DRAM. (b) TTFT with prefix caching when 80% of prefix KV caches are reused from either HBM or CPU DRAM.

conversation with an average of n turns, the theoretical reuse probability of KV caches can be approximated by $\frac{n-1}{n}$. Based on our analysis of two real conversation datasets, LMSYS-Chat-1M [59] and WildChat-1M [58], we observe that more than 30% and 40% of conversations are multi-turn, respectively, as shown in Figure 2(a), with average turns exceeding two in both datasets. Similarly, RAG workloads exhibit skewed retrieval locality. We analyze the document retrieval pattern of the HotpotQA dataset [52] to quantify this skewness. Specifically, we first convert all Wikipedia documents in HotpotQA into vector representations using the text-embedding-3-small model [33], and then perform retrieval for each query to identify the top-1 most relevant document. Figure 2(b) plots the cumulative distribution function (CDF) of accessed documents across all queries. The result demonstrates a strong retrieval skew: a small subset of documents is frequently retrieved by many queries, while the majority of documents are rarely accessed. For instance, the top 10% of documents are referenced by approximately 40% of all requests. Such skewness suggests that caching frequently accessed prefixes (retrieved contexts) can significantly reduce redundant prefill computation overhead.

Disparity between prefill computation and cache reuse latency. KV cache reuse is generally faster than prefill computation [6, 10, 38, 54]. Figure 3(a) compares the TTFT of full prefill with the TTFT when the full prefix KV cache is reused from CPU DRAM and transferred to GPU HBM for varying context lengths. Experiments are conducted with vLLM [44] and the Llama-8B model [42] at batch size 1 on an NVIDIA A100-40GB GPU connected via PCIe 4.0, which provides a theoretical peak bandwidth of 32 GB/s. Although effective PCIe transfer throughput in practice can be lower due

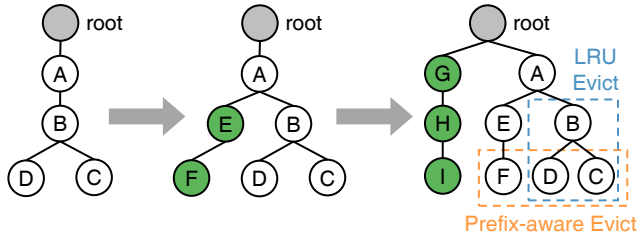


Fig. 4. Example of LRU and prefix-aware eviction for a relative cache size of 6 in prefix caching, where green nodes denote prefixes newly inserted after a cache miss.

to DMA overhead and transfer granularity, reusing KV caches from DRAM still incurs much lower TTFT than recomputing the same prefix through full prefill. Furthermore, we evaluate scenarios where 80% of KV caches are reused for a request, and the remaining 20% are newly computed during prefill computation. As shown in Figure 3(b), the overall TTFT varies depending on whether the reused caches are placed in HBM or DRAM. HBM-resident cache hits consistently achieve lower TTFT than DRAM-resident cache hits because they avoid CPU-to-GPU KV transfer. These results highlight that KV cache reuse significantly improves end-to-end inference performance, and the placement of reused caches across memory tiers also impacts prefill latency.

3 Rethinking Cache Replacement for Prefix Caching

3.1 Prefix-matched Access Characteristics

Prefix caching fundamentally differs from traditional object caching due to its prefix-matched access characteristics. In prefix caching, a cache entry corresponds to a prefix of a request sequence at the chunk granularity, and cache hits occur when multiple requests share common prefixes. As a result, cache accesses are not independent object references, but are strongly constrained by the prefix lookup process: accessing a deeper prefix chunk necessarily implies accessing all of its ancestor prefix chunks.

Notably, prefix lookup structures and cache replacement policies address orthogonal aspects of prefix caching. For instance, vLLM [20] uses hash-based mechanisms to identify prefix cache blocks based on token IDs and prefix hashes, while SGLang [60] employs a radix tree to track shared prefixes. These lookup structures optimize the cache lookup process, i.e., how quickly matched prefix chunks can be identified. In contrast, cache replacement policies determine which prefix chunks are retained under limited cache capacity, thereby affecting cache hit rates and the distribution of hits across memory tiers. In practice, despite using different lookup structures, many existing prefix cache management systems still rely on LRU-based replacement [20, 27, 38, 60].

Several recent works have explored prefix-aware replacement policies, such as PGDSF [17]. We use a simple example to illustrate the differences between LRU and prefix-aware replacement. As illustrated in Figure 4, consider a prefix KV cache that can hold the KV caches for up to six chunks. Assume that the first three requests contain the chunk sequences (A, B, C), (A, B, D), and (A, E, F). After serving these requests, the cache becomes full and the chunks form a prefix tree. When a new chunk sequence (G, H, I) is inserted into the cache, eviction is triggered. Under the LRU policy, the least recently used chunks C, D, and B are evicted to make space for the new chunks G, H, and I. In contrast, the prefix-aware policy tends to evict leaf nodes in the prefix tree, evicting chunks C, D, and F in this example. As a result, compared to LRU, prefix-aware replacement can better preserve shared prefixes.

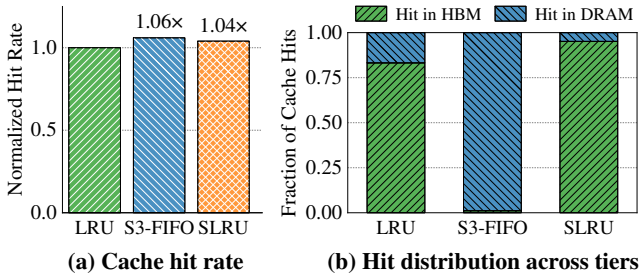


Fig. 5. Overall and per-tier (HBM and DRAM) hit rate comparison among three cache policies.

However, existing prefix-aware cache replacement policies incorporate prefix-related information into eviction priority computation, much like traditional cache replacement policies encode recency, frequency, and other access information into a unified eviction priority. From this perspective, prefix-aware methods can be viewed as extending priority-based replacement with an additional dimension, where cache efficiency largely depends on how well the priority function adapts to the underlying workload. In contrast, for prefix caching, other cache optimization mechanisms, such as cache admission, remain underexplored. Such mechanisms play a critical role in improving cache efficiency, yet existing methods do not leverage prefix information through these mechanisms. This gap motivates us to extend prefix awareness beyond eviction priority.

3.2 Efficiency Benefits of Quick Demotion

As mentioned in §2.2, the access distribution of KV caches is highly skewed. Recent studies [50, 51, 57] suggest that *quick demotion* is an important property of cache replacement policies for improving cache efficiency under skewed workloads. The key intuition behind quick demotion is to remove most objects quickly after they are inserted to prevent useless objects from occupying excessive cache space, reducing cache pollution. A common way to achieve quick demotion is through segmented cache replacement: dividing the cache into a probationary segment and a protected segment. New objects are first inserted into the probationary segment. If they become hot, they are promoted to the protected segment; otherwise, they are promptly evicted.

S3-FIFO [51] and SLRU [19] are two representative segmented cache replacement policies that can easily fit hierarchical prefix caching. Specifically, in S3-FIFO, HBM can be used as the probationary segment while DRAM serves as the protected segment, whereas in SLRU, DRAM is used as the probationary segment and HBM acts as the protected segment. To demonstrate the efficiency benefits of quick demotion, we evaluate S3-FIFO and SLRU across different traces and cache configurations, and measure their average cache hit-rate improvements over LRU. Figure 5(a) shows that S3-FIFO achieves an average hit-rate improvement of 1.06× over LRU, while SLRU improves the hit rate by 1.04× compared to LRU. Furthermore, under our experimental cache configurations, the DRAM capacity is larger than the HBM capacity (as is typically the case in practice). As a result, SLRU has a large probationary segment, which is the primary reason why it is less efficient than S3-FIFO.

Overall, quick demotion offers clear efficiency benefits for prefix caching because many prefixes are unpopular. However, as discussed in the next subsection, improved cache efficiency alone does not fully determine LLM inference performance in hierarchical cache replacement, as relying on the small HBM segment for quick demotion can limit performance gains.

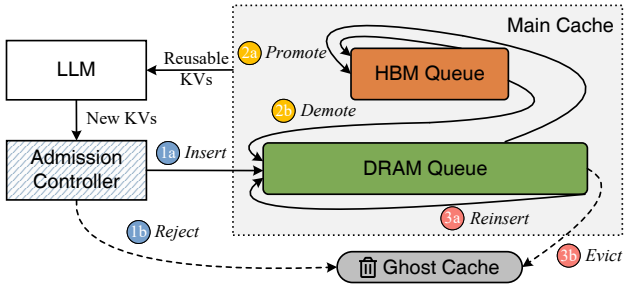


Fig. 6. Overview of Buoy.

