HotPrefix: Hotness-Aware KV Cache Scheduling for Efficient Prefix Sharing in LLM Inference Systems

YUHANG LI, State Key Laboratory for Novel Software Technology, Nanjing University, China
RONG GU*, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHENGYING HUAN, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHIBIN WANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
RENJIE YAO, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHEN TIAN, State Key Laboratory for Novel Software Technology, Nanjing University, China
GUIHAI CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

Prompt engineering techniques are widely used to enhance the generation quality of large language models (LLMs). However, the long prompts significantly increase inference latency and reduce inference throughput. Since many prompts share common prefixes, prefix sharing has been proposed to reuse shared prefix KV caches during inference. Nevertheless, the large number of prefix KV caches and the limited GPU memory capacity make it impractical to store all prefix KV caches in GPU memory. This limitation necessitates the use of external memory storage strategies, which often suffer from high I/O overhead and frequent cache misses with traditional approaches.

To address these challenges, this paper proposes HotPrefix, a hotness-aware KV cache scheduling framework designed for efficient prefix sharing in LLM inference systems. HotPrefix introduces three core innovations: (1) **Dynamic Hotness Tracking**, which dynamically monitors and updates the hotness of prefix tree nodes over time; (2) **Selective KV Cache Admission**, which evaluates evicted KV caches from GPU memory, retaining only high-hotness caches in CPU memory to expand GPU memory capacity and reduce KV cache transfer overhead; (3) **Hotness Promotion**, which periodically promotes high-hotness prefix tree KV caches from CPU memory to GPU memory. This is combined with an efficient pipeline strategy for I/O and computation, ensuring GPU memory is allocated to the most critical prefixes while masking the I/O overhead associated with KV cache transmission. These mechanisms significantly improve cache hit rates, reduce inference latency, and enhance throughput. Implemented in the SGLang framework, HotPrefix reduces inference latency and increases throughput by up to 2.25× compared with vLLM with prefix sharing enabled. Against SGLang, it achieves up to 2× latency reduction and throughput increase.

CCS Concepts: • **Information systems → Hierarchical storage management**.

*Rong Gu is the corresponding author of this paper.

Authors' Contact Information: Yuhang Li, liyuhang@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Rong Gu, gurong@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Chengying Huan, huanchengying@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Zhibin Wang, wzbwangzhibin@gmail.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Renjie Yao, 522024330111@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Chen Tian, tianchen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Guihai Chen, gchen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China.

ACM 2836-6573/2025/9-ART250
https://doi.org/10.1145/3749168

Additional Key Words and Phrases: Large Language Model Inference, KV Cache, Prefix Sharing

ACM Reference Format:
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. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 250 (September 2025), 27 pages. https://doi.org/10.1145/3749168

1 Introduction

Large language models (LLMs) such as GPT-4 [2], LLaMA3 [12], and DeepSeek [6] have achieved remarkable success across a wide range of applications, including natural language understanding, reasoning, text generation, and question answering [41, 42, 49]. To further enhance LLM performance and adapt them to diverse tasks, prompt engineering [7, 32, 33, 64, 65, 71] has emerged as a key technique, enabling LLMs to perform effectively with minimal hardware resource costs. By designing structured and informative prompts, including techniques like few-shot learning [7], chain-of-thought [56] and retrieval-augmented generation (RAG) [32] that incorporate external knowledge, users can guide LLMs to address specific tasks with high precision and flexibility.

However, prompt engineering techniques also introduce challenges in inference. A key issue is the redundant storage of shared prefixes, such as system instructions [28, 39], reusable templates [57, 63], or RAG retrieved documents [9]. For example, system prompts are shared by all queries in various LLM applications, such as Claude [4], ChatGPT [44] and Grok [58], and the promopts often exceed 1,000 tokens [28]. For popular benchmarks, like MMLU [21], BBH [51] and CEVAL [23], the chain-of-thought or few-shot prompts' average token length can be as long as 723, 567 and 1673, and each prompt is shared by hundreds of different queries [43]. This duplication results in inefficient use of GPU memory, limiting space for other data. To address this, prefix sharing [30, 66, 72] has been proposed to reuse shared KV caches, reducing redundant computations and optimizing GPU memory utilization.

Existing work has made notable advancements in prefix sharing for LLM inference. vLLM [30] reserves a fixed set of physical memory blocks for shared prefixes, enabling reuse of prefixes that align with predefined block sizes. SGlang [72] introduces the RadixAttention mechanism to optimize memory usage and facilitate flexible prefix sharing, managing memory with a simple LRU-based eviction policy. However, these methods fail to address several critical problems: (1) As the number of shared prefixes increases, the limited capacity of GPU high-bandwidth memory (HBM) makes it impossible to store all prefix KV caches. For example, GPT-3 requires approximately 4.5 MB of KV cache per token [17]. With a context length of 1024 and batch size of 16, the KV cache demand exceeds 73 GB. (2) Current methods do not account for the dynamic nature of user requests, where reuse probabilities of prefixes can vary significantly. Studies [25] have shown that prefix reuse probabilities follow a long-tail distribution, with a small subset of high-hotness prefixes responsible for the majority of requests, while low-hotness prefixes exhibit unpredictable access patterns, making static caching strategies inefficient. (3) They fail to efficiently utilize abundant CPU memory for storing high-hotness prefixes while ensuring that KV cache transfers and computations are overlapped to minimize performance disruptions. As a result, high-hotness prefixes are often evicted prematurely, leading to frequent recomputation and suboptimal inference performance.

1.1 Challenges

As discussed above, developing an efficient KV cache management framework that optimizes the locality of prefix KV caches, combined with a hotness-aware KV cache scheduling strategy, to improve latency and throughput presents several significant challenges.

Challenge I: Effective cache scheduling relies on dynamically tracking the hotness of prefix tree nodes, which is challenging due to the constantly changing access patterns during inference.

Achieving accurate tracking, low overhead, and real-time updates for prefix hotness simultaneously is particularly difficult, as these objectives often conflict with each other. Without effectively balancing these factors, the system risks introducing significant overhead, which can degrade inference performance. Therefore, designing an efficient mechanism to track prefix hotness dynamically and with minimal cost remains a key challenge.

Challenge II: Offloading KV caches from GPU memory to host memory is crucial for overcoming the limited capacity of GPU memory. However, this process incurs substantial I/O overhead, as frequent migrations can create bottlenecks, particularly in large-scale or dynamic inference workloads. For example, when SGLang runs MMLU benchmark and simply evicts redundant KV caches from GPU to CPU with Llama-2 13B model and batch size 8, the total inference latency is 2138 seconds and the offload time is 1200 seconds in our experiments, which is prohibitively expensive and makes this approach impractical for production deployment. Therefore, naive strategies that indiscriminately offload all KV caches to host memory exacerbate this problem, leading to degraded system performance.

Challenge III: Given the limited capacity of GPU memory, it is critical to ensure that it is reserved for high-hotness KV caches that are essential for ongoing inference tasks. Existing methods [30, 72] often fail to effectively manage GPU memory, resulting in frequent eviction of high-hotness caches in favor of low-hotness ones. This inefficiency reduces cache hit rates and increases recomputation, ultimately degrading inference latency and throughput.

1.2 Contributions

To address the challenges outlined above and improve inference latency and throughput, this paper proposes HotPrefix, a novel hotness-aware KV cache scheduling mechanism designed to maximize the reuse of prefix KV caches during LLM inference. HotPrefix leverages GPU-host memory collaboration and dynamic cache management to efficiently allocate limited memory resources, prioritizing high-hotness prefix KV caches for storage in GPU memory. By significantly enhancing the reuse of prefix KV caches, HotPrefix improves overall inference performance. Specifically, the key contributions of this work are as follows.

To address Challenge I, this paper introduces a **Dynamic Hotness Tracking**(§ 3.2) mechanism that monitors the hotness of each prefix tree node in real time, allowing it to quickly adapt to changes in access patterns. By continuously updating hotness information, HotPrefix ensures that cache management decisions are based on accurate, up-to-date metrics, prioritizing high-hotness prefixes for storage in GPU memory and improving overall cache performance.

To tackle Challenge II, HotPrefix incorporates a **Selective KV Cache Admission**(§ 3.3) strategy coupled with a Hotness-Aware Eviction Policy. The eviction policy removes KV caches with the lowest reuse probability first, optimizing GPU memory usage. Meanwhile, the selective admission strategy evaluates the hotness of evicted caches and only migrates high-hotness KV caches to host memory, discarding lower-hotness ones. This dual optimization minimizes unnecessary transfers, ensuring that critical KV caches remain accessible, significantly reducing I/O overhead while maintaining high inference performance.

To address Challenge III, we employ a **Hotness Promotion**(§ 3.4) strategy to asynchronously transfer shallow-depth high-hotness KV caches from host memory to GPU memory. This prioritizes the most important caches for storage in GPU memory. To minimize the latency of KV cache transmission, HotPrefix pipelines these transfers with ongoing GPU computation during the decoding phase, where prefix KV caches are rarely reused. This alignment ensures that data transmission and computation do not interfere with each other, reducing disruption to inference performance while improving GPU memory utilization.

Together with the above technology, **HotPrefix** is implemented in the SGLang, and we evaluate it on various datasets and representative LLMs. Extensive experimental results demonstrate that HotPrefix significantly improves performance, reducing inference latency by up to 2.25× and increasing throughput by up to 1.91× compared to vLLM with prefix sharing enabled. Compared to SGLang, HotPrefix achieves up to a 2× reduction in latency and a 1.64× increase in throughput.

2 Background and Motivation

This section first introduces the fundamentals of large language models(§ 2.1). Then we present the process of generative inference and how KV cache works(§ 2.2). Finally, we discuss the challenges faced in prefix sharing inference systems and opportunities to deal with them.

2.1 Large Language Models

Large language models (LLMs) have emerged as a foundational technology in natural language processing, excelling in generative tasks such as text generation, translation, and question answering. These models are predominantly based on the Transformer [20, 55] architecture, which has established itself as the de facto standard for LLMs such as GPT [44] and LLaMA [53, 54] due to its scalability and its ability to effectively model complex sequence dependencies.

At the core of the Transformer architecture is a stack of l transformer layers, each comprising two main components: *self-attention* and the *feed-forward network* (FFN). During inference, LLMs process input prompts as token sequences $X = [x_1, x_2, \ldots, x_s]$. Each token is projected using three learned weight matrices, W_Q, W_K, and W_V, to produce the Query (Q), Key (K) and Value (V) matrices. Self-attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{1}$$

Here, Q, K, and V are matrices derived from input, reshaped to dimensions $H \times N \times d$, where H is the number of attention heads, N is the sequence length, and d is the head dimension such that $D = H \times d$. This mechanism dynamically adjusts the weight of each token in the context of all others, allowing the model to effectively capture long-range dependencies.

