

LLM KV-cache: To Restore or To Recompute, That Is the Question

Amirhossein Najafizadeh

Computer Science Department
Stony Brook University
Stony Brook, NY, USA
anajafizadeh@cs.stonybrook.edu

Vasily Tarasov

IBM Research
San Jose, CA, USA
vtarasov@us.ibm.com

Alex Merenstein

IBM Research
Yorktown Heights, NY, USA
alex.merenstein@ibm.com

Yue Zhu

IBM Research
Yorktown Heights, NY, USA
yue.zhu@ibm.com

Erez Zadok

Computer Science Department
Stony Brook University
Stony Brook, NY, USA
ezk@cs.stonybrook.edu

Abstract

Key-Value (KV)-caches are essential to modern Large Language Model (LLM) inference, transforming inference from a computationally prohibitive task into a practical one. However, the limited capacity of GPU High Bandwidth Memory (HBM) is often insufficient to retain KV-cache data for all active requests, especially when large models and long contexts are served. To address this limitation, modern inference systems support offloading KV-cache data from HBM to CPU DRAM, local, or remote storage. A known issue is the trade-off between restoring offloaded KV-cache data versus recomputing; the right policy is non-trivial and depends on dynamic factors such as storage bandwidth, interconnect performance, workload characteristics, context length, and service-level objectives (SLOs). In this paper, we first investigate the challenges of the restoration-recomputation trade-off and quantify its impact on inference performance when left unaddressed. We then present an I/O-aware KV-cache management policy that dynamically navigates this trade-off. Our approach maximizes inference throughput while satisfying service-level objectives by either restoring or recomputing KV-cache blocks based on the performance characteristics of the GPU and storage tiers. Initial theoretical results show a 10× improvement in performance over simplistic static policies while maintaining SLO compliance.

CCS Concepts

• **Information systems** → **Hierarchical storage management**; Distributed storage; • **Computing methodologies** → *Natural language processing*; • **Software and its engineering** → *Secondary storage*.

Keywords

LLM inference, KV-cache management, storage hierarchy, I/O-aware caching, restore-versus-recompute

ACM Reference Format:

Amirhossein Najafizadeh, Vasily Tarasov, Alex Merenstein, Yue Zhu, and Erez Zadok. 2026. LLM KV-cache: To Restore or To Recompute, That Is the Question. In *18th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '26)*, September 28–29, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3837053.3837336>

1 Introduction

Large Language Model (LLM) inference is a rapidly growing, important workload [27]. It relies on large matrix multiplications, typically executed on GPUs, to generate output tokens from the input context [11, 18]. Despite hardware acceleration, inference latency—or Time To First Token (TTFT)—still increases quadratically with context length [52]. For example, the NVIDIA H200 GPU requires 18 seconds to produce the first token for the context length of 128K tokens using Llama-3.1-70B [8, 17].

To reduce this overhead, prior work has proposed caching computation results [7, 10, 42, 56–59, 65]. These results are stored as Key-Value (KV)-cache data. Despite its name, a KV-cache is not a traditional key-value store. Instead, it contains the key and value vectors generated by transformer attention layers for previously processed tokens [36, 50]. The key and value vectors are stored together and reused by future



This work is licensed under a Creative Commons Attribution 4.0 International License.

HotStorage '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2902-7/26/09

<https://doi.org/10.1145/3837053.3837336>

attention computations, avoiding redundant recomputation. A KV-cache effectively reduces computational costs from $O(N^2)$ to $O(N)$, as vectors for prior tokens are retrieved from the cache rather than recomputed. For example, with KV-cache reuse enabled, the H200 GPU can produce the first token for a 128K-token context in less than 2 seconds [19].

Traditionally, KV-cache data resides in the GPU’s High Bandwidth Memory (HBM) [2, 24, 40, 63]. With many users and longer context lengths, the KV-cache data can quickly exceed GPU memory space. For example, a 128K-token context on Llama-3.1-70B requires roughly 30GB of KV-cache. With only 10 users, KV-cache demand exceeds 300GB, far beyond the HBM capacity of a single GPU (e.g., H200 HBM capacity is 141GB). Moreover, a significant portion of GPU memory must be reserved for ever-increasing number of model weights, further reducing the amount of space available for KV-cache data.

To address these limitations, KV-cache offloading to multi-tier storage has emerged as a key solution [6, 9, 12, 30, 43, 47, 56, 62]. Inference engines often serve multiple users (or agents) concurrently. Each user typically creates a session and interacts with the system by submitting requests. After some time, inactive user sessions’ KV-cache data may be evicted from GPU memory to lower storage tiers (e.g., CPU DRAM, local storage, remote storage) to make room for active users. As the number of users and session context lengths increase, the likelihood of eviction to lower storage tiers grows. Consequently, when inactive users return, restoration of evicted KV-cache data becomes more frequent across all tiers.