3.3 Gap between Efficiency and Performance

Cache efficiency is commonly used as a primary indicator of caching effectiveness. However, in modern LLM inference systems with a hierarchical caching architecture, high cache efficiency does not directly translate into high prefill performance. This section shows that a substantial gap can exist between cache efficiency and LLM performance due to KV cache I/O overhead.

Upon a cache hit, only the KV caches residing in HBM can be directly accessed by the GPU. If the corresponding KV caches reside in DRAM, they must first be transferred to HBM, incurring non-trivial latency along the critical path [10, 25]. As a result, DRAM hits, while avoiding recomputation, provide limited benefit to prefill latency and time-to-first-token (TTFT) as shown in Figure 3(b), whereas HBM hits enable immediate reuse of prefix KV caches and deliver significantly larger performance gains.

Figure 5(b) illustrates the distribution of cache hits across HBM and DRAM for different replacement policies. With S3-FIFO, less than 1% of cache hits occur in HBM, while the majority remain in DRAM. In contrast, LRU and SLRU concentrate most cache hits in HBM. Consequently, despite achieving high overall hit rates, S3-FIFO fails to fully exploit the potential performance benefits of prefix caching due to its DRAM-biased hit locality.

These results suggest that, for prefix caching in multi-tier cache hierarchies, high cache efficiency alone is insufficient and must be complemented by an effective tiering strategy that prioritizes hits in latency-critical tiers.

4 Buoy Design

In this section, we present Buoy, a novel and general cache replacement algorithm tailored for prefix caching. Our primary objective is to improve cache efficiency while ensuring effectiveness.

4.1 Overview

At a high level, as illustrated in Figure 6, Buoy consists of three main components: the main cache, the ghost cache, and the admission controller. The main cache is organized with an HBM queue and a DRAM queue, each storing prefix chunk nodes that reference KV cache chunks physically residing in GPU HBM and CPU DRAM, respectively. The ghost cache keeps track of recently evicted KV caches to make smarter future eviction decisions. The admission controller decides whether newly generated KV caches are retained.

During LLM inference, reusable KV caches can be directly supplied to the LLM, avoiding redundant recomputation during the prefill phase. KV caches residing in HBM can be immediately accessed by the GPU, while KV caches in DRAM are transferred to the GPU before reuse.

Algorithm 1: Hierarchical Cache Replacement in Buoy

```
Input: Prefix chunk nodes C, HBM queue H, DRAM queue M, ghost FIFO queue G
function read(C):
    for x in C do
        if x in H or x in M then                                  ▷ Cache hit
            remove x from H or M                                  ▷ Promote later
            x.target_queue ← H
            x.visited ← True
        else                                                      ▷ Cache miss
            x.target_queue ← M
            if G.exists(x.chunk_id) then
                x.visited ← True
    for x in reversed(C) do                                       ▷ Reverse to preserve prefix order
        insert x to head of x.target_queue                        ▷ Insert or promote
    while H is full do
        evictH()
    while M is full do
        evictM()
function evictH():
    x ← tail of H
    move x to head of M
function evictM():
    x ← tail of M
    if x.visited then
        x.visited ← False
        move x to head of M
    else
        G.add(x.chunk_id)
        remove x from M
```

When the LLM generates new KV caches, the corresponding chunk nodes are examined by the admission controller, which either inserts them into the DRAM queue of the main cache (①a) or directly rejects them (①b). Upon a cache hit, nodes are promoted to the HBM queue (②a). When the HBM queue is full, existing nodes are demoted to the DRAM queue (②b). When nodes in the DRAM queue are selected for eviction, they are reinserted into the main cache if they have been visited (③a); otherwise, they are evicted (③b).

We next introduce the core design of Buoy cache replacement in §4.2 and describe how the admission controller performs cache admission in §4.3.

4.2 Hierarchical Cache Replacement

Below, we detail Buoy 's hierarchical cache replacement for managing limited cache space across the GPU–CPU memory hierarchy.

Ghost cache. Following prior caching designs [18, 51], Buoy introduces a ghost cache to track recently evicted prefix KV cache chunks. Unlike the main cache, the ghost cache does not store KV cache chunks; instead, each ghost cache entry consists only of a chunk ID, which is the hash key used in prefix caching to identify the corresponding KV cache chunk. This design is similar to simulated buffers in database buffer management [41], as both track recently evicted items using

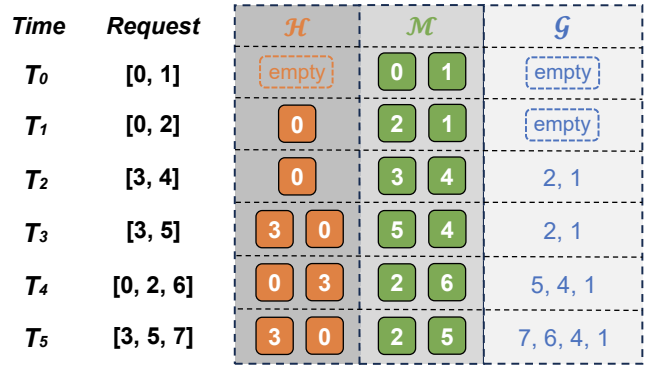


Fig. 7. Example of the cache layout after each request, assuming HBM and DRAM KV cache capacities of two chunks each. Each row corresponds to a request issued at a different timestamp.

lightweight metadata rather than storing the items themselves. The ghost cache is implemented as a FIFO queue in Buoy. When a KV cache chunk is evicted, the chunk ID is inserted into the ghost queue. As each ghost entry consists only of an ID, it incurs minimal additional memory overhead. Moreover, by organizing the ghost cache as a FIFO queue, Buoy preserves temporal information about recent evictions while incurring minimal computational overhead.

Buoy operations. Upon receiving a sequence of prefix chunk nodes for a request, Buoy processes each chunk node by first checking whether it already resides in the HBM queue or the DRAM queue. Each node in the queues uses one bit to track the visited/non-visited status. On a cache hit, the node is moved to the head of the HBM queue, with its visited bit set to true. On a cache miss, the node is inserted into the head of the DRAM queue. To avoid misclassifying recently evicted but reusable prefixes as unpopular, Buoy looks up the ghost cache. If the chunk ID is found in the ghost cache, the visited bit of the node is set to true, allowing it to receive a second chance during future eviction decisions. When the HBM queue exceeds its capacity, the node at the tail is demoted to the head of the DRAM queue. When the DRAM queue becomes full, Buoy evicts nodes from its tail. If the evicted chunk is marked as visited, it is reinserted at the head of the DRAM queue with the bit reset, granting it a second chance to be reused by future accesses. Otherwise, the chunk node is finally evicted from the main cache and the chunk ID is added to the ghost cache.

Algorithm 1 summarizes the replacement procedure. Notably, since each cache access involves a sequence of prefix chunk nodes for a single request, cache reads naturally follow a path from the root to the leaf in the prefix tree. To preserve the prefix order within the queues, cache hits temporarily remove the accessed nodes from their current positions (line 4), and Buoy inserts all nodes into their target queues in reverse order at the end (line 11–12). This reverse insertion places the accessed nodes in ancestor-to-descendant order within each target queue, so that deeper nodes along the accessed path are closer to the queue tail than their shallower ancestors. To further illustrate the replacement procedure, Figure 7 presents a concrete example of Buoy processing a sequence of requests.

Quick demotion. As discussed in §3.2, quick demotion is a key idea for improving cache efficiency in prefix caching workloads. Buoy achieves quick demotion through its hierarchical cache replacement design in two complementary ways. First, newly admitted chunk nodes are inserted into the DRAM queue, and their corresponding KV caches are offloaded to DRAM. Such offloading is asynchronous and not on the user-facing critical path. The user's request can be completed without waiting for this transfer to finish, so offloading has little effect on TTFT. After serving the request,

newly generated KV chunks are not kept in HBM, thereby avoiding displacing more popular chunks already retained in HBM. Second, Buoy employs the visited bit to provide a second-chance mechanism in the DRAM queue, similar to CLOCK. For a node marked as visited to be evicted, it is reinserted at the head instead of being immediately evicted, with its visited bit reset. This mechanism passively accelerates the demotion of other unpopular ones.

Effective cache tiering. Since HBM hits can be reused directly by the GPU while DRAM hits require host-to-GPU transfer before serving, Buoy seeks to reduce overall access latency by maximizing the HBM hit ratio. The HBM queue retains only recently accessed hot nodes, while the DRAM queue primarily contains one-hit wonders [57] or relatively cold nodes that have not been accessed recently. Buoy promotes a node from DRAM to HBM only upon a cache hit, when the corresponding KV cache chunk must already be transferred to the GPU for request serving, thereby avoiding unnecessary data movement. As a result, cache replacement decisions do not introduce additional host-to-GPU transfer overhead beyond what is required for serving the request itself.