After the self-attention phase, the outputs are passed through a feed-forward network, which comprises two fully connected layers with an activation function, such as GeLU, applied between them. The FFN enhances the model's ability to learn non-linear representations. Residual connections and layer normalization are employed to stabilize training and improve convergence. Notably, the output dimensions of each Transformer layer match its input dimensions, enabling seamless stacking of layers.

2.2 Generative Inference and KV Caching

Generative inference in large language models (LLMs) typically consists of two sequential stages: the prefill stage and the decoding stage. These stages work together to process input prompts and generate relevant text in an autoregressive manner.

Prefill Stage: The prefill stage processes the input prompt, $X = [x_1, x_2, \ldots, x_s]$, and computes a summary of its context. During this phase, the model generates the key (K) and value (V) tensors for all tokens in the input sequence, which are stored as the KV cache. This cache encapsulates the relationships among tokens in the input prompt and serves as the foundation for the subsequent decoding stage. The prefill stage ends with the generation of a single token, x_{s+1}, which serves as the initial input for the decoding phase.

Decoding Stage: The decoding stage generates tokens iteratively, one at a time. At each iteration i, the model processes the token x_{s+i} along with the KV cache accumulated during the prefill stage and previous decoding iterations. This stage involves the following two key steps:

- **Updating the KV Cache:** The keys and values of the current token are computed and appended to the existing KV cache, which grows incrementally. At the i-th iteration, the updated KV cache has dimensions $H \times (N + i) \times d$, where H is the number of attention heads, N is the length of the input prompt, and d is the head dimension.
- **Generating the Next Token:** Using the updated KV cache, the model computes the attention scores for the current token based on all previous tokens. The next token, x_{s+i+1}, is then generated and passed to the next iteration.

Unlike the prefill stage, the decoding stage must operate sequentially due to its dependency on the output of the previous iteration.

Reusable KV Cache: The KV cache is a crucial component in generative inference, as it significantly reduces redundant computation. During the decoding phase, all previously computed K and V tensors are reused to compute self-attention for new tokens. Without this mechanism, the model would need to recompute keys and values for all prior tokens at every iteration, resulting in prohibitive computational overhead. However, the KV cache introduces challenges due to its large memory footprint. For instance, GPT-3 generates a KV cache of approximately 4.5MB per token [17], and its size grows linearly with the number of tokens in a session.

Prefix Sharing: Prefix sharing is an optimization technique primarily used during the prefill stage to reduce redundant computation and memory usage [19, 25, 66, 72]. In scenarios where multiple requests share the same input prompt prefix, the KV cache for the shared prefix is computed only once and reused across all requests. This eliminates the need to recompute the keys (K) and values (V) for the common prefix, significantly enhancing the efficiency of the prefill stage.

2.3 Motivation and Limitations of SOTAs

Opportunities for Reusing KV Caches: A significant portion of input prefixes in LLM inference are reusable [66], presenting an opportunity to reduce redundant computation through KV cache reuse. In many LLM-based applications, such as chatbots, document analysis, and few-shot learning, prefixes often include system instructions, reusable templates, or shared context.

For example, in multi-tenant architectures, system prompts are shared across multiple requests and frequently contain detailed instructions or examples to guide the model [19]. In ChatGPT-like applications, a single system prompt can exceed 1,700 tokens when multiple plugins are activated [66], and all requests share this same prompt. Additionally, in long-context applications such as legal analysis [10], healthcare [50], and education [47], documents from a common pool are often reused across prompts. These redundancies create a clear opportunity for KV cache reuse, eliminating the need to recompute keys and values for identical prefixes during inference.

Insufficient GPU Memory: While GPUs excel in computation, their limited memory capacity poses a significant challenge for caching large-scale prefixes [13, 48]. As the number of tasks and input prompts increases, the KV cache size grows linearly, rapidly consuming available GPU memory [31]. This issue is particularly pronounced in scenarios with long prefixes or shared prefixes across multiple requests [25], as the GPU memory must simultaneously accommodate KV caches for all active requests.

Limited GPU memory not only restricts total storage capacity but also hampers effective caching of frequently accessed prefixes. Due to memory constraints, high-hotness prefixes that are repeatedly reused across requests may be prematurely evicted to make room for new KV caches. This leads to unnecessary recomputation of keys and values for these prefixes, negating the potential

benefits of cache reuse. The lack of sufficient memory to cache high-hotness prefixes directly increases inference latency and reduces throughput. To address these challenges, we focus on leveraging host memory to expand KV cache storage capacity.

Inefficiency of Simple GPU-Host Collaboration: A naive approach to extending KV cache capacity is to offload all KV caches evicted from GPU memory to host memory and reload them back to the GPU when needed. However, this approach is highly impractical due to the significant overhead it incurs [24].

First, transferring KV caches from GPU memory to host memory results in substantial transmission overhead [48]. Since KV cache tensors are large and memory bandwidth between the GPU and host is limited, frequent offloading operations introduce significant delays in the inference process. Second, reloading KV caches from host memory back to GPU memory when reuse is required is even more problematic [26]. The time required for this operation often exceeds the time needed to recompute the keys and values for the same prefixes directly on the GPU. Consequently, relying on simple GPU-host collaboration can reduce overall efficiency rather than improve it, especially in scenarios that demand high throughput or low latency performance.

To overcome these limitations, our approach introduces an intelligent KV cache scheduling strategy. This strategy aims to minimize the overhead of offloading KV caches to host memory while optimizing cache reuse. Specifically, it prioritizes which KV caches should be retained in GPU memory and anticipates future cache access patterns. By proactively and asynchronously preloading high-hotness prefix KV caches from host memory back to the GPU, the strategy ensures that KV caches with a higher probability of reuse are readily available when needed, thereby improving overall inference performance.

3 The HotPrefix Design

In this section, we first present the design of HotPrefix(§ 3.1). Then we introduce Dynamic Hotness Tracking to monitor the hotness of prefix tree nodes(§ 3.2). Next, we present *Selective KV Cache Admission* to improve the GPU memory usage and reduce KV cache transfer overhead(§ 3.3). Finally, we introduce *Hotness Promotion* to ensure most hot prefix KV caches kept in GPU memory(§ 3.4).

3.1 Overview

To reduce the computational overhead during the prefill stage and improve both end-to-end inference latency and throughput, this paper proposes an efficient KV cache management and scheduling mechanism called HotPrefix. HotPrefix maximizes the reuse of previously computed KV caches from earlier inference queries.

Key Insight: The key insight behind HotPrefix is to utilize the high bandwidth of GPU memory to store high-hotness prefix KV caches, which are more likely to be reused in the future, thus reducing the need for prefill computations. HotPrefix then optimizes the sharing of these prefixes, improving overall system performance.

However, when a new batch of prompts arrives, GPU memory may not have sufficient space to store all KV caches. In this case, HotPrefix avoids relying on traditional cache replacement strategies such as LRU, which could lead to higher cache misses and additional I/O for data transfer between host and GPU memory. Instead, HotPrefix introduces a more sophisticated and effective data storage and memory management strategy. It selectively offloads high-hotness KV caches that are more likely to be accessed in the future, while discarding lower-hotness caches. This selective approach ensures that only the most relevant KV caches are retained in GPU memory, and the others are transferred to host memory.

Additionally, HotPrefix incorporates a dynamic promotion mechanism to further optimize cache management. When the hotness of KV caches in host memory significantly exceeds that of those in

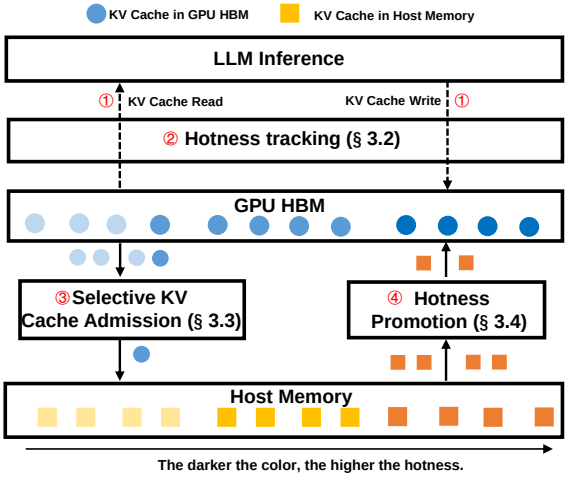

Fig. 1. The workflow of HotPrefix.

GPU memory, HotPrefix removes less relevant caches from the GPU and loads higher-hotness caches from the host memory. This process ensures that GPU memory always contains the most critical KV caches, maximizing cache hit rates, reducing inference latency, and improving throughput.

Figure 1 illustrates the workflow of HotPrefix in managing hierarchical memory. The workflow consists of three core components: (1) **Dynamic Hotness Tracking**, which monitors and updates the hotness of prefix tree nodes in real time; (2) **Selective KV Cache Admission**, which implements a hotness-aware eviction policy to determine which KV caches should be offloaded from GPU memory to host memory based on their hotness and reuse potential; (3) **Hotness Promotion**, which prioritizes high-hotness nodes for promotion from host memory to GPU memory. These components collaboratively address the challenges outlined in Section 1.

To address Challenge I and enable real-time tracking of prefix tree node hotness, **HotPrefix** employs a **Dynamic Hotness Tracking** mechanism. This mechanism records and updates the hotness of prefix tree nodes during KV cache reads and writes, providing precise hotness information for cache management decisions in a low-overhead manner (§ 3.2).

To address Challenge II and reduce the overhead of KV cache transfers between GPU HBM and host memory while increasing the cache hit ratio, HotPrefix adopts a Selective KV Cache Admission strategy. This approach offloads only high-hotness KV caches to host memory and discards low-hotness ones, minimizing data transfer costs and enhancing overall efficiency (§ 3.3).

To address Challenge III and ensure that GPU memory contains the most critical KV caches, HotPrefix incorporates a Hotness Promotion mechanism. This mechanism prioritizes high-hotness KV caches with shallower depths for promotion from host memory to GPU memory, thereby maximizing cache hit rates and reducing inference latency (§ 3.4).

3.2 Dynamic Hotness Tracking

For this part, we first introduce the basis of prefix tree based KV cache orgnization and then present the design of Dynamic Hotness Tracking in detail.