Depending on which tier has the required KV-cache data and hardware characteristics, recomputing keys and values for a request could be faster than retrieving the data from KV-cache storage. The fundamental question then is: when to restore vs. when to recompute? The large configuration space of storage tiers and hardware options makes it difficult to design a unified static solution that generalizes across deployments [13, 14].

KV-cache offloading involves multiple hardware components, including interconnection links, storage devices, and memory tiers. Thus, selecting restoration and recomputation thresholds (based on time or number of tokens) often requires deployment-specific benchmarking [4, 20, 21, 44], which is time-consuming, expensive, and can quickly become suboptimal as workloads and system conditions change [1]. Furthermore, the restoration-recomputation trade-off is inherently dynamic due to evolving data access patterns, unpredictable data distributions, and varying service-level objectives (SLOs). These factors make it increasingly challenging to determine when KV-cache data should be restored from storage or recomputed on the GPU.

In this paper, we introduce a new approach that dynamically switches between recomputation and restoration based on the current system state. At the core of our design is an I/O-aware KV-cache management policy that makes decisions based on service-level objectives (SLOs) requests and real-time observations of GPU and storage performance. Unlike existing approaches, our prototype simulator enforces a cache policy that continuously adapts to changing SLO requirements and the observed behavior of the underlying storage systems, enabling more accurate and responsive decision-making. Our early experiments demonstrate over 10× improvement over existing static policies.

2 Background

KV-cache Offloading. During LLM inference, the KV-cache is stored as blocks of data, where each block holds the key and value vectors for a fixed number of tokens (e.g., 16 or 32). To process a token, the model must read its relevant KV-cache blocks. Rather than copying these blocks into a contiguous buffer, the inference engine reads them in place (i.e., zero-copy): a block table maps each block to its physical location in GPU memory, and the model reads each block directly from that location. Consequently, the relevant KV-cache blocks must be resident in GPU memory before they can be used [24]. Modern cache systems [30, 32] offload KV-cache blocks into external storage tiers, including CPU DRAM, local storage devices (e.g., SSD), and remote storage systems (e.g., S3). Inference engines [24, 63] integrate with cache systems through pluggable connectors [43, 51], which coordinate data placement and retrieval. Higher tiers provide greater throughput but are expensive and capacity-constrained, whereas lower tiers offer larger capacity at lower throughput (e.g., 141GB of HBM provides up to 3,360GB/s of bandwidth, whereas terabytes of NVMe Gen5 storage provide roughly 10GB/s). Consequently, inference performance depends strongly on the underlying hardware. In this exploratory work, we consider a hierarchy consisting of GPU HBM, single-node CPU DRAM, and a local SSD connected through a single interconnect (e.g., PCIe); more complex hierarchies are left to future work.

I/O Overhead. KV-cache offloading is the main source of I/O overhead during inference [30]. Cache systems transfer KV-cache blocks between storage tiers and track their location so that the correct blocks can be returned to GPU memory when the inference engine requests them. The three main KV-cache operations are LOOKUP, STORE, and RESTORE [30, 51]. Lookup operations access only KV-cache metadata, which is small enough to reside in CPU DRAM and therefore incurs negligible I/O overhead. Store operations execute asynchronously and hence have limited impact

on performance. Restore operations are the most latency-critical. Although restore operations are asynchronous from the cache’s perspective, they are effectively blocking from the inference perspective, as requests must wait for the required KV-cache data before proceeding. As a result, restore operations introduce queueing delays attributable to I/O overhead [16]. The magnitude of this overhead depends on factors such as the hardware characteristics, data distribution, and access patterns, all of which influence effective I/O performance [33, 55].

Service Level Objectives. The I/O overhead caused by KV-cache offloading indirectly impacts inference performance. The magnitude of this impact depends on the storage tier a block is fetched from: the slowest tiers (e.g., remote disks) degrade performance the most, whereas faster tiers (e.g., CPU DRAM) cause less impact. Inference services are typically evaluated using a variety of service-level objectives (SLOs), including time-to-first-token (TTFT), time-to-last-token (TTLT), and service throughput (measured in requests or tokens per second). In this work, we focus on two general QoS metrics: service throughput and tail latency. Service throughput is defined as the rate of requests completed per unit of time and is primarily influenced by the computational capacity of the GPU. Tail latency is defined as the 95th-percentile (P95) of prompt-processing phase (prefill) latency, representing the latency experienced by the slowest 5% of requests during input processing. We focus on prefill latency because restoration and recomputation affect only this stage, while token-generation phase (decode) latency is assumed to be unchanged across policies. Beyond the SLOs above, other objectives such as energy consumption, financial costs, and data durability are also relevant, but they are outside the scope of this work.