4.3 Depth-aware Cache Admission

Cache admission has been recognized as an effective mechanism for improving cache efficiency by preventing low-value objects from entering the cache. Unlike eviction, which decides what to remove when the cache is full, admission determines what to admit upon a cache miss. Representative designs such as TinyLFU [8] demonstrate that admitting objects based on estimated reuse frequency can significantly reduce cache pollution and improve hit rates.

Existing cache admission methods typically treat each admitted object independently. In contrast, Buoy exploits a property unique to prefix caching: cache candidates naturally form prefix paths whose reuse behavior is correlated with depth. As mentioned in §3.1, prefix caching exhibits prefix-matched access characteristics. Shallow prefixes are more likely to be shared across requests, while deeper prefixes tend to be request-specific and exhibit lower reuse probability. Consequently, deep prefixes may occupy valuable cache space at the expense of more reusable shallow prefixes. These observations motivate a cache admission method that is aware of prefix depth and adapts to workload-specific reuse behavior.

Depth-aware cache admission is designed around three key principles: (1) Depth-aware selectivity. Prefixes at larger depths should be admitted more conservatively than shallow prefixes, as their likelihood of reuse decreases with depth. (2) Reuse-informed thresholding. The boundary between shallow and deep prefixes should not be predefined; instead, it should be estimated from the observed depths of prefix reuse. (3) Reuse-proven override. Prefixes that have demonstrated reuse in the past should not be penalized due to their depth, even if they are deep.

Following these design principles, we propose a lightweight and practical admission method. First, a depth threshold is maintained to distinguish shallow prefixes from deep ones. Prefixes whose depth does not exceed this threshold are admitted by default, while deeper prefixes are rejected by default. This selective admission prevents a single long request from filling the cache with its entire request-specific suffix, allowing the available capacity to cover a broader set of prefix paths. Second, to avoid rejecting reusable deep prefixes, the admission method also leverages the ghost cache. Prefixes rejected due to the depth threshold are inserted into the ghost cache. If a deep prefix reappears and triggers a ghost hit, indicating that it was recently rejected or evicted, it is immediately admitted regardless of its depth. Third, the depth threshold is estimated online using the cumulative average of observed reuse depths. We focus on reuse depths rather than averaging over all accessed prefixes because reuse events directly reflect which prefix depths are valuable to retain in the cache. Including non-reused deep prefixes would bias the estimate toward larger depths dominated by one-hit accesses, leading to overly permissive admission and increased cache pollution. Each reuse event contributes its prefix depth to the estimator, which updates

the cumulative mean in $O(1)$ time. The resulting average reuse depth determines the admission threshold, allowing the method to tailor its admission decisions to the reuse-depth distribution of the observed workload without offline profiling.

By combining depth-based selectivity with reuse-informed thresholding, depth-aware cache admission improves cache efficiency in Buoy by reducing cache pollution while preserving reusable prefixes. Moreover, it operates independently of the underlying cache replacement policy and can in principle be integrated with other cache replacement policies.

4.4 Summary

Buoy achieves its design goals through three key measures. (1) Buoy performs quick demotion to prevent unpopular KV caches from occupying cache space for a long time. Specifically, newly generated KV caches are first inserted into the DRAM queue and are promptly evicted if not reused. In addition, a second-chance mechanism further accelerates the demotion of unpopular KV caches. Finally, based on prefix-matched access characteristics, cache admission enforces more aggressive quick demotion by rejecting deep prefix nodes. (2) Buoy employs a ghost cache to allow recently evicted KV caches to be retained longer when the prefixes are requested again, further improving cache efficiency. (3) By ensuring that unpopular KV caches are never admitted to the HBM queue, Buoy maximizes the fraction of cache hits served from HBM, thereby reducing host-to-GPU transfer overhead on cache hits. Taken together, these strategies make Buoy an efficient and effective cache replacement policy applicable to prefix caching in general.

5 Evaluation

5.1 Implementation

We first implement a prefix KV cache simulator to evaluate and compare different cache replacement policies for prefix caching in LLM inference. The simulator replays request traces and models prefix-matched accesses at the granularity of KV cache chunks. Each request is decomposed into a sequence of prefix chunks, and the simulator tracks the storage footprint of KV caches using model-specific per-token KV sizes, which are deterministic and can be computed analytically for a given LLM and numerical precision. All policies can be evaluated under identical traces, cache capacities, and memory hierarchy configurations, ensuring fair and reproducible comparison. For each request, the simulator invokes prefix accesses in sequence and records detailed cache statistics to compare cache efficiency, including hit rates at different tiers.

To validate the practical effectiveness of cache replacement policies beyond simulation, we further build a serving engine implementation based on `vLLM` [44] v0.15.1. Our implementation extends vLLM's prefix KV cache management with configurable replacement policies. It enables standardized end-to-end prefill performance comparisons under a unified execution framework.

5.2 Experiment Methodology

Models and Hardware. We use Llama-8B [42] and Mistral-24B [3]. Experiments are conducted on two servers: one equipped with a 40 GB A100 GPU for running Llama-8B, and the other equipped with an 80 GB A800 GPU for running Mistral-24B. Each server is allocated 64 GB of CPU DRAM as the DRAM tier of the prefix KV cache, and the GPU is connected to the CPU via a PCIe 4.0 x16 link.
Workloads. Our experiments use open-source request traces from Aliyun Bailian [45] and Kimi [38]. We list the dataset information in Table 1. We primarily use Qwen Trace A and Qwen Trace B from Aliyun Bailian, as they contain a larger number of requests. We replay these traces sequentially, sending a new request only after the previous request finishes. This setup is practical given our limited GPU resources and allows us to focus on the impact of cache replacement policies.

Table 1. Real-world traces used in our evaluation.

Trace	Chunk Size	# Req.	Description
Qwen Trace A [45]	16	43,058	Two-hour trace of chat services (to-C), sampled from Aliyun Bailian.
Qwen Trace B [45]	16	172,800	Two-hour trace of API calling (to-B), sampled from Aliyun Bailian.
Mooncake Conversation [38]	512	12,031	One-hour trace of online conversations, sampled from Kimi.
Mooncake Tool&Agent [38]	512	23,608	One-hour trace of tool&agent interactions, sampled from Kimi.

Metrics. We adopt the following metrics:
- Cache hit rate. The average cache hit rate across all requests.
- Prefill throughput. The average number of input tokens served per second during the prefill phase, counting both cache-hit tokens and newly computed tokens.
- Time-to-first-token (TTFT). The latency from request arrival to the generation of the first output token. In the vLLM-based end-to-end evaluation, we report both mean TTFT and tail TTFT, measured as the P90, P95, and P99 of per-request TTFT.

Baselines. We compare Buoy against traditional cache replacement algorithms (FIFO, LRU, and LFU) and the following policies:
- SLRU. An LRU policy viewed as two logical segments. New KV cache nodes are first inserted into the DRAM queue and promoted to the HBM queue upon cache hits. Nodes evicted from the HBM queue are reinserted to the head of the DRAM queue.
- S3-FIFO [51]. A FIFO-based replacement policy with static queues. Nodes evicted from the HBM FIFO queue are either inserted into the DRAM FIFO queue or undergo quick demotion based on their access frequency. Nodes at the tail of the DRAM FIFO queue are either reinserted to the head of the DRAM queue or evicted based on access frequency. On a ghost hit or an external-storage hit, the node is thereafter retained in DRAM.
- Prefix-aware Greedy-Dual-Size-Frequency (PGDSF) [17]. A cache replacement policy that considers prefix position, frequency, size, and recency to decide node placement in hierarchical caches.
- Workload-aware policy (WA) [45]. A workload-aware cache replacement policy that computes a two-dimensional priority for each KV cache node, defined as *(ReuseProb, -Offset)*, where *ReuseProb* estimates the node's future reuse probability based on historical workloads, and *Offset* is reversely proportional to the prefix length of the KV cache node in the entire request.
- HotPrefix [25]. A hotness-aware policy that tracks and updates the hotness of prefix tree nodes over time to retain hot prefixes.

5.3 Efficiency

We first evaluate cache efficiency using the prefix KV cache simulator. For Llama-8B, we configure cache sizes of 16 GB HBM and 64 GB DRAM, while for Mistral-24B, we use cache sizes of 32 GB HBM and 64 GB DRAM. The simulator models prefix-matched KV cache accesses at the chunk level and reports cache hit rates at different memory tiers.