Figure 2 illustrates how the prefix tree KV caches architecture in HotPrefix where circular nodes represent KV caches stored in GPU HBM, while square nodes represent KV caches stored in host memory. We can see that in HotPrefix, KV caches are managed at the granularity of prefix tree

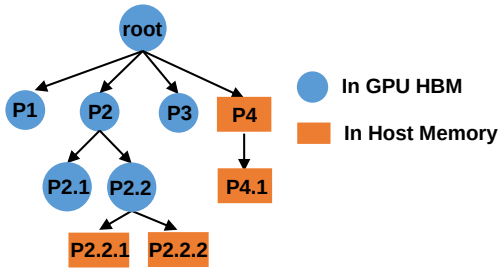

Fig. 2. The basic prefix tree structure of HotPrefix.

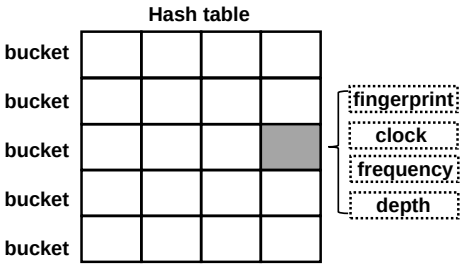

Fig. 3. The hotness-aware cuckoo filter.

nodes. In prefix tree, the parent node stands for the prefix of the child node. For example, in figure 2 child nodes $P2.1$ and $P2.2$ share the same prefix $P2$.

To identify prefixes KV caches that are most likely to be accessed in the future which can be previously cached in GPU, HotPrefix introduces a lightweight and accurate mechanism for dynamically tracking the hotness of prefix tree nodes when facing different llm inference queries.

To achieve this, the paper first introduces an enhanced cuckoo filter structure to record the hotness information of prefix tree nodes (§ 3.2.1). This structure offers low storage overhead while maintaining high search efficiency. Second, a dynamic update mechanism is implemented to continuously track changes in node hotness over time (§ 3.2.2). This mechanism updates the cuckoo filter dynamically with minimal overhead.

3.2.1 Hotness Recording Data Structure.

Firstly, this paper utilizes a cuckoo filter [14], which is a lightweight and efficient data structure, to track the hotness of prefix tree nodes. As illustrated in Figure 3, a novel enhanced hotness-aware cuckoo filter is proposed to facilitate dynamic and efficient management of prefix KV caches.

The hotness-aware cuckoo filter is implemented as a hash table with a contiguous address space, consisting of n buckets, where each bucket holds four entries. Each entry stores metadata for a prefix tree node: fingerprint, clock, frequency, and depth. Fingerprint is a compact identifier that enables efficient storage and quick lookup. Clock indicates the time elapsed since the node was loaded into the GPU HBM, with higher values representing more recent entries. Frequency captures the access frequency of the node, reflecting its hotness and potential for reuse. Depth records the node's position in the prefix tree, providing structural context for cache management decisions.

Specifically, for KV cache insertion, when a new node m is generated in the prefix tree, the tokens from the root to m are extracted as its unique identifier, denoted as x in Equation 2. Equation 2 gets the fingerprint of x and Equation 3 and 4 get possible space for storing x's information. Therefore, using Equations 2 to 4, the fingerprint f of the node, along with its two candidate buckets b_1 and b_2 in the cuckoo filter, can be computed. The hash function in 3 and 4 is MurmurHash [5]. The slots in buckets b_1 and b_2 are then examined. If an empty slot is found, the fingerprint f of node m is stored in that slot, with its clock value initialized to the predefined max_age, its frequency set to 1, and its depth recorded as the node's depth in the prefix tree at the current moment. The max_age is the maximum value of clock value.

$$f = fingerprint(x), \tag{2}$$

$$b_1 = hash(f), \tag{3}$$

$$b_2 = b_1 \oplus hash(f). \tag{4}$$

If both buckets b_1 and b_2 are full, one of them is randomly selected, and a random slot within the selected bucket is chosen. The information of the node corresponding to that slot, denoted as node n, is then relocated to its alternate bucket b_3, calculated using Equation 4. If b_3 has an empty slot, the information of node n is moved to that slot, freeing its original slot to store the information of node m. However, if b_3 is full, the same procedure is applied iteratively until an empty slot is found.

If the maximum number of iterations is reached without finding an empty slot, the insertion fails, indicating insufficient space in cuckoo filter. If failure, some unused entries may be shuffled to host memory to free up space for new insertions. However, such insertion failures are highly unlikely for two reasons. First, the cuckoo filter is a compact data structure capable of storing information for tens of thousands of nodes while requiring only a few kilobytes of memory [14]. Second, the cuckoo filter is stored in host memory, which provides sufficient space to accommodate all entries.

3.2.2 *Hotness Updating Mechanism.*

When handling new LLM queries, HotPrefix updates the hotness of nodes in the cuckoo filter through two steps:

- **Step 1:** When a node m in the prefix tree is reused, the tokens from m to the root are collected to form x, as defined in Equation 2.
- **Step 2:** A periodic aging mechanism is designed to decrease the clock values of all entries in the cuckoo filter.

For **Step 1**, using Equations 2 to 4, the buckets b_1 and b_2 in the cuckoo filter that store the information of node m are identified. Based on the insertion process outlined in Section 3.2.1, it is guaranteed that one of these buckets contains the metadata of node m. Furthermore, by traversing b_1 and b_2, the metadata of node m is located, its frequency is increased by 1, and its clock value is reset to the predefined *max_age*. For example, in Figure 4, 'C' represents the clock value and 'F' represents the frequency value, with *max_age* set to 15. After reusing the node on the GPU, whose clock is 9 and frequency is 5, HotPrefix will update its clock to 15 and set its frequency to 6.

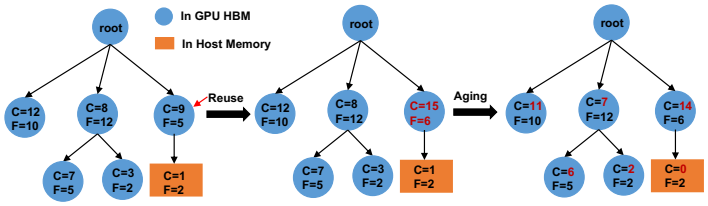

Fig. 4. The hotness updating strategy of hotness-aware cuckoo filter.

For **Step 2**, after a fixed number of user requests have been processed, the thread traverses the entire cuckoo filter and decrements the clock value of every entry by 1. For example, in Figure 4, after aging, the clock value of all the nodes in the prefix tree is reduced by 1. If a node's clock value reaches 0, indicating it has not been reused for a long time, the clock value is simply maintained at 0. This aging mechanism gradually reduces the clock values of prefix tree nodes, reflecting their decreasing recency. By periodically adjusting the clock values, HotPrefix effectively differentiates recently accessed nodes from less relevant ones.

3.3 Selective KV Cache Admission

HotPrefix leverages GPU HBM and host memory to manage prefix KV caches. However, when HBM space becomes insufficient, some KV caches must be evicted to host memory for storage. Since writing KV caches from HBM to host memory incurs significant I/O overhead, and an inefficient

strategy can result in frequent GPU memory cache misses, it is essential to develop an efficient KV cache selection strategy to offload less important KV caches from HBM to host memory. To mitigate this issue and ensure that the evicted KV caches are the least likely to be accessed in the future, HotPrefix employs a **Selective KV Cache Admission policy**, which utilizes a hotness-aware eviction strategy to identify and remove prefix tree nodes and their corresponding KV caches with the lowest hotness from GPU memory (§ 3.3.1). Subsequently, only KV caches with relatively high hotness are admitted into host memory, which significantly reduces offloading overhead (§ 3.3.2).

3.3.1 Hotness-aware Eviction Policy.

When encountering the out-of-GPU-memory problem, it becomes necessary to offload some KV caches from HBM to host memory. Existing approaches typically rely on a simple Least Recently Used (LRU) strategy for eviction. However, this approach does not account for the unique hierarchical structure of prefix trees and the varying access frequencies of nodes.

To overcome the limitations of LRU, HotPrefix adopts a hotness-aware replacement strategy. Unlike traditional LRU, this strategy considers the access characteristics of the prefix tree while balancing both recency and frequency metrics during eviction decisions. Since ancestor nodes in the prefix tree are inherently hotter than their descendant nodes, eviction decisions focus solely on the leaf nodes currently stored in GPU memory. Among these leaf nodes, HotPrefix determines node priority based on three key metrics: *frequency, clock, and length.* The priority is defined as:

$$priority = frequency + \frac{clock}{length}. \tag{5}$$

With the defined priority, nodes with lower priority are evicted first. In the priority definition equation, *frequency* represents the total number of times a node has been accessed, not limited to the time it has spent in GPU memory. If a node has been frequently swapped in and out of GPU memory, its overall access frequency remains cumulative. This frequency metric can be directly retrieved from the hotness record structure in Section 3.2.1. Next, *clock* reflects the time elapsed since the node was last loaded into GPU memory, which is also obtained from the hotness record structure. Finally, *length* denotes the number of tokens the node contains.

Equation 5 is designed to balance the contributions of long-term usage trends and short-term recency dynamics. The frequency term directly reflects the cumulative access count of the node, indicating its overall reuse potential. The clock term captures the node's recency, while the length term penalizes larger nodes, ensuring efficient memory utilization. The addition operator combines these orthogonal factors without disproportionately amplifying either, and the division operator introduces a proportional adjustment that smooths the impact of node size. This combination results in a balanced and interpretable prioritization metric for memory management. For example, for relatively long contexts, like RAG documents with thousands of tokens, Equation 5 helps to prioritize evicting them from GPU. While for extremely long contexts with tens of thousands of tokens, the KV caches of them are often stored off-node and fetched to inference servers via compressed transmission [13]. By leveraging these metrics, the hotness-aware replacement strategy minimizes the risk of evicting high-hotness nodes, ensuring more efficient GPU memory utilization and reducing the performance impact of suboptimal evictions.

Algorithm 1 outlines the procedure for evicting prefix KV caches from GPU memory to host memory when the memory capacity is exceeded. The algorithm first identifies all leaf nodes in the prefix tree and computes their priorities (lines 3–5). Nodes are then iteratively evicted based on their priority until the total GPU memory usage falls below the maximum capacity (lines 7–11). For each evicted node, its memory usage is subtracted from the GPU memory, and it is removed from both the GPU and the leaf node set. Finally, the incoming node and its KV cache are added to GPU memory.