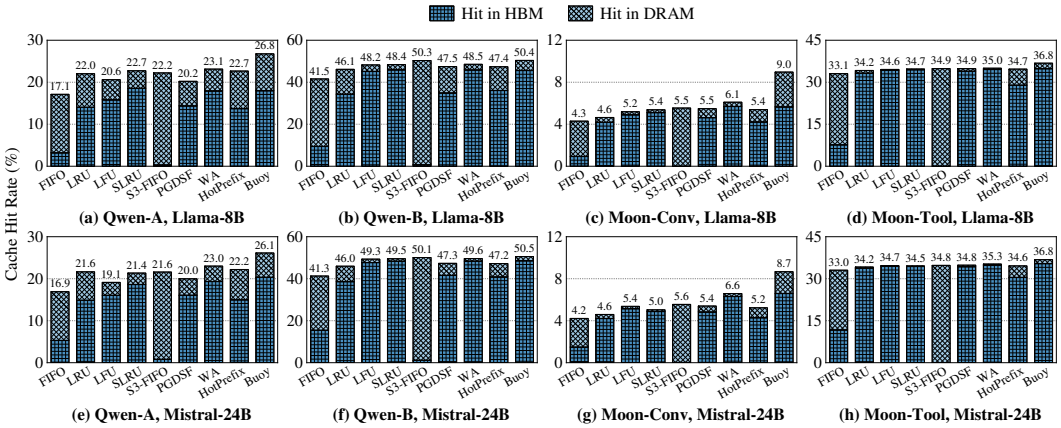


Fig. 8. Cache hit rates of different algorithms in the simulator across four traces for Llama-8B on the A100 GPU server and Mistral-24B on the A800 GPU server, showing the hit rate breakdown.

Figure 8 summarizes the cache hit rates of different algorithms. Buoy achieves the highest overall cache hit rate across all traces and both models. For example, on the Qwen-A trace with Llama-8B, Buoy achieves a cache hit rate of 26.8%, compared to LRU's 22.0%, corresponding to a 1.22× improvement. In contrast, the next best policy only reaches a hit rate of 23.1%, providing a marginal 1.05× improvement over LRU. This improvement is even more pronounced on the Moon-Conv trace, where Buoy achieves a cache hit rate that is nearly twice that of LRU. Other algorithms (PGDSF, WA, and HotPrefix) specifically designed for prefix caching generally outperform LRU as well. However, they do not always surpass the traditional policies SLRU and S3-FIFO. The good performance of SLRU and S3-FIFO corroborates our insight that quick demotion is critical for cache efficiency.

We further observe that cache efficiency varies significantly across different workload types. In particular, conversational workloads (Qwen-A and Moon-Conv) consistently exhibit much lower overall hit rates than API (Qwen-B and Moon-Tool) workloads. This is because conversational requests typically involve more diverse and less repetitive prefix paths, limiting the upper bound of achievable cache reuse. In contrast, API workloads exhibit stronger prefix reuse, allowing most policies to achieve relatively high hit rates, which in turn narrows the performance gap among different algorithms.

Beyond overall hit rates, Figure 8 also reveals substantial differences in how cache hits are distributed across memory tiers. While several policies achieve similar overall hit rates, their HBM and DRAM hit rate breakdowns differ markedly. Buoy consistently achieves the highest or near-highest HBM hit rate across all traces. This behavior is a direct result of Buoy 's hierarchical cache replacement design, which preferentially promotes reusable prefixes to HBM while aggressively demoting cold prefixes. In contrast, S3-FIFO attains high overall hit rates primarily by accumulating hits in DRAM, with extremely low HBM hit rates. Although such DRAM hits improve cache efficiency, they provide limited benefit for prefill latency, as the corresponding KV caches must still be transferred to the GPU upon reuse.

Moreover, the relative performance trends remain consistent across model scales. Moving from Llama-8B to the larger Mistral-24B model increases the footprint of prefix KV caches and further tightens the HBM capacity constraint. Nevertheless, Buoy maintains its advantage over all baselines

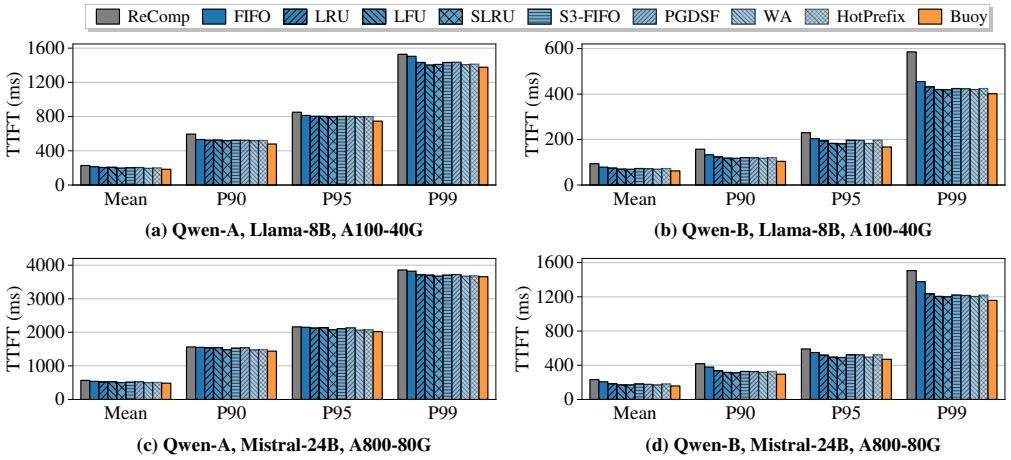


Fig. 9. TTFT under different cache replacement policies on vLLM. Mean, P90, P95, and P99 TTFT are reported on Qwen-A and Qwen-B with Llama-8B and Mistral-24B.

in both overall hit rate and HBM hit rate. This suggests that Buoy scales well with model size and remains effective even when cache capacity becomes increasingly constrained.

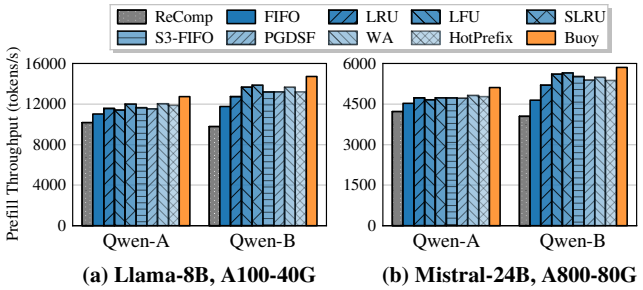
Overall, these results indicate that high cache efficiency for prefix caching is not solely determined by prefix-awareness or overall hit rate. Instead, it critically depends on a cache replacement policy's ability to aggressively demote cold prefixes while ensuring that reusable prefixes are retained in fast memory.

5.4 Prefill Performance

We evaluate the prefill performance of different cache replacement policies under realistic workloads using our vLLM-based serving-engine implementation. Our evaluation focuses on how effectively each policy improves end-to-end prefill performance by reusing prefix KV caches across requests. We follow the methodology proposed in prior work [45] to evaluate serving performance on anonymous traces. In our experiments, GPU memory is used by both the LLM and prefix KV caching. We set the GPU memory utilization ratio to 0.9, allowing vLLM to use up to 90% of GPU memory for model execution and KV cache management. All policies are evaluated under identical model, workload, cache capacity, and serving configurations. In addition to comparing different cache replacement policies, we compare with the recomputation baseline (ReComp). ReComp computes the full KV cache for all input tokens during prefill, without caching or reusing KV caches across requests. This baseline quantifies the performance benefit brought by prefix caching and serves as a lower bound on prefill performance.

Figure 9 presents the TTFT results, including mean, p90, p95, and p99 latency. Prefix caching substantially reduces TTFT relative to ReComp, and Buoy achieves the lowest mean TTFT across all evaluated configurations. For Llama-8B, Buoy reduces mean TTFT by 1.24× on Qwen-A and 1.52× on Qwen-B compared to ReComp. Compared with LRU, Buoy reduces mean TTFT by 1.11× on Qwen-A and 1.19× on Qwen-B. For Mistral-24B, Buoy reduces mean TTFT by 1.16× on Qwen-A and 1.46× on Qwen-B over ReComp, and by 1.07× and 1.16× over LRU, respectively.

Beyond mean latency, Buoy also improves tail TTFT. Compared with LRU, Buoy reduces p90/p95/p99 TTFT by 8.2%/7.0%/4.0% on Llama-8B with Qwen-A and by 15.4%/13.9%/7.2% with Qwen-B. For Mistral-24B, Buoy reduces p90/p95/p99 TTFT by 6.2%/5.1%/1.5% on Qwen-A and by



(a) Llama-8B, A100-40G **(b) Mistral-24B, A800-80G**

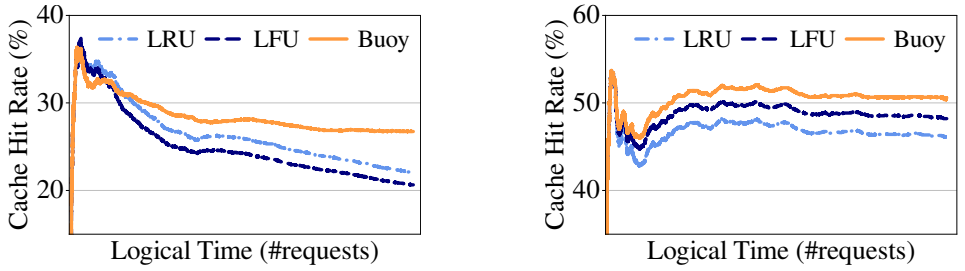
Fig. 10. Prefill throughput under different cache replacement policies on vLLM. (a) Llama-8B running on the A100-40GB GPU. (b) Mistral-24B running on the A800-80GB GPU.