Algorithm 1 Hotness-Aware Eviction Policy for GPU Memory

Require: Prefix tree nodes' total tokens in GPU memory N_{GPU}, maximum GPU capacity for tokens C_{GPU}, prefix tree root node $root\_node$
Ensure: Updated GPU memory after eviction

Input: Incoming node m, corresponding KV cache
while $|N_{GPU}| \geq C_{GPU}$ **do**
  $LeafNodes \leftarrow$ FindLeafNodes($root\_node$)
  for each $n \in LeafNodes$ **do**
    priority(n) $\leftarrow$ frequency(n) + $\frac{\text{clock}(n)}{\text{length}(n)}$
  end for
  while $|N_{GPU}| \geq C_{GPU}$ and $LeafNodes$ not empty **do**
    $EvictNode \leftarrow \text{argmin}_{n \in LeafNodes}(\text{priority}(n))$
    Remove $EvictNode$ from GPU
    Remove $EvictNode$ from $LeafNodes$
    N_{GPU} -= $EvictNode$.length
  end while
end while
Add node m to GPU

3.3.2 Selective Admission Policy.

After evicting nodes from GPU HBM, we selectively transfer some of their KV caches from GPU HBM to host memory, which incurs significant latency. If all evicted KV caches were directly offloaded to host memory without selection, the time cost would be prohibitive, leading to substantial degradation in inference latency and throughput. Such indiscriminate offloading would also result in host memory being flooded with low-value KV caches, further increasing the burden of managing these caches.

Observation: Our key observation is that while certain shared prefix KV caches are frequently reused across different queries, many other KV caches, especially those from user-specific queries or LLM outputs, are rarely reused. Retaining such low-hotness KV caches in host memory offers little to no benefit and can unnecessarily increase offloading overhead.

This insight forms the foundation for designing a selective admission mechanism to ensure that only valuable KV caches are retained. To address the high transfer overhead of offloading KV caches from HBM to host memory, HotPrefix implements a selective KV cache admission strategy, as shown in Figure 5. This strategy regulates both the quality and quantity of KV caches, focusing on preserving high-hotness caches while discarding low-value ones. The detailed process operates as follows:

Collect Evicted KV Caches From GPU: The blue circles on the left of Figure 5 represent KV cache nodes evicted from GPU memory, determined by the hotness-aware eviction policy described in Section 3.3.1. These evicted KV caches are potential candidates for offloading to host memory.

Frequency Threshold Filtering: Each evicted KV cache node undergoes a frequency threshold check. If the node's access frequency is below a predefined threshold, it is considered to have insufficient reuse potential and is immediately discarded. This step filters out KV caches with minimal future utility, ensuring that host memory is not burdened with low-frequency data.

Hotness Comparison: For KV cache nodes that pass the frequency threshold check, a further comparison is performed against the hotness of the least critical nodes currently stored in host

memory, represented by the orange rectangles in Figure 5. The hotness of a node is evaluated based on its access frequency and recency, which is defined as:

$$hotness = frequency * clock. \tag{6}$$

If the hotness of the current evicted node is lower than that of the least critical nodes in host memory, it is discarded. By ensuring that only KV caches with meaningful reuse potential are offloaded to host memory, this comparison step optimizes memory utilization and minimizes unnecessary transfer overhead, preserving host memory for high-hotness cache entries.

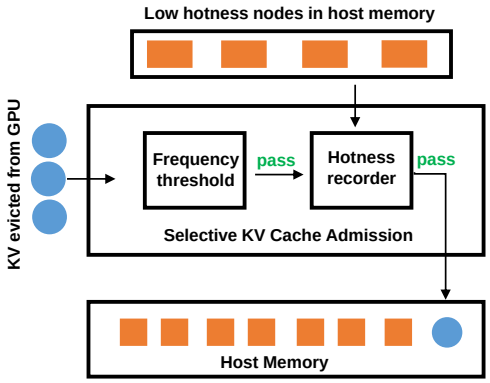

Fig. 5. The host memory admission process of HotPrefix.

Offloading to Host Memory: Only KV cache nodes that pass both the frequency threshold and hotness comparison are finally offloaded to host memory.

By carefully controlling the admission process, HotPrefix achieves two key objectives:

- **(1) Reducing transfer overhead:** The two-stage filtering process minimizes the amount of data transferred between GPU and host memory, significantly reducing offload latency.
- **(2) Improving memory utilization:** By discarding low-hotness KV caches, host memory is reserved for high-hotness data, improving the overall efficiency of memory usage.

It is important to note that while the selective admission policy reduces offloading overhead and memory congestion, it inherently introduces a trade-off. Some nodes with marginal reuse potential may be discarded prematurely, which could lead to occasional cache misses and slightly higher inference latency for specific queries. However, the benefit of preserving only high-value KV caches far outweighs this drawback, particularly in large-scale inference scenarios where memory and latency constraints are critical.

Additionally, HotPrefix utilizes host memory with a capacity equal to the space allocated for KV caches in GPU memory. This relatively small host memory footprint is sufficient because the hotness admission strategy ensures that only high-hotness KV caches are admitted to host memory. By filtering out low-hotness prefixes during the eviction process, HotPrefix minimizes the memory requirements for storing offloaded KV caches, while maintaining their utility for future reuse.

3.4 Hotness Promotion

In our design, a large number of prefix KV caches are stored in host memory. Loading reusable KV caches from host memory to GPU HBM only after a user request arrives incurs significant data transfer overhead, which can often exceed the cost of recomputing the KV caches. Therefore, this paper proposes an efficient prefetch strategy that proactively loads potentially reusable KV caches from host memory to GPU HBM.

Since user requests are unpredictable, accurately determining which KV caches should be loaded in advance is challenging. To maximize the likelihood of reuse, after the current batch prefill stage completes, HotPrefix simultaneously selects the hottest nodes from host memory and the least hot nodes from GPU memory (§ 3.4.1). HotPrefix then formulates a node promotion strategy (§ 3.4.2) to replace low-hotness nodes in GPU memory with high-hotness and shallow-depth nodes from host memory (§ 3.4.3). This approach ensures that GPU memory retains the most valuable KV caches, optimizing memory usage and reducing inference latency.

3.4.1 Nodes Promotion Strategy.

For nodes promotion, there are three design principles. First, multi-factor weighting scheme, in which node depth, KV cache size and node hotness are all considered to make the promotion decision. If the GPU tree node is far from root node, with large KV cache size and low hotness, it should be prioritized for offloading to CPU memory. Second, criteria for valid node selection. HotPrefix prioritizes exchange of coldest GPU leaf nodes with hottest CPU root nodes to ensure the KV caches in the GPU memory are of high hotness and improve its cache hit ratio. Thirdly, the timing strategy for promotion. By initiating promotion after the prefill phase where prefixes exhibit higher reuse potential than in the decode phase, HotPrefix maximizes overlap with decoding.

Figure 6 provides an example illustrating the complete process of hotness promotion. This promotion process is initiated only after the prefill phase for the current batch is completed. The primary rationale behind this design is that prefix sharing primarily benefits the prefixes generated during the prefill stage, whereas the content generated during the decode phase has a significantly lower likelihood of being shared. By deferring the promotion until the prefill phase is complete, HotPrefix enables the promotion process to overlap with batch decoding, ensuring that the two operations do not interfere with each other.

During hotness promotion, HotPrefix selects the coldest nodes in GPU memory and the hottest nodes in host memory from the prefix tree. For instance, as shown in Figure 6, nodes $P1.2$, $P2.1$, and $P2.2$ are identified as the coldest nodes in GPU memory, while nodes $P3$ and $P4$ are the hottest nodes in host memory. The selected nodes are then compared, and a promotion plan is created to determine which nodes in host memory should replace their counterparts in GPU memory.

The promotion plan considers multiple factors, including node hotness, depth, and KV cache size. Specifically, in the promotion plan illustrated in the figure, nodes $P1.2$ and $P2.1$ in GPU memory are replaced with the KV cache of node $P3$ from host memory, while node $P2.2$ is replaced with node $P4$. Finally, the Promote module executes the plan, promoting the hot nodes from host memory into GPU memory.

Regardless of whether the nodes are in GPU memory or host memory, the hotness of ancestor nodes is always higher than that of their descendants. Therefore, by collecting all leaf nodes in GPU memory, we can identify the coldest nodes in the GPU. In this context, leaf nodes in GPU memory refer to nodes that either have no descendants or have all their descendants stored in host memory. Similarly, by collecting all root nodes in host memory, we can identify the hottest nodes stored there. Here, root nodes in host memory are defined as nodes whose parent resides in GPU memory, while they themselves are located in host memory.

Next, we will introduce the process of promoting these high-hotness nodes from host memory to GPU memory.

3.4.2 Generate Promotion Plan.

When generating the promotion plan, nodes in host memory are first arranged in descending order of hotness, while nodes in GPU memory are sorted in ascending order of hotness. The hotness values used in this process are derived from the hotness record structure described in Section 3.2.1. For instance, as shown in Figure 6, the hotness of node $P3$ is higher than that of $P4$, while $P1.2$ is

less hot than $P2.1$, and $P2.1$ is less hot than $P2.2$. The promotion process evaluates the nodes in host memory sequentially. Starting with node $P3$, its hotness is compared to that of $P1.2$, the least hot node in GPU memory. Although $P3$ is hotter than $P1.2$, the number of tokens associated with $P3$ exceeds those of $P1.2$. Therefore, the next comparison is made between $P3$ and $P2.1$. In this case, $P3$ is hotter, and the combined number of tokens of $P1.2$ and $P2.1$ is sufficient to accommodate the tokens of $P3$. Consequently, $P3$ is selected for promotion, replacing the KV cache space occupied by $P1.2$ and $P2.1$. Similarly, node $P4$, being hotter than $P2.2$ and having a smaller token count, forms another executable promotion plan to replace $P2.2$.

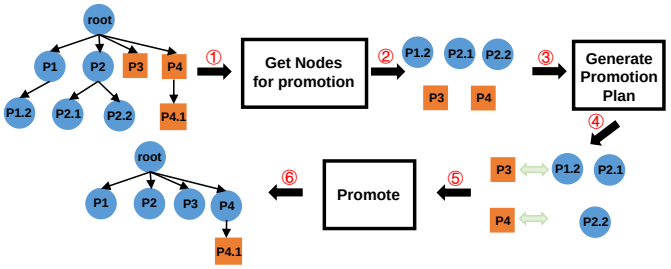

Fig. 6. The promotion process of HotPrefix.