11.6%/9.3%/6.3% on Qwen-B. The tail-latency improvement is especially pronounced on Qwen-B, where higher prefix reuse creates more opportunities for effective KV cache retention and tier-aware replacement.

Figure 10 reports the prefill throughput under different cache replacement policies. Consistent with the TTFT results, prefix caching improves prefill throughput compared to ReComp under all evaluated replacement policies. For Llama-8B, Buoy improves prefill throughput by 1.25× on Qwen-A and 1.51× on Qwen-B over ReComp. Compared with LRU, Buoy improves throughput by 1.10× on Qwen-A and 1.16× on Qwen-B. For Mistral-24B, Buoy improves prefill throughput by 1.21× on Qwen-A and 1.45× on Qwen-B over ReComp, and by 1.08× and 1.13× over LRU, respectively. These results show that the cache efficiency advantages of Buoy translate into higher end-to-end prefill throughput in a real serving engine.

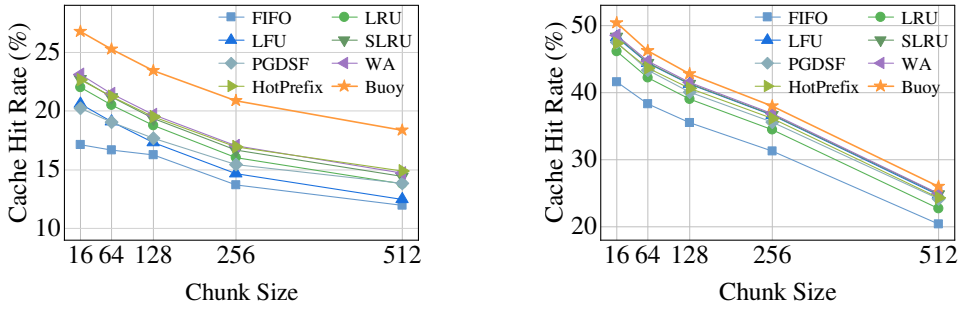
Notably, policies with similar cache hit rates can still exhibit different prefill performance because hits at different tiers have different latency costs. For example, S3-FIFO may increase overall cache hits but place more reusable KV chunks in slower tiers, leading to limited TTFT improvement. A DRAM hit avoids recomputation, but the reused KV cache must still be reloaded into HBM before prefill can proceed. This H2D transfer lies on the TTFT critical path and can also reduce throughput by consuming PCIe bandwidth. Prior work [27, 46] has shown that PCIe transfer throughput is sensitive to transfer granularity: small transfers achieve low effective bandwidth, whereas MB-scale transfers can utilize the PCIe link more efficiently. In our vLLM-based evaluation, each 16-token KV page corresponds to an MB-scale physical KV block, about 2 MB for Llama-8B and 2.5 MB for Mistral-24B. On our PCIe 4.0 servers, such MB-scale transfers achieve an effective single-direction bandwidth of about 20 GB/s. For example, reusing 4096 DRAM-resident tokens for Llama-8B requires transferring about 0.5 GiB of KV cache data, which takes roughly 25 ms at 20 GB/s. This overhead is non-negligible and lies on the TTFT critical path.

We further analyze the overheads introduced by prefix caching and cache replacement. For prefix lookup, both LRU and Buoy take only a few microseconds per request on average, which is negligible compared with end-to-end TTFT. For replacement bookkeeping, LRU takes 0.4 ms/request on average, while Buoy takes 2 ms/request. The higher bookkeeping cost of Buoy comes from its additional metadata operations, including maintaining tiered queues, updating the visited flag, performing ghost-cache lookup/update, and applying depth-aware admission. Nevertheless, this overhead is much smaller than the TTFT reduction obtained by increasing HBM hits and reducing critical-path H2D transfers. Overall, the TTFT and throughput results reflect the net effect of cache reuse benefits, cache bookkeeping overhead, and DRAM-to-HBM transfer overhead. Buoy consistently outperforms prior cache replacement policies in both TTFT and throughput, demonstrating its practical effectiveness in improving LLM prefill performance.



(a) Qwen-A (b) Qwen-B

Fig. 11. Cache hit rate over time for two traces. The gaps between Buoy's hit rate and others enlarge over time.



(a) Qwen-A (b) Qwen-B

Fig. 12. Large chunk sizes decrease cache hit rate.

5.5 Robustness Analysis

We now analyze the robustness of Buoy along multiple dimensions, including temporal behavior, chunk granularity, and cache size. All results in this analysis are obtained from the KV cache simulator.

Efficiency over time. Figure 11(a) and Figure 11(b) show the cumulative cache hit rate over time on the Qwen-A and Qwen-B traces. After an initial warm-up phase, the gap between Buoy and other algorithms steadily widens. This trend indicates that Buoy indeed sifts out unpopular prefixes and retains reusable prefixes over time.

Chunk granularity. In LLM inference systems, cache granularity plays a critical role in cache hit rates. Existing systems commonly adopt chunk-based caching by grouping tokens into chunks [10, 20, 38], each typically containing 16 to 512 tokens. Smaller chunk sizes allow for more precise cache matching, whereas larger chunk sizes reduce matching effectiveness, leading to higher miss rates and longer latencies. In our evaluation, the Qwen-A and Qwen-B traces are recorded with a chunk size of 16 tokens, while the Moon-Conv and Moon-Tool traces are recorded with 512 tokens per chunk by default. To study the impact of chunk granularity, we derive Qwen-A and Qwen-B traces with larger chunk sizes by merging consecutive 16-token chunks into coarser chunks, while preserving the original prefix access order. Figures 12(a) and 12(b) illustrate the impact of chunk size on cache hit rates for the Qwen-A and Qwen-B traces, where chunk sizes are varied from 16 to 512 tokens. Cache hit rates decrease as chunk size increases for all cache replacement algorithms.

Table 2. Approximate total size of prefill KV caches generated by two LLMs after prefix deduplication on different traces.

Trace	Llama-8B	Mistral-24B
Qwen-A	4942 GB	7721 GB
Qwen-B	7481 GB	11689 GB
Moon-Conv	11424 GB	17850 GB
Moon-Tool	11456 GB	17900 GB

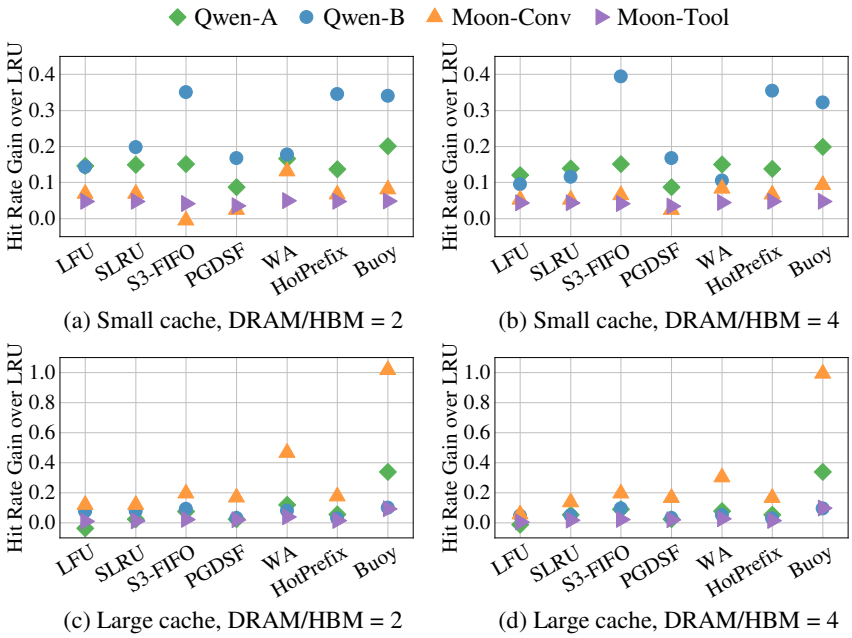


Fig. 13. Hit rate gain over LRU on different traces, where DRAM/HBM denotes the ratio of DRAM cache capacity to HBM cache capacity.

Nevertheless, Buoy achieves higher cache hit rates than other algorithms across all chunk sizes. Moreover, Buoy exhibits the smallest performance degradation as chunk size increases.

Cache size. We further study the robustness of Buoy under different cache size settings. Table 2 reports the approximate total size of prefix-deduplicated prefill KV caches generated by each trace for the two models, which ranges from several terabytes to tens of terabytes. Based on these statistics, we define two cache regimes: a small cache size corresponding to 0.1% of the total KV cache footprint and a large cache size corresponding to 1%. For each regime, we additionally vary the DRAM-to-HBM cache capacity ratio (DRAM/HBM) between 2 and 4. Figure 13 shows the cache hit rate gain over LRU under different cache size configurations. Across all traces and cache regimes, Buoy consistently outperforms LRU. In most cases, Buoy also achieves the largest performance gains among all policies. We observe only a few exceptions under small cache sizes: on the Qwen-B trace, Buoy performs comparably to or slightly below S3-FIFO and HotPrefix, and on the Moon-Conv trace, it performs comparably to or slightly below WA. Nevertheless, in all

Table 3. Cache hit rate breakdown under a three-tier cache hierarchy (16 GB HBM, 64 GB DRAM, 512 GB SSD) for different cache replacement policies with Llama-8B.

		Cache Hit Rate (%)			
		HBM	**DRAM**	**SSD**	**Overall**
Qwen-A	FIFO	0.71	8.34	32.52	41.57
	LRU	14.04	7.99	29.33	51.37
	LFU	16.66	4.97	29.74	51.37
	SLRU	15.23	7.29	28.98	51.50
	S3-FIFO	0.50	23.23	27.79	51.53
	PGDSF	13.71	5.25	31.91	50.87
	WA	18.01	5.20	28.25	51.45
	HotPrefix	13.77	11.61	26.18	51.56
	Buoy	18.93	4.56	28.01	51.50
Qwen-B	FIFO	1.99	6.32	41.28	49.59
	LRU	34.36	11.78	5.57	51.71
	LFU	45.56	2.93	3.26	51.75
	SLRU	44.99	3.24	3.56	51.79
	S3-FIFO	0.65	50.99	0.81	52.45
	PGDSF	33.70	13.46	4.75	51.92
	WA	45.89	2.64	3.25	51.78
	HotPrefix	36.46	11.45	3.88	51.78
	Buoy	45.94	3.21	2.70	51.85

remaining settings, Buoy outperforms all other algorithms by a clear margin, demonstrating strong robustness across a wide range of cache capacities and memory hierarchy configurations.

5.6 Extending to External Storage

Increasingly, LLM serving systems [46] extend prefix KV caching to external storage such as SSDs to increase cache capacity and hit rates. In this section, we evaluate the effect of external storage from two perspectives. We first use our simulator to analyze the cache hit-rate breakdown across HBM, DRAM, and SSD. We then extend our vLLM-based implementation with SSD-tier KV offloading and evaluate end-to-end prefill throughput and TTFT.

With its large capacity, external storage can expand the total cache capacity, allowing more prefix KV caches to be retained and reducing complete evictions from the cache hierarchy. As a result, overall cache hit rates tend to increase, while the performance gap among different cache replacement policies tends to narrow because cached KV chunks are less likely to be evicted from the entire cache hierarchy.

However, this capacity expansion shifts the performance emphasis from overall cache hit rate to hit distribution across tiers and cross-tier data movement costs. In particular, KV caches stored on SSD must be transferred back to the GPU before reuse. In our implementation, upon an SSD hit, the corresponding KV cache is first loaded into DRAM and then transferred to GPU HBM, incurring both SSD I/O and host-to-GPU transfer latency. Consequently, although external storage alleviates capacity constraints and boosts overall hit rates, it also introduces non-trivial data movement overheads that diminish the end-to-end performance benefits of cache hits. In short, external storage alleviates capacity limitations, but simultaneously exacerbates the gap between cache efficiency and actual LLM inference performance.

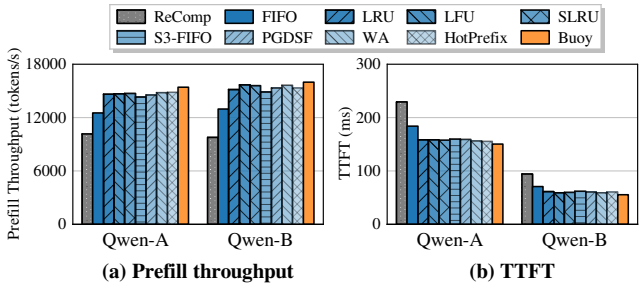


Fig. 14. Prefill performance on Qwen-A and Qwen-B under a three-tier hierarchy with KV cache capacities of 16 GB in HBM, 64 GB in DRAM, and 512 GB in SSD for different cache replacement policies with Llama-8B: (a) prefill throughput and (b) TTFT.

To study this trade-off, we extend the original two-tier KV cache hierarchy to a three-tier design with SSD. Under this design, KV chunks evicted from DRAM are demoted to SSD. For Buoy, chunks rejected by the admission controller are also inserted into SSD because the SSD tier replaces its metadata-only ghost cache.

KV chunks placed in SSD are generally colder and exhibit weaker temporal locality than those retained in HBM or DRAM. Applying a fine-grained eviction policy at the external storage tier therefore provides limited additional efficiency benefits while incurring substantial metadata maintenance overhead at scale. Instead, we adopt a simple FIFO policy for managing the SSD tier, which is lightweight and sufficient for capturing occasional long-term reuse. Upon an SSD hit, the chunk is removed from the SSD FIFO queue and promoted to the HBM queue after being staged through DRAM. Thus, FIFO determines only the eviction order among chunks that remain resident in SSD. For fairness, all baseline policies use the same SSD-tier FIFO eviction and hit-promotion behavior.

Table 3 reports the cache hit rate breakdown across HBM, DRAM, and SSD after extending the cache hierarchy with a 512 GB SSD. Introducing external storage increases the overall cache hit rate for all cache replacement policies. In particular, the overall hit rates become highly similar across policies, indicating that the large SSD capacity effectively absorbs most evicted KV caches and reduces capacity-induced misses. Despite comparable overall hit rates, however, the distribution of cache hits across tiers differs significantly. Buoy consistently increases the fraction of cache hits served from the highest tier (HBM), while reducing the fraction of hits occurring in lower tiers such as DRAM and SSD. This shift is critical, since only HBM-resident KV caches can be reused without incurring cross-tier data transfer overhead. As a result, a larger fraction of reused KV chunks can be served with minimal access latency, even though the overall cache hit rates remain similar.

Figure 14 reports end-to-end prefill performance under the three-tier cache hierarchy. Across both Qwen-A and Qwen-B traces, all cache-based approaches outperform recomputation, confirming the effectiveness of prefix caching even when KV caches are partially stored on SSD. On Qwen-A, Buoy achieves 15,210 tokens/s and 152 ms TTFT, improving throughput by 1.50× and reducing TTFT by 1.50× compared with ReComp. On Qwen-B, Buoy achieves 15,807 tokens/s and 58 ms TTFT, improving throughput by 1.62× and reducing TTFT by 1.63× compared with ReComp.

Among all cache replacement policies, Buoy consistently achieves the highest prefill throughput and the lowest TTFT. On Qwen-A, Buoy improves throughput by 2.3% over the strongest baseline and reduces TTFT by 2.2%. On Qwen-B, Buoy improves throughput by 0.7% over the strongest baseline and reduces TTFT by 2.7%. Although the performance gap becomes smaller than in the

Table 4. Mean TTFT under distributed serving on vLLM with Llama-8B using two A100-40GB GPUs.

Deployment	Trace	LRU	Buoy
Tensor-parallel (TP)	Qwen-A	131 ms	117 ms
	Qwen-B	52 ms	45 ms
Data-parallel (DP)	Qwen-A	200 ms	180 ms
	Qwen-B	73 ms	61 ms

two-tier setting because the large SSD capacity narrows the differences in overall hit rate, Buoy still provides the best end-to-end performance by increasing the fraction of hits served from HBM.

These results highlight that, under a three-tier cache hierarchy, cache replacement policies must prioritize effective HBM residency for reusable prefixes in order to translate cache efficiency into tangible performance benefits. By biasing cache hits toward the highest tier and reducing costly DRAM- and SSD-to-GPU transfers, Buoy remains effective even when cache capacity is expanded through external storage.

5.7 Applicability to Distributed Serving

We further evaluate whether Buoy remains effective under distributed serving in vLLM. Experiments are conducted on a server with two A100-40GB GPUs connected by NVLink. The traces are replayed sequentially, following the same methodology as our end-to-end evaluation. Table 4 reports the mean TTFT under two deployment modes: tensor parallelism (TP), where a request is executed by a group of GPUs that jointly host one model replica, and data parallelism (DP), where requests are served by replicated workers with separate cache instances. Across both TP and DP, Buoy consistently reduces mean TTFT compared with LRU on both traces.