It is important to note that when generating the promotion plan, if a node x in host memory has its parent node y in GPU memory and y is already marked as a candidate for eviction in the promotion plan, node x will be skipped. This is because promoting a child node x to GPU memory under the condition that its parent node y is being evicted is ineffective, as the hotness of a child node is always lower than that of its parent. Additionally, GPU nodes are processed in ascending order of hotness, while host memory nodes are processed in descending order. These sorting rules ensure that, during the creation of the promotion plan, node x will not be selected only to later find that its parent node y is being evicted. Consequently, this special case does not disrupt the formation of the promotion plan. In such scenarios, node x can be safely skipped without requiring any additional processing.

Algorithm 2 outlines the process for constructing a promotion plan to transfer high-hotness nodes from host memory to GPU memory. Nodes in host memory are first sorted in descending order of hotness, while GPU nodes are sorted in ascending order of hotness. The hotness values used for sorting are obtained from the hotness record structure (§ 3.2.1). For each node in host memory, the algorithm evaluates whether it can replace one or more GPU nodes while satisfying token space constraints (lines 6–15). If a node in host memory has a parent node in GPU memory that is already marked for eviction, it is skipped to maintain logical consistency (lines 7–9). The final promotion plan is then returned, ensuring that GPU memory retains nodes with the highest reuse potential.

3.4.3 Pipeline Promotion and Decoding.

Promote Nodes to GPU: Once the promotion plan is finalized, the *Promote* module executes the replacements by removing the selected low-hotness nodes from GPU memory and loading the corresponding high-hotness nodes from host memory into GPU memory. The evicted GPU nodes are directly discarded, as their low hotness indicates a very low probability of reuse. By discarding these nodes, the Promote module eliminates unnecessary data management overhead as well as ensuring that GPU memory is refreshed with high-hotness KV caches.

Algorithm 2 Promotion Plan Generation Workflow

Require: GPU nodes N_{GPU} and host memory nodes N_{HOST}
Ensure: Executable promotion plan

Input: GPU leaves N_{GPU} and host memory leaves N_{HOST}
Sort N_{GPU} in ascending order of hotness
Sort N_{HOST} in descending order of hotness
$PromotionPlan \leftarrow \emptyset$ ▷ Initialize promotion plan
for each node $x \in N_{HOST}$ **do**
  if x has a parent node $y \in N_{GPU}$ **then**
    if $y \in PromotionPlan$ **then**
      continue ▷ Skip node x if its parent is evicted
    end if
  end if
  for each node $z \in N_{GPU}$ **do**
    if hotness(x) > hotness(z) **then**
      if x.tokens $\leq$ z.tokens **then**
        Add $\{x : z\}$ to $PromotionPlan$
        break ▷ Move to the next host memory node
      else if x.tokens > z.tokens **then**
        Select more GPU nodes
        Add $\{x$: GPU nodes$\}$ to $PromotionPlan$
        break
      end if
    end if
  end for
end for
return $PromotionPlan$

The hotness promotion process does not increase GPU memory usage, as the promotion plan guarantees that the total memory footprint remains constant. However, if the total GPU memory required for the subsequent batch decoding stage, combined with previously stored KV caches, exceeds the available GPU memory capacity, HotPrefix applies the eviction strategy described in Section 3.3.1 to free up space for the decode stage.

To further optimize performance, the *Promote* module prioritizes batch transfers of KV caches from host memory to GPU memory. By grouping multiple transfers into a single operation, it significantly reduces the total number of memory copy calls and the resulting data transfer-associated latency.

Pipeline Promotion and Decoding: Additionally, HotPrefix pipelines data transfer operations over the PCIe bus between the GPU and CPU with ongoing GPU computations during the decode stage, as illustrated in Figure 7. By scheduling data transfers immediately after the prefill phase and aligning them with the decode phase, HotPrefix ensures that data movement occurs asynchronously, thereby minimizing disruptions to inference performance. This design leverages the key observation that the decoding process typically generates unique content, which is unlikely to reuse prefix KV caches. Consequently, overlapping promotion with decoding effectively hides the latency of transferring KV caches behind GPU computations, as decoding does not depend on the KV caches being transferred.

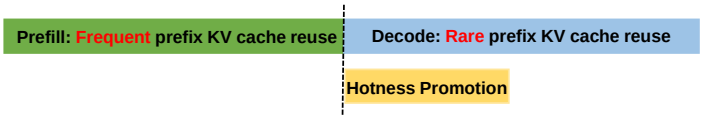

Fig. 7. Pipeline hotness promotion with decode stage.

However, a trade-off exists in the promotion: GPU nodes included in the promotion plan are marked as unavailable during the execution of the plan and will ultimately be discarded. This may result in minor performance degradation, as these nodes cannot be accessed during this period. Nonetheless, it is important to emphasize that the nodes selected for the promotion plan are inherently low-hotness nodes with a very low likelihood of reuse. Therefore, the impact of marking these nodes as unavailable is negligible. In contrast, the benefit of promoting high-hotness nodes from host memory to GPU memory is significant, as it greatly enhances cache hit rates and reduces future inference latency.

In summary, the proposed *Promote* Module efficiently refreshes GPU memory by replacing low-hotness KV caches with high-hotness KV caches from host memory, while avoiding unnecessary data transfers. This process ensures that GPU memory consistently retains the most valuable KV caches, thereby reducing inference latency and maximizing throughput.

4 Implementation

We implement HotPrefix based on SGLang, version 0.4.1.post1, a widely used LLM inference system. Firstly, following SGLang's design, we manage KV caches at per-token granularity across both CPU memory and GPU device memory. HotPrefix dynamically allocates and releases KV cache space for each user request on demand. Secondly, we employ a prefix tree to uniformly manage KV caches residing in both CPU and GPU memory, enabling efficient prefix sharing. Thirdly, for **Dynamic Hotness Tracking**, when a node is reused, its access frequency is incremented by 1 and its clock value is reset to pre-defined *max_age*. Additionally, a dedicated CPU-managed thread periodically decays node clock values to maintain temporal locality, ensuring zero overhead on GPU computation resources.

For equation 5 and 6, our implementation provides users with a convenient customization interface. Taking equation 5 as an example, the API is abstracted as "x op1 y op2 z", where x, y, z can be clock, frequency and length, and op1, op2 can be add(+), multiplication(*) and division(/). For example, if the data access exhibits obvious temporal variability, we can use clock + freq / length to capture the dynamic recency pattern. Forthly, for GPU KV cache eviction, if rejected by the **Selective Admission Policy**, the tree node metadata is immediately deleted and its corresponding GPU memory is released. Otherwise, if admitted to the CPU memory, the corresponding KV cache of the tree node will be transferred to CPU memory and the node metadata will be updated. Lastly, for **Hotness Promotion**, a dedicated CUDA stream handles asynchronous memory transferring and it executes concurrently with the decoding compute stream. This design achieves overlap between GPU computation and memory transfers due to the absence of prefix sharing during decoding phase.

5 Performance Evaluation

In this section, we first introduce the experimental setup (§ 5.1). Then we evalute HotPrefix with vLLM and SGLang on cache hit ratio, inference latency and throughput (§ 5.2). Finally, we do ablation studies on the techniques used in HotPrefix (§ 5.3).

5.1 Experimental Setup

Models: We evaluate HotPrefix using the LLaMA-2 13B model [53], Gemma 9B model [52], LLaMA-3 8B model [12] and Qwen-2 72B model [60]. These models represent different sizes and configurations, enabling us to assess the effectiveness of HotPrefix across a range of inference scenarios.

Testbed: The server for the experiments is configured with 256 GB of DRAM and a 3.84 TB NVMe SSD. The GPUs are connected to the host via PCIe Gen 3. HotPrefix is evaluated using both 13B, 8B and 72B models. For the 13B and 9B models, inference tasks are executed on a single NVIDIA A6000 GPU and we also run data parallel experiments on 4 A6000 GPUs using 13B model.To evaluate HotPrefix's robustness across different hardwares, we run 8B model on a single RTX 3090, A30, A6000 and A100 respectively with the same HBM memory configuration. To evaluate the scalability of HotPrefix, we run 72B model for tensor parallel experiments on 4 A6000 GPUs.

Workloads: We evaluate HotPrefix using four representative benchmarks: 5-shot MMLU [21], 10-shot Hellaswag [68], BBH [51] and CEVAL [23], with the latter two enhanced through Chain-of-Thought (CoT) prompting [56]. 5-shot MMLU is a multiple-choice knowledge benchmark where the model predicts a single token (A/B/C/D) as the answer after being primed with five in-context examples. Hellaswag is a commonsense dataset containing multiple-choice questions designed to evaluate model's understanding of everyday situations. The Big-Bench Hard (BBH) suite is designed to assess the model's ability to handle complex tasks. CEVAL is a Chinese-language evaluation dataset covering multiple domains. Applied with few-shot or chain-of-thought, the average number of input token of MMLU, Hellaswag, BBH, CEVAL are 567, 753, 723 and 1673. To simulate the randomness of user input arrival, queries for each dataset are shuffled. This setup ensures that workloads reflect realistic, dynamic query patterns, providing a robust evaluation of HotPrefix's adaptability and performance.

Metrics: We evaluate system performance using the following four key metrics, measured during batch execution:

- **Throughput:** Defined as the number of tokens processed by the LLM per second (thousand tokens per second, kt/s), including both prefill tokens and decode tokens.
- **Latency:** Measured as the total time required to process an entire dataset, recorded as the overall execution time for the workload. Lower latency indicates faster task completion.
- **Offload Time:** Defined as the KV cache transfer time between GPU memory and CPU memory. As part of inference latency, offload time only exists in HotPrefix, because other methods only store KV caches in GPU memory. It reflects the additional overhead introduced by HotPrefix.
- **Cache Hit Ratio:** Defined as the percentage of KV cache accesses served directly from GPU memory. A higher cache hit ratio indicates more efficient GPU memory utilization, thereby reducing inference latency.

Baselines: We compare HotPrefix with state-of-the-art LLM serving systems, which support prefix sharing:

- **vLLM [30]:** A state-of-the-art serving system for large language models, designed to reduce memory fragmentation through the PagedAttention mechanism. In addition, vLLM supports prefix sharing by caching the KV values of commonly used shared prefixes. We use vLLM version 0.6.4.post1.
- **SGLang-LRU [72]:** A high-performance LLM serving system that enables efficient KV cache reuse across requests in GPU memory. It introduces the RadixAttention mechanism to optimize memory usage and facilitate flexible prefix sharing. Memory management is handled using an LRU-based eviction policy. We use SGLang version 0.4.1.post1.
- **SGLang-LFU:** SGLang with Frequently Used(LFU) eviction policy.

- **SGLang-FIFO:** SGLang with First Come First Serve(FIFO) eviction policy.
- **SGLang-2Q:** SGLang with Two Queue(2Q) [27] eviction policy.

SGLang, its variants and vLLM only use GPU memory for KV cache storage and do not perform any offloading. For a fair comparison, all baselines are configured with the same maximum batch size and GPU memory settings as HotPrefix.

5.2 End-to-end Performance Comparison

In this section, we evaluate the end-to-end performance of HotPrefix by comparing it against SGLang and vLLM. The evaluation is conducted using the LLaMA-2 13B model, Gemma 9B model, LLaMA-3 8B model and Qwen-2 72B model on four representative workloads: 5-shot MMLU, 10-shot Hellaswag, BBH, and CEVAL. The comparison focuses on three key metrics: GPU cache hit ratio, latency, and throughput, all measured under batch execution.

In the experiments, the batch size is set to 64 for all MMLU and BBH tasks, and 32 for CEVAL and Hellaswag. Both vLLM and SGLang are tested with prefix-sharing optimizations enabled. For HotPrefix, the cuckoo filter is configured with an 8-bit length for the fingerprint, clock, frequency, and depth fields, and the *max_age* value is set to 255. The frequency threshold for the selective KV cache admission module is set to 10. The KV cache dtype we use is bfloat16 and the decoding strategy is top-p sampling at 0.95. For other decoding strategies compatibility, like parallel sampling, each sequence can locally applies top-p. For beam search, we can filter candidates with top-p.

5.2.1 *Cache Hit Ratio.*

Figure 8 illustrates the GPU cache hit ratio of SGLang, its variants, vLLM and HotPrefix across the MMLU, Hellaswag, BBH, and CEVAL workloads. HotPrefix achieves highest hit ratio across different tasks, with a 1.17-2.38× improvement over SGLang-LRU, 1.16-2.28× improvement over SGLang-LFU, 1.19-2.56× improvement over SGLang-FIFO, 1.17-2.51× improvement over SGLang-2Q and 1.11-3.27× improvement over vLLM. The highest improvement is observed on the CEVAL dataset due to its diverse and complex access patterns, which vLLM and SGLang's simple cache management strategies struggle to handle effectively. In contrast, HotPrefix adapts to these characteristics by dynamically tracking the hotness of KV cache entries and selectively retaining or offloading them based on their reuse potential.

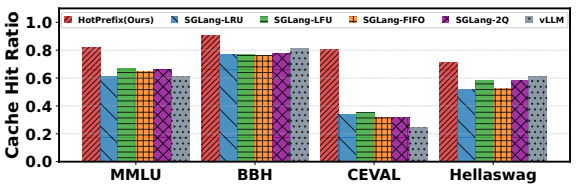

Fig. 8. The cache hit ratio of different workloads under Llama-2 13B inference model. Higher is better.

Specifically, the discrepancy arises from the limitations in the cache management strategies of SGLang and vLLM, which are unable to retain high-hotness KV caches in GPU memory effectively. SGLang employs a simple LRU eviction strategy, which often results in the premature eviction of high-hotness KV cache nodes when workloads are shuffled. And vLLM uses a coarse-grained prefix-sharing approach, where prefixes that do not align with predefined block sizes cannot be fully utilized. HotPrefix addresses these issues by employing a hotness-aware eviction strategy that prioritizes the removal of low-hotness KV caches. Furthermore, when GPU memory becomes insufficient, HotPrefix offloads evicted high-hotness KV caches to host memory and asynchronously reloads them into GPU memory when required, ensuring minimal performance degradation even under constrained memory conditions.

5.2.2 *Inference Latency and Throughput.*

Figure 9 shows the inference latency of SGLang, its variants, vLLM and HotPrefix across the MMLU, Hellaswag, BBH, and CEVAL workloads. HotPrefix improves end-to-end inference latency and throughput that is 1.55-2.00× over SGLang-LRU, 1.45-1.89× over SGLang-LFU, 1.55-1.92× over SGLang-FIFO, 1.41-1.90× over SGLang-2Q and 1.54-2.25× over vLLM. We also run the evaluation on Gemma 9B model, achieving similar inference latency performance and throughput improvement, with 1.45-1.64× over SGLang-LRU and 1.58-1.91× than vLLM. Table 1 shows that the HotPrefix method introduces offload time overheads of 25.66s, 24.8s, 68.28s, and 27.03s on the MMLU, BBH, CEVA, and Hellaswag datasets respectively. These overheads account for only 2.6%, 5.9%, 1.1%, and 2.2% of the average inference latency of other methods, yet deliver average improvements of 1.28×, 1.16×, 2.6×, and 1.27× in overall inference latency compared to other approaches.

The highest improvement is observed on the CEVAL dataset, as it features more diverse workloads and longer request lengths. HotPrefix, with its superior ability to adapt to the dynamic nature of user requests, achieves a significantly higher prefix KV cache hit ratio on this dataset. This reduces the need for KV cache recomputation, leading to the greatest improvement in latency performance and throughput compared to the other methods.

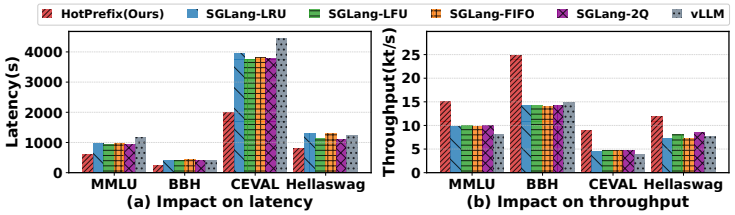

Fig. 9. The inference latency and throughput of different workloads under Llama-2 13B inference model.

Table 1. Offload time of HotPrefix

Datasets	MMLU	BBH	CEVAL	Hellaswag
Offload time (s)	25.66	24.8	68.28	27.03

Specifically, the reduction in inference latency and throughput achieved by HotPrefix primarily results from its ability to significantly increase the cache hit ratio for prefix KV caches. By selectively offloading only high-hotness KV caches to host memory, HotPrefix minimizes the adverse impact of offloading time on inference latency. Additionally, during the decode phase, HotPrefix asynchronously reloads high-hotness KV caches from host memory into GPU memory, thereby enhancing prefix cache reuse.

5.2.3 *Scalability.*

This section evaluates the scalability of HotPrefix on multi-GPUs and different kinds of GPUs. For multi-GPUs evaluation, we run tensor parallelism across 4 A6000 GPUs with Qwen-2 72B model to fully ultizing GPU resources and LLaMA-2 13B for data parallelism. For different kinds of GPUs evaluation, we use small model LLaMA-3 8B because original model LLaMA-2 13B exceed device memory of some GPUs, like A30 and RTX 3090. Figure 10 presents the cache hit ratio and inference latency of HotPrefix, SGLang and vLLM, all using tensor parallelism across 4 A6000 GPUs under Qwen-2 72B model. Under this configuration, HotPrefix also achieves the highest cache hit ratio and improves end-to-end latency that is 1.57-2.42× over SGLang and 1.60-2.61× over vLLM.

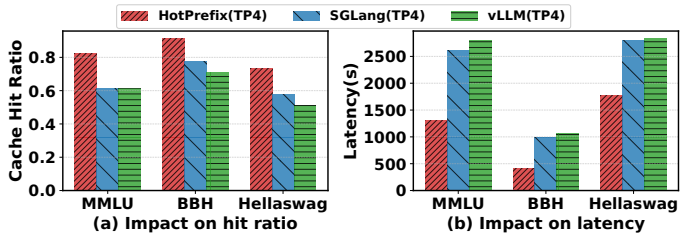

Fig. 10. The cache hit ratio and inference latency of tensor parallelism across 4 GPUs under Qwen-2 72B.

Figure 11 shows the inference latency of HotPrefix of data parallelism 1, 2, 4 under LLaMA-2 13B model on A6000s. As the number of GPUs used for data parallelism increases, the overall inference latency of HotPrefix decreases linearly, demonstrating its scalability in multi-GPU data parallel.

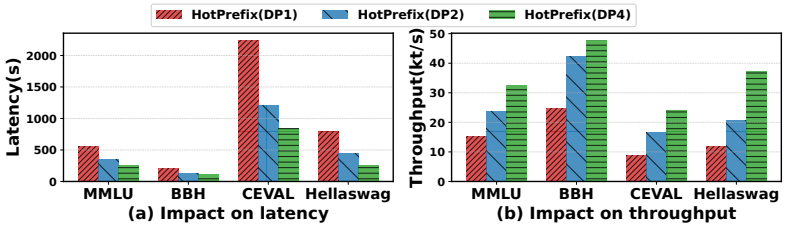

Fig. 11. The inference latency of different data parallelism configurations under LLaMA-2 13B model.

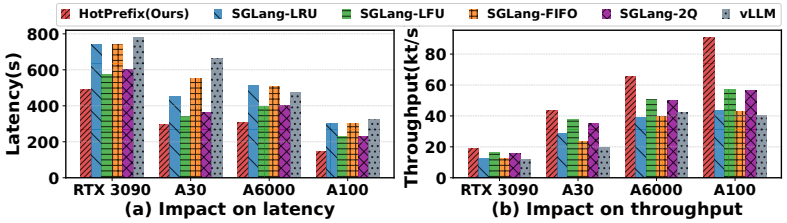

Fig. 12. The inference latency and throughput of MMLU task across different GPUs under LLaMA-3 8B model (not original LLaMA 13B because 13B model exceeds device memory of A30 and RTX 3090).

Figure 12 shows the inference latency and throughput of HotPrefix, vLLM, SGLang and its variants of different types of GPUs under MMLU dataset and LLaMA-8B model. To ensure fairness, all experiments are run on the same KV cache memory size. Different GPUs have different computing ability, but HotPrefix outperforms other methods across various GPUs, showing its robustness. Compared with other methods, Hotprefix achieves 1.06-1.53×, 1.08-1.24×, 1.10-1.28× and 1.18-1.40× cache hit ratio improvement and 1.17-1.60×, 1.16-2.25×, 1.30-1.68× and 1.58-2.27× inference latency performance improvement on RTX 3090, A30, A6000 and A100 respectively.

5.3 Ablation Studies

In this section, we present ablation studies to evaluate the individual optimization components of HotPrefix. The experiments are performed on the LLaMA-2 13B model, with the Cuckoo Filter configured to use an 8-bit length for the fingerprint, clock, frequency, and depth fields, and a *max_age* value set to 255. The batch size for MMLU, Hellaswag and CEVAL is 32, 32, 4.