The benefit of Buoy carries over to distributed serving because Buoy operates on prefix KV cache replacement and does not depend on a specific execution strategy. In TP, Buoy can be applied within each TP group by managing the cached prefix as a logical cache object across participating GPUs. In DP, Buoy can be deployed independently at each replica, while affinity-aware routing can improve prefix locality.

5.8 Ablation Studies

We explore the effects of key design choices in Buoy via ablation studies conducted on the KV cache simulator based on Llama-8B with 16 GB HBM and 64 GB DRAM.

Impact of cache admission. We first evaluate the impact of cache admission by disabling the admission control mechanism in Buoy, allowing all missed prefix chunks to be directly inserted into the cache. As shown in Figure 15, enabling cache admission improves cache hit rates across all traces. The improvement is particularly pronounced on the conversation trace, where cache capacity is highly constrained and a large fraction of prefixes exhibit weak temporal reuse. This result confirms that cache admission is critical for filtering out unpopular prefixes early and preventing them from polluting limited cache space.

Impact of second chance. We next study the effect of the second-chance mechanism by disabling the visited bit and reinsertion logic in the DRAM queue, effectively turning the DRAM queue from a CLOCK-style policy into a pure FIFO queue. As shown in Figure 15, enabling the second-chance mechanism yields consistent but relatively modest improvements across all traces. This indicates that second chance helps recover from occasional misclassification of reusable prefixes by granting them an additional opportunity before eviction. However, its impact is secondary, suggesting that

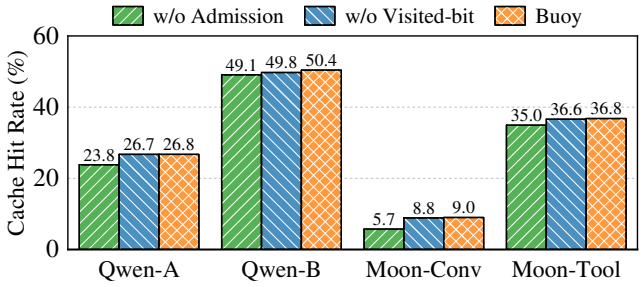


Fig. 15. Impact of cache admission and the second-chance (visited-bit) mechanism on cache hit rate.

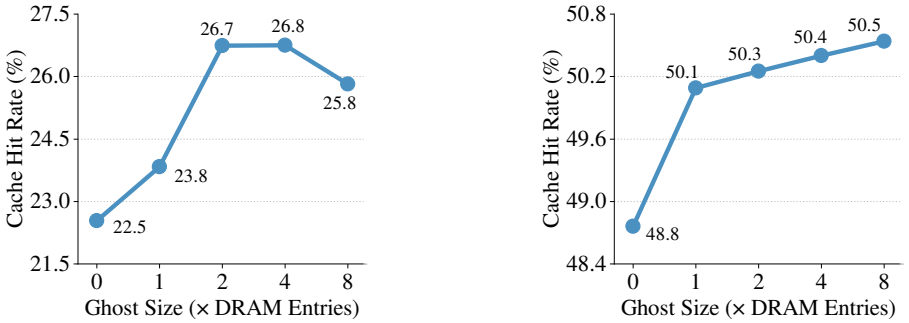


Fig. 16. Impact of ghost cache size for Buoy on the Qwen-A and Qwen-B traces.

second chance primarily serves as a complementary refinement rather than the dominant factor in improving cache efficiency.

Impact of ghost cache. We further study the impact of the ghost cache size in Buoy. By default, the ghost cache stores 4× as many entries as the DRAM queue, while each entry only records a chunk ID without storing KV data. We vary the ghost cache size from 0 to 8× the DRAM queue size and evaluate its effect on cache hit rates, as shown in Figure 16. Enabling the ghost cache obviously improves cache efficiency compared to having no ghost cache. On the Qwen-A trace, increasing the ghost cache size initially leads to notable gains, as it allows recently evicted but reusable prefixes to be accurately identified upon re-access. Beyond 4× the DRAM queue size, the benefit of further enlarging the ghost cache gradually diminishes. On the Qwen-B trace, increasing the ghost cache size continues to provide improvements, but the gains are relatively small. Based on these observations, we adopt 4× the DRAM queue size as the default ghost cache configuration, which achieves a good balance between effectiveness and overhead.

6 Related Work

Cache replacement. Cache replacement has been extensively studied for decades, with the primary goal of improving cache efficiency. For example, various LRU variants (such as LRU-K [32], SLRU [19], TwoQ [18], EELRU [40], LIRS [15], ARC [31], and CAR [4]) employ different definitions of recency to select eviction candidates. Some algorithms (such as LRFU [21], TinyLFU [8], FRD [35], S3FIFO [51], and Hill-Cache [26]) incorporate both recency and frequency to score objects using different methods, prioritizing them to reduce cache miss ratios. Several studies (such as LeCaR [43], LHD [5], CACHEUS [39], GL-Cache [49], and 3L-Cache [62]) have explored using machine learning

to improve eviction strategies. SIEVE [57] has recently been proposed as a cache eviction algorithm that is simpler than LRU yet more effective, using a FIFO queue with a moving pointer to retain popular objects in place and quickly remove unpopular ones. However, these algorithms are designed for traditional caching units such as objects, in-memory key-value pairs, or blocks, and do not account for the prefix-matched access characteristics of KV caches in LLM inference. Recently, several studies on cache replacement for prefix caching have also been proposed [17, 25, 45], demonstrating performance improvements for LLM inference. Nevertheless, the effectiveness of these policies varies significantly across workloads, as different designs favor different access patterns, and they are relatively complex to implement. In contrast, Buoy is a simple and general cache replacement design that achieves robust performance across diverse LLM workloads. Future work can explore learned eviction policies for prefix caching, enabling more precise eviction decisions.

Prefix caching. Storing and reusing prefix KV caches has emerged as a key technique for improving the throughput and latency of LLM serving. One representative approach is paged memory management in GPU, adopted in vLLM [20] and SGLang [60]. However, the limited capacity of GPU memory leads to rapidly diminishing cache effectiveness. To address this bottleneck, later designs extend KV cache management to multi-tier memory hierarchies by migrating evicted caches to CPU DRAM or other lower-cost storage layers [23, 27, 34, 47]. Representative systems include LMCache [27], which chunks KV caches and accelerates CPU–GPU transfers via high-performance CUDA kernels. Mooncake [38] is designed to manage disaggregated KV caches by pooling CPU, DRAM, SSD and RDMA resources of the GPU cluster. RAGCache [17] leverages the retrieval pattern to organize the prefix KV caches of retrieved knowledge in a knowledge tree across the GPU and host memory hierarchy, reducing LLM generation time in RAG. CachedAttention [10] employs layer-wise pre-loading and scheduler-aware fetching to mitigate KV cache access overheads from slow storage tiers and to place KV caches along the critical path in the fastest memory hierarchy. Pensieve [54] manages partially dropped context by efficiently moving data between GPU and CPU cache tiers and recomputing missing segments as needed. Nevertheless, these approaches couple cache management with specific system designs or continue to rely on conventional replacement policies such as LRU or LFU. In contrast, our work focuses on cache replacement itself, proposing a simple and general policy for prefix caching.

Other KV cache related optimizations. Beyond vanilla prefix caching, recent work has explored KV cache blending [1, 12, 53, 55] to enable non-prefix KV cache reuse, and KV cache compression [11, 29, 30, 48] to reduce KV cache memory overhead. CacheBlend [53] and CacheCraft [1] extend KV cache reuse beyond strict prefix positions by merging KV caches from multiple documents and selectively recomputing a small subset of tokens to preserve output quality. DepCache [55] proposes a graph-based KV cache reuse strategy that aligns KV caches across varying prompt contexts to enable efficient cross-request reuse. CacheGen [29] compresses and streams KV caches to reduce KV cache transfer delay for long-context inference. EVICPRESS [9] applies lossy compression and adaptive eviction to manage KV caches across multiple storage tiers while jointly optimizing generation quality and latency. DroidSpeak [28] explores cross-model KV cache sharing by enabling KV cache reuse across different LLMs. These optimizations are orthogonal to our work and can be combined with Buoy. Buoy can apply to non-prefix KV caches by disabling depth-aware cache admission and remains independent of whether KV caches are compressed.

Accelerating LLM serving. Another line of work accelerates LLM serving through system-level optimizations. These efforts include improved scheduling strategies to reduce pipeline bubbles across different inference stages [2, 7]; prefill–decode disaggregation, which places the prefill and decode phases on separate GPUs to mitigate SLO violations [36, 61]; and speculative decoding techniques that leverage a smaller and faster draft model to generate candidate token sequences, which are

then verified in parallel by a larger target model [14, 22]. These approaches are complementary to prefix caching with Buoy and can be integrated with it to further improve overall LLM inference performance.