5.3.1 *Selective Admission Policy.*

Figure 13 illustrates the impact of adopting the selective admission policy in HotPrefix on cache hit ratio and end-to-end latency. When the admission policy is disabled, all KV caches evicted from GPU memory are offloaded to host memory.

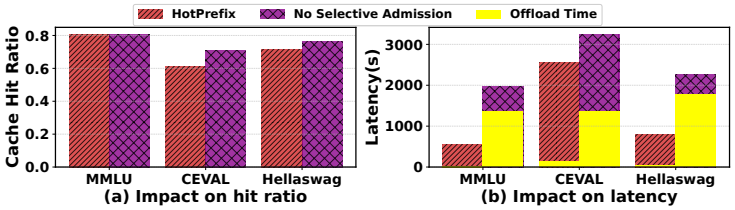

Fig. 13. The ablation study on selective admission. Yellow portion is offload time between GPU and CPU.

As shown in Figure 13(a), compared to disabling the admission policy, HotPrefix with the admission policy reduces the cache hit ratio by only 0.2% on MMLU and 10% on CEVAL and 5% on Hellaswag, but reduces 10-65× offload time. Finally, HotPrefix achieves lower inference latency than without selective admission, with a 1.28-3.54× improvement. This improvement stems from its ability to prevent the offloading of KV caches that are unlikely to be reused into host memory, greatly reducing the offload time between CPU and GPU memory. Although HotPrefix may occasionally reject certain KV caches that could be reused in the future, causing a little lower cache hit ratio, the overall advantages of Selective Admission Policy outweigh the disadvantages.

5.3.2 *Comparison of Optimization Components.*

In Figure 14, we make the comparison of all optimization components.

We combine "A" and "P" together because offloading with admission is for KV cache promotion and we have analyzed the individual impact of the Selective Admission Policy in Section 5.3.1.

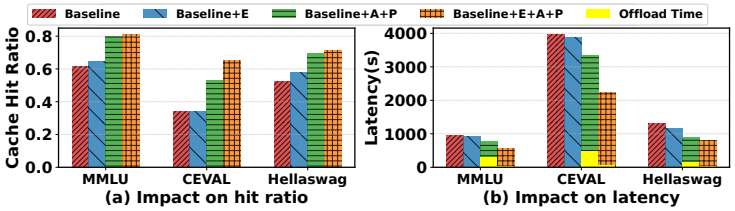

Fig. 14. The ablation study on all optimization components. The yellow portion is the offload time between GPU and CPU. The "Baseline" means SGLang without any optimization, "E" means Hotness-aware Eviction Policy, "A" means Selective Admission Policy and "P" means Hotness Promotion.

Compared with "Baseline", "Baseline+E" achieves 1.02-1.11× cache hit ratio increase and 1.02-1.44× latency performance improvement, "Baseline +A+P" achieves 1.30-1.56× cache hit ratio increase and 1.19-1.47× latency performance improvement, and "Baseline+E+A+P" achieves 1.32-1.92× cache hit ratio increase and 1.65-1.77× latency performance improvement.

Although "Baseline+E" only marginally improves upon "Baseline", "Baseline+E+A+P" outperforms "Baseline+A+P" 1.07× on cache hit ratio, 7.77× on offload time and 1.38× on inference latency. The interesting result shows that "E" alone yields modest performance gains, while the combined application of "E" and "A+P" leads to significant performance improvements. The benefits of combination of "E+A+P" lies in two aspects. First, the combined "E+A+P" together improves 1.08× cache hit ratio than "A+P". Second, with "E", HotPrefix evicts lowest hotness KV caches to CPU memory, avoiding frequent KV cache transfer between GPU and CPU and leading to 7.77× offload time improvement compared with "A+P".

5.3.3 *Hotness-aware Eviction Policy.*

We compare HotPrefix with versions of HotPrefix that use simple LRU, LFU, FIFO and 2Q as the eviction policy to evaluate the effectiveness of Hotness-aware Eviction. Figure 15 shows the cache hit ratio and inference latency for MMLU, CEVAL and Hellaswag. HotPrefix achieves the lowest inference latency, with a 1.11-1.37×, 1.07-1.89×, 1.13-1.63×, and 1.02-1.34× improvement over LRU, LFU, FIFO, and 2Q respectively. The improvement of inference latency comes from two aspects. First, HotPrefix needs least offload time between CPU and GPU KV cache transfer, because Hotness-aware Eviction evicts the lowest hotness KV caches, avoiding frequent KV cache transferring. Second, HotPrefix achieves the highest cache hit ratio with the help of Hotness-aware Eviction Policy.

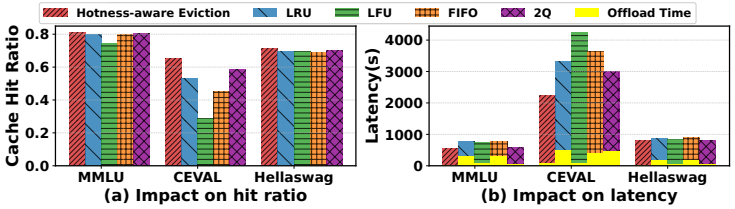

Fig. 15. The ablation study on cache eviction policy. Yellow portion is offload time between GPU and CPU.

5.3.4 *Hotness Promotion.*

In HotPrefix, during the decode phase, KV caches are asynchronously loaded from host memory to GPU memory based on their hotness.We compare HotPrefix with versions of HotPrefix that use simple recency, frequency and depth as the criterion of selecting KV caches to load from CPU memory to GPU memory to evaluate the effectiveness of Hotness Promotion criterion. The recency and frequency metrics are the clock value and frequency value in the hotness record structure described in Section 3.2.1. The depth metric is the KV cache nodes depth in the prefix tree.

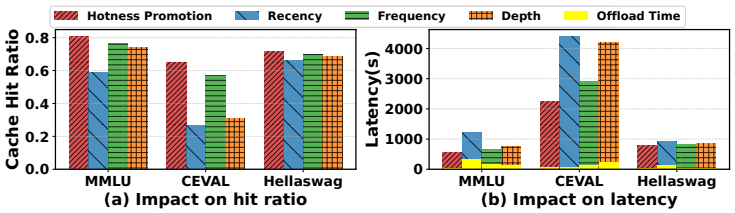

Fig. 16. The ablation study on hotness promotion. Yellow portion is offload time between GPU and CPU.

Figure 16 shows the cache hit ratio and inference latency for MMLU, CEVAL and Hellaswag. As shown in Figure 16(a), Hotness Promotion achieves the highest cache hit ratio, with a 1.13-2.57×, 1.01-1.14×, and 1.05-2.09× improvement over recency, frequency, and depth promotion respectively. Figure 16(b) demonstrates that Hotness Promotion achieves lowest inference latency, with a 1.17-2.22×, 1.04-1.30×, and 1.09-1.88× improvement over recency, frequency, and depth promotion respectively. And the offload time between GPU and CPU of Hotness Promotion is also the lowest.

6 Related Work

Caching Systems: Traditional Caching systems [8, 16, 38, 40, 45, 46] are widely used in various applications, like operation system caching, web caching and CDN caching. Operating system caching systems, like Linux page cache [38] and ZFS ARC [40], are optimized for uniform cache-line sizes, failing to address the variable computational costs of different-length KV cache misses in LLMs. Web caching systems, like Redis [8] and Memcached [16], are designed for independent

request handling with simple eviction costs, incapable of modeling prefix-tree dependencies or asymmetric CPU-GPU transfer overheads in LLM inference. CDN caching systems, like Akamai [46] and Cloudflare [45], focuse on network level efficiency, rather than computational cost, making them inherently unsuitable for managing the latency sensitive, variable computational cost KV cache management in LLM inference.

Unlike these generic solutions, HotPrefix primarily focuses on managing prefix sharing LLM inference KV cache. In the prefix tree, different KV cache nodes vary significantly in length, and the computational cost of a KV cache node miss is substantial. These factors collectively result in a wide disparity in the cost of cache misses across different KV cache nodes. Thus, a more dynamic, precise, and low-overhead cache management solution is needed to handle the KV cache in prefix sharing LLM inference.

KV Cache Management: KV caches are extensively employed to accelerate inference in large language models (LLMs). Recent advancements in this area have focused on reducing the memory footprint of KV caches through various techniques, including quantization, compression, and layer sharing. Quantization methods [11, 22, 29, 37, 44, 48, 62, 69], aim to compress KV caches by reducing their bit precision, from 16-bit to as low as 4-bit or even lower. For example, CacheGen [13] combines KV offloading with quantization-based compression to reduce the overhead of transferring KV caches between GPU and host memory.

Compression techniques [3, 18, 34, 36, 59, 61, 70] exploit the sparsity in attention layers, retaining only the most important KV cache entries in GPU memory while discarding less significant ones. Representative works include [1, 5, 13, 26], which aim to optimize memory usage without significantly impacting model performance.

Unlike these methods, which may compromise the model's prediction accuracy, HotPrefix adopts a different approach by utilizing both GPU memory and host memory to retain high-hotness prefix KV caches. This strategy maximizes KV cache reuse without sacrificing inference accuracy.

KV Cache Reuse: KV cache reuse [1, 13, 15, 17, 19, 25, 30, 35, 66, 67, 72] has been extensively studied to enhance inference efficiency by reducing redundant computations. Among these, vLLM [30] reuses KV caches by reserving fixed physical memory blocks for shared prefixes. While this mechanism simplifies memory allocation, it suffers from coarse granularity. If the prefix length does not align with block sizes, some parts of the prefix cannot be effectively reused, leading to inefficiencies. SGLang [72] organizes KV caches into a radix tree using RadixAttention to maximize reuse. This structure facilitates prefix sharing. However, its reliance solely on GPU memory limits scalability. Additionally, SGLang employs a simple LRU eviction strategy, which struggles to adapt to dynamic user requests. Notably, SGLang's original experiments [72] benefited from unshuffled datasets, which allowed its radix tree structure to maximize prefix reuse. However, shuffled workloads, which better represent real-world user access patterns, expose its limitations in adapting to dynamic and less predictable prefix reuse scenarios. In these cases, its performance converges with vLLM. CachedAttention [17] extends KV cache storage across disk, host memory, and GPU memory, primarily targeting multi-turn conversation scenarios. However, it heavily depends on job queue information to prefetch KV caches. In dynamic user environments, such as when new requests arrive after the previous ones are processed, this approach becomes ineffective due to the lack of proactive adaptation. PromptCache [19] proposes modular reuse beyond prefixes but sacrifices accuracy, with reported performance drops of up to 43% [72], making it unsuitable for tasks requiring high precision.