7 Conclusion

In this paper, we revisit cache replacement for prefix caching in LLM inference systems. We show that conventional cache replacement policies, such as LRU, are insufficient because they ignore both prefix-matched access characteristics and the hierarchical nature of GPU–CPU memory. We present Buoy, an efficient and effective cache replacement policy specifically designed for prefix caching. Its high efficiency comes from quick demotion and depth-aware cache admission, while its effectiveness is achieved by maximizing cache hits in HBM, reducing host-to-GPU KV cache transfers. Extensive evaluation on real-world traces demonstrates that Buoy outperforms state-of-the-art cache replacement policies.

Acknowledgments

This work was sponsored by the National Natural Science Foundation of China under Grant No. 62502170, the Special High-Quality Program of MIIT under Grant No. 2540STC62590, the Shandong Provincial Natural Science Foundation under Grant No. ZR2024LZH004, and the Shenzhen Science and Technology Program under Grant No. JCYJ20240813153433043.

References

[1] Shubham Agarwal, Sai Sundaresan, Subrata Mitra, Debabrata Mahapatra, Archit Gupta, Rounak Sharma, Nirmal Joshua Kapu, Tong Yu, and Shiv Saini. 2025. Cache-craft: Managing chunk-caches for efficient retrieval-augmented generation. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[2] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369* (2023).

[3] Mistral AI. 2025. *Mistral Small 3.* https://mistral.ai/news/mistral-small-3

[4] Sorav Bansal and Dharmendra S Modha. 2004. Car: Clock with adaptive replacement. In *FAST*, Vol. 4. 187–200.

[5] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 389–403.

[6] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. 2025. IMPRESS: An Importance-Informed Multi-Tier Prefix KV Storage System for Large Language Model Inference. In *23rd USENIX Conference on File and Storage Technologies (FAST)*. 187–201.

[7] Weihao Cui, Yukang Chen, Han Zhao, Ziyi Xu, Quan Chen, Xusheng Chen, Yangjie Zhou, Shixuan Sun, and Minyi Guo. 2025. Optimizing SLO-oriented LLM Serving with PD-Multiplexing. *arXiv preprint arXiv:2504.14489* (2025).

[8] Gil Einziger, Roy Friedman, and Ben Manes. 2017. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)* 13, 4 (2017), 1–31.

[9] Shaoting Feng, Yuhan Liu, Hanchen Li, Xiaokun Chen, Samuel Shen, Kuntai Du, Zhuohan Gu, Rui Zhang, Yuyang Huang, Yihua Cheng, et al. 2025. EVICPRESS: Joint KV-Cache Compression and Eviction for Efficient LLM Serving. *arXiv preprint arXiv:2512.14946* (2025).

[10] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (ATC)*. 111–126.

[11] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems* 37 (2024), 1270–1303.

[12] Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. 2025. EPIC: efficient position-independent caching for serving large language models. In *Proceedings of the 42nd International Conference on Machine Learning*.

[13] Zhengding Hu, Vibha Murthy, Zaifeng Pan, Wanlu Li, Xiaoyi Fang, Yufei Ding, and Yuke Wang. 2025. Hedrarag: Co-optimizing generation and retrieval for heterogeneous rag workflows. In *Proceedings of the ACM SIGOPS 31st symposium on operating systems principles*. 623–638.

[14] Kaiyu Huang, Hao Wu, Zhubo Shi, Han Zou, Minchen Yu, and Qingjiang Shi. 2025. Adaspec: Adaptive speculative decoding for fast, slo-aware large language model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*. 361–374.

[15] Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 31–42.

[16] Wenqi Jiang, Suvinay Subramanian, Cat Graves, Gustavo Alonso, Amir Yazdanbakhsh, and Vidushi Dadu. 2025. Rago: Systematic performance optimization for retrieval-augmented generation serving. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 974–989.

[17] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. 2025. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems* 44, 1 (2025), 1–27.

[18] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*. 439–450.

[19] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. 1994. Caching strategies to improve disk system performance. *Computer* 27, 3 (1994), 38–46.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 611–626.

[21] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers* 50, 12 (2001), 1352–1361.

[22] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[23] Hanchen Li, Qiuyang Mang, Runyuan He, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. 2025. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. *arXiv preprint arXiv:2511.02230* (2025).

[24] Peizheng Li, Chaoyi Chen, Hao Yuan, Zhenbo Fu, Hang Shen, Xinbo Yang, Qiange Wang, Xin Ai, Yanfeng Zhang, Yingyou Wen, et al. 2025. NeutronRAG: Towards Understanding the Effectiveness of RAG from a Data Retrieval Perspective. In *Companion of the 2025 International Conference on Management of Data*. 163–166.

[25] Yuhang Li, Rong Gu, Chengying Huan, Zhibin Wang, Renjie Yao, Chen Tian, and Guihai Chen. 2025. HotPrefix: Hotness-Aware KV Cache Scheduling for Efficient Prefix Sharing in LLM Inference Systems. *Proceedings of the ACM on Management of Data* 3, 4 (2025), 27 pages.

[26] Yunfan Li, Huiqi Hu, Chaojing Lei, Xuan Zhou, and Weining Qian. 2024. Hill-Cache: Adaptive Integration of Recency and Frequency in Caching with Hill-Climbing. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3947–3960.

[27] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. 2025. LMCache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665* (2025).

[28] Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, et al. 2024. DroidSpeak: KV Cache Sharing for Cross-LLM Communication and Multi-LLM Serving. *arXiv preprint arXiv:2411.02820* (2024).

[29] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM)*. 38–56.

[30] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. KIVI: a tuning-free asymmetric 2bit quantization for KV cache. In *Proceedings of the 41st International Conference on Machine Learning*. 32332–32344.

[31] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*.

[32] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[33] OpenAI. 2024. New embedding models and API updates. https://openai.com/blog/new-embedding-models-and-api-updates/.

[34] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. 2024. Instinfer: In-storage attention offloading for cost-effective long-context llm inference. *arXiv preprint arXiv:2409.04992* (2024).

[35] Sejin Park and Chanik Park. 2017. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*.

[36] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[37] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vattention: Dynamic memory management for serving llms without pagedattention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1133–1150.

[38] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST)*. 155–170.

[39] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 341–354.

[40] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. 1999. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review* 27, 1 (1999), 122–133.

[41] Adam J Storm, Christian Garcia-Arellano, Sam S Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive self-tuning memory in DB2. In *Proceedings of the 32nd international conference on Very large data bases*. 1081–1092.

[42] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).

[43] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[44] vLLM Project. 2026. vLLM. https://github.com/vllm-project/vllm

[45] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider. In *2025 USENIX Annual Technical Conference (ATC)*. 465–482.

[46] Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. 2026. Strata: Hierarchical context caching for long context language model serving. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. 1–16.

[47] Yi Xiong, Hao Wu, Changxu Shao, Ziqing Wang, Rui Zhang, Yuhong Guo, Junping Zhao, Ke Zhang, and Zhenxuan Pan. 2024. Layerkv: Optimizing large language model serving with layer-wise kv cache management. *arXiv preprint arXiv:2410.00428* (2024).

[48] Ceyu Xu, Yongji Wu, Xinyu Yang, Beidi Chen, Matthew Lentz, Danyang Zhuo, and Lisa Wu Wills. 2025. LLM.265: Video Codecs are Secretly Tensor Codecs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. 445–460.

[49] Juncheng Yang, Ziming Mao, Yao Yue, and KV Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 115–134.

[50] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. 2023. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS)*. 70–79.

[51] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 130–149.

[52] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600* (2018).

[53] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*. 94–109.

[54] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. Stateful Large Language Model Serving with Pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*. 144–158.

[55] Hao Yuan, Xin Ai, Qiange Wang, Peizheng Li, Jiayang Yu, Chaoyi Chen, Xinbo Yang, Yanfeng Zhang, Zhenbo Fu, Yingyou Wen, et al. 2025. DepCache: A KV Cache Management Framework for GraphRAG with Dependency Attention. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–29.

[56] Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. 2025. Pqcache: Product quantization-based kvcache for long context llm inference. *Proceedings of the ACM on Management*

of Data 3, 3 (2025), 1–30.

[57] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. 2024. SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1229–1246.

[58] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. Wildchat: 1m chatgpt interaction logs in the wild. *arXiv preprint arXiv:2405.01470* (2024).

[59] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P Xing, et al. 2023. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998* (2023).

[60] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems (NeurIPS)*. 62557–62583.

[61] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 193–210.

[62] Wenbin Zhou, Zhixiong Niu, Yongqiang Xiong, Juan Fang, and Qian Wang. 2025. 3L-Cache: Low Overhead and Precise Learning-based Eviction Policy for Caches. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 237–254.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026