HotPrefix addresses the limitations of these methods by introducing a hotness-aware KV cache scheduling framework that dynamically evaluates the reuse probabilities of prefix KV caches. Unlike vLLM's fixed block allocation or SGLang's GPU-only strategy, HotPrefix leverages both GPU and host memory collaboratively. By dynamically tracking prefix hotness, it employs hotness-aware

eviction and asynchronous promotion mechanisms to adapt to user request patterns. HotPrefix utilizes fine-grained hotness evaluation and KV cache promotion to achieve higher reuse rates and improved performance , addressing the inefficiencies inherent in vLLM's coarse-grained prefix sharing. Furthermore, compared with SGLang, HotPrefix demonstrates robustness even under shuffled workloads, which reflect more realistic user access patterns, highlighting the adaptability of HotPrefix in handling diverse and unpredictable request scenarios.

7 Conclusion and Future Work

This paper presents HotPrefix, a novel hotness-aware KV cache scheduling framework designed to maximize the reuse of prefix KV caches during LLM inference. HotPrefix reduces end-to-end inference latency and increases throughput by incorporating optimizations, including hotness tracking, hotness-aware eviction strategy, selective admission policy, and hotness promotion. Extensive experimental results demonstrate that HotPrefix significantly enhances LLM inference efficiency, achieving up to a 2.25× reduction in inference latency and increase in throughput compared to vLLM with prefix sharing enabled. Similarly, compared to SGLang, HotPrefix delivers up to a 2× reduction in latency and improvement in throughput.

For future work, we plan to extend HotPrefix to support multi-modal inference. By addressing the complex KV cache dependencies shared across text, image, and other modalities, this extension could enable more efficient multi-modal applications.

Acknowledgments

We sincerely thank the reviewers for their time and expertise in the review process. This work is funded in part by Nanjing "U35" Talent Cultivation Program (No. U (2024) 001), the National Natural Science Foundation of China(NO.62325205, 62072230), Postgraduate Practice & Research Innovation Program of Jiangsu Province (NJU KYCX24_0257), Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University. Rong Gu is the corresponding author of this paper.

References

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. APIServe: Efficient API Support for Large-Language Model Inferencing. In *arXiv preprint arXiv:2402.01869*.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. In *arXiv preprint arXiv:2303.08774*.

[3] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. In *Proceedings of Machine Learning and Systems (MLSys)*. 114–127.

[4] Anthropic. 2025. https://www.anthropic.com

[5] Appleby and Austin. 2008. https://github.com/aappleby/smhasher

[6] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. In *arXiv preprint arXiv:2401.02954*.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 1877–1901.

[8] Josiah L. Carlson. 2013. *Redis in Action*.

[9] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 17754–17762.

[10] Jiaxi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. 2023. Chatlaw: Open-source legal large language model with integrated external knowledge bases. In *arXiv preprint arXiv:2306.16092*.

[11] Shichen Dong, Wen Cheng, Jiayu Qin, and Wei Wang. 2024. QAQ: Quality Adaptive Quantization for LLM KV Cache. In *arXiv preprint arXiv:2403.04643*.
[12] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. In *arXiv preprint arXiv:2407.21783*.
[13] Yuhan Liu et al. 2024. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of ACM SIGCOMM Conference (SIGCOMM)*. 38–56.
[14] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 75–88.
[15] Yuan Feng, Hyeran Jeon, Filip Blagojevic, Cyril Guyot, Qing Li, and Dong Li. 2023. AttMEMO: Accelerating Transformers with Memoization on Big Memory Systems. In *arXiv preprint arXiv:2301.09262*.
[16] Brad Fitzpatrick. 2003. Memcached Official Documentation. https://memcached.org
[17] 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 *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 111–126.
[18] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model tells you what to discard: Adaptive kv cache compression for llms. In *arXiv preprint arXiv:2310.01801*.
[19] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems (MLSys)*. 325–338.
[20] Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. 2021. Transformer in transformer. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 15908–15919.
[21] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *arXiv preprint arXiv:2009.03300*.
[22] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. In *arXiv preprint arXiv:2401.18079*.
[23] Yuzhen Huang, Yuzhuo Bai, Zhihao Zhu, Junlei Zhang, Jinghan Zhang, Tangjun Su, Junteng Liu, Chuancheng Lv, Yikai Zhang, Jiayi Lei, Yao Fu, Maosong Sun, and Junxian He. 2023. C-Eval: A Multi-Level Multi-Discipline Chinese Evaluation Suite for Foundation Models. In *arXiv preprint arXiv:2305.08322*.
[24] Chaoyi Jiang, Lei Gao, Hossein Entezari Zarch, and Murali Annavaram. 2024. Efficient llm inference with i/o-aware partial kv cache recomputation. In *arXiv preprint arXiv:2411.17089*.
[25] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. In *arXiv preprint arXiv:2404.12457*.
[26] Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Z Morley Mao. 2024. Compute or load kv cache? why not both?. In *arXiv preprint arXiv:2410.03065*.
[27] Theodore Johnson, Dennis Shasha, et al. 1994. 2Q: a low overhead high performance bu er management replacement algorithm. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 439–450.
[28] jujumilk3. Collection of leaked system prompts. 2023. https://github.com/jujumilk3/leaked-system-prompts.
[29] Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. 2024. Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm. In *arXiv preprint arXiv:2403.05527*.
[30] 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.
[31] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *proceedings of USENIX Symposium on Operating Systems Design and Implementation (SOSP)*. 155–172.
[32] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 9459–9474.
[33] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023. Chain of code: Reasoning with a language model-augmented code emulator. In *arXiv preprint arXiv:2312.04474*.
[34] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation. In *arXiv preprint arXiv:2404.14469*.
[35] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. 2024. Optimizing llm queries in relational workloads. In *arXiv preprint arXiv:2403.05821*.

[36] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2024. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 52342–52364.
[37] 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 *arXiv preprint arXiv:2402.02750*.
[38] Robert Love. 2010. *Linux kernel development*.
[39] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 43447–43478.
[40] Nimrod Megiddo and Dharmendra S Modha. 2004. Outperforming LRU with an adaptive replacement cache algorithm. *IEEE Computer* 37, 4 (2004), 58–65.
[41] Yuhong Mo, Hao Qin, Yushan Dong, Ziyi Zhu, and Zhenglin Li. 2024. Large language model (llm) ai text generation detection based on transformer deep learning algorithm. In *arXiv preprint arXiv:2405.06652*.
[42] Arbi Haza Nasution and Aytug Onan. 2024. ChatGPT Label: Comparing the Quality of Human-Generated and LLM-Generated Annotations in Low-resource Language NLP Tasks. *IEEE Access* 12 (2024), 71876–71900.
[43] FranxYao. Collection of chain-of-thought prompts. 2023. https://github.com/FranxYao/chain-of-thought-hub
[44] OpenAI. 2025. https://openai.com/blog/chatgpt
[45] Matthew Prince. 2012. Cloudflare Architecture. https://www.cloudflare.com/
[46] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K Sitaraman. 2020. Akamai dns: Providing authoritative answers to the world's queries. In *Proceedings of ACM SIGCOMM Conference (SIGCOMM)*. 465–478.
[47] Jia Tracy Shen, Michiharu Yamashita, Ethan Prihar, Neil Heffernan, Xintao Wu, Ben Graff, and Dongwon Lee. 2021. Mathbert: A pre-trained language model for general nlp tasks in mathematics education. In *arXiv preprint arXiv:2106.07340*.
[48] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *Proceedings of International Conference on Machine Learning (ICML)*. 31094–31116.
[49] Shamane Siriwardhana, Rivindu Weerasekera, Elliott Wen, Tharindu Kaluarachchi, Rajib Rana, and Suranga Nanayakkara. 2023. Improving the domain adaptation of retrieval augmented generation (RAG) models for open domain question answering. *Transactions of the Association for Computational Linguistics (TACL)* 11 (2023), 1–17.
[50] Ethan Steinberg, Ken Jung, Jason A Fries, Conor K Corbin, Stephen R Pfohl, and Nigam H Shah. 2021. Language models are an effective representation learning technique for electronic health record data. *Journal of biomedical informatics (JBI)* 113 (2021), 103637.
[51] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, , and Jason Wei. 2022. Challenging BIG-Bench Tasks and Whether Chain-of-Thought Can Solve Them. In *arXiv preprint arXiv:2210.09261*.
[52] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. In *arXiv preprint arXiv:2403.08295*.
[53] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. In *arXiv preprint arXiv:2307.09288*.
[54] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. In *arXiv preprint arXiv:2307.09288*.
[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *arXiv preprint arXiv:1706.03762*.
[56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 24824–24837.
[57] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. In *arXiv preprint arXiv:2302.11382*.
[58] xAI. 2025. https://x.ai/
[59] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. In *arXiv preprint arXiv:2309.17453*.
[60] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, and Chengpeng Li. 2024. Qwen2 Technical Report. In *arXiv preprint arXiv:2407.10671*.

[61] Dongjie Yang, XiaoDong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. 2024. PyramidInfer: Pyramid KV Cache Compression for High-throughput LLM Inference. In *arXiv preprint arXiv:2405.12532*.

[62] June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. 2024. No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization. In *arXiv preprint arXiv:2402.18096*.

[63] Sherry Yang, Ofir Nachum, Yilun Du, Jason Wei, Pieter Abbeel, and Dale Schuurmans. 2023. Foundation models for decision making: Problems, methods, and opportunities. In *arXiv preprint arXiv:2303.04129*.

[64] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 11809–11822.

[65] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *arXiv preprint arXiv:2210.03629*.

[66] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. In *arXiv preprint arXiv:2402.15220*.

[67] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2023. Stateful large language model serving with pensieve. In *arXiv preprint arXiv:2312.05516*.

[68] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence?. In *arXiv preprint arXiv:1905.07830*.

[69] Tianyi Zhang, Jonah Yi, Zhaozhuo Xu, and Anshumali Shrivastava. 2024. KV Cache is 1 Bit Per Channel: Efficient Large Language Model Inference with Coupled Quantization. In *arXiv preprint arXiv:2405.03917*.

[70] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 34661–34710.

[71] Xufeng Zhao, Mengdi Li, Wenhao Lu, Cornelius Weber, Jae Hee Lee, Kun Chu, and Stefan Wermter. 2023. Enhancing zero-shot chain-of-thought reasoning in large language models through logic. In *arXiv preprint arXiv:2309.13339*.

[72] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. In *arXiv preprint arXiv:2312.07104*.

Received January 2025; revised April 2025; accepted May 2025