Related Work. Existing solutions employ a range of optimizations to reduce restoration overhead. At the hardware and system level, these include high-speed interconnections [25, 26] and GPU direct-access methods [35, 37, 38]. At the application level, optimizations include request scheduling across multiple inference engines [22, 30], divide-and-conquer parallelism of inference steps [5, 34, 54, 64], model and KV-cache quantization [23, 28, 29], and data compression [31, 46, 47] to reduce KV-cache size. Other systems navigate the restoration-recomputation trade-off using tunable thresholds and storage tier selection policies [30, 53, 55], context importance-aware offloading policies [48], or model-weight offloading techniques to free GPU memory for KV-cache data [61]. As such, our approach is orthogonal to existing optimization techniques and can leverage improvements from faster storage, more efficient offloading, compression, and quantization methods while still making restoration-versus-recomputation decisions.

3 Problem Statement

Tiered Caching Simulation. To understand how hardware characteristics impact KV-cache offloading performance, we developed a performance-modeling simulator that models the GPU compute speed, the storage tiers (GPU HBM, CPU DRAM, and local or remote disk), and the transfer links between them. Using this model, we analyzed the restoration and recomputation times across 1,764 hardware configurations. The simulator is implemented in approximately 1,000 lines of Python and Go code.¹

The simulator assumes that KV-cache data is restored from higher memory tiers until their capacity is exhausted; thereafter, restoration falls back to lower storage tiers. This approach approximates typical cache systems’ behavior [30]. To make the simulated model tractable, we omit complex mechanisms such as prefetching, inter-tier data movement, and random-access patterns. The simulator models the time required to restore N KV-cache blocks during inference with a tiered storage hierarchy of one HBM, one DRAM, and one disk. To simplify the analysis, we assume that only model weights occupy the HBM, and that the remaining capacity is fully available for KV-cache space. After reserving space for weights (e.g., $2P$ bytes for bf16 precision, where P is the parameter count), the remaining effective HBM capacity is

$$\text{Effective Capacity}_{\text{HBM}} = \text{Capacity}_{\text{HBM}} - 2P$$

which determines how many blocks fit on HBM:

$$\text{KVC Space}_{\text{HBM}} = \left\lfloor \frac{\text{Effective Capacity}_{\text{HBM}}}{S_{\text{block}}} \right\rfloor,$$

where S_{block} is the byte size of one KV block. The KV-cache block size is derived from the model architecture. For a Llama-3 model with 16-token blocks, each block occupies approximately 2MB.

The capacity of each tier determines how many KV-cache blocks can be stored before spilling to the next tier. The KV-cache restoration time is computed piece-wise across the hierarchy from HBM to disk based on the transferred data volume, its capacity, and effective bandwidth (β_{tier}). No restoration is required when the data resides entirely in HBM. For DRAM and disk, restoration throughput is limited by the slower of the storage-tier bandwidth and the CPU-GPU interconnect bandwidth, resulting in effective bandwidths of $\min(\beta_{\text{dram}}, \beta_{\text{link}})$ and $\min(\beta_{\text{disk}}, \beta_{\text{link}})$, respectively.

The compute time is modeled using a roofline-based approximation that combines the model’s computational cost with the GPU effective compute throughput [41, 60]. Because processing each block requires attending to all preceding blocks, processing N blocks is modeled as:

$$t_{\text{compute}} = N^2 \cdot T_{\text{block}} \cdot \hat{t},$$

¹<https://github.com/sbu-fsl/KVC-TierSim>

where T_{block} is the number of tokens per block and $\hat{t}$ is the per-token compute time. We estimate $\hat{t}$ as:

$$\hat{t} = \frac{F_{\text{token}} (\approx 2P)}{\Phi_{\text{peak}} \cdot \eta},$$

where F_{token} is the number of FLOPs required to process one token for the target LLM, Φ_{peak} is the peak FLOP/s of each GPU, and η is the hardware utilization efficiency, or MFU [8]. Therefore, larger models increase compute time through F_{token} , while different GPUs affect compute time through Φ_{peak} and η . In this model, we assume that $\eta = 1$, corresponding to perfect GPU utilization. All other hardware specifications, including device bandwidths and capacities, were collected from NVIDIA [39], FS [15], Bizon-Tech [3], and UserBenchmark [49].

In our simulated experiments, we assume a request’s entire context is already stored as KV-cache in the storage hierarchy, and we use the equations described earlier to measure the time to transfer all of the necessary KV-cache entries into GPU memory so the context is available for processing. Figure 1 presents the results of this theoretical analysis. Each (gray) violin plot summarizes restoration times across 1,764 hardware configurations formed from combinations of 6 GPUs, 6 PCIe interconnects, 7 DRAM configurations, and 7 storage devices for a given session context length. The wide spread within each violin shows that, even at a fixed context length, hardware selection alone can change restoration time by 2–3 orders of magnitude. At smaller context lengths, restoration times exhibit substantial variation because the KV-cache data may fit entirely within GPU HBM or CPU DRAM, depending on the hardware configuration. As the context length exceeds the effective HBM capacity, restoration performance becomes increasingly constrained by disk bandwidth and capacity. In contrast, recomputation times remain independent of the storage configuration because they are determined by the model architecture and GPU compute capability. These results indicate substantial variation in restoration performance across hardware configurations and context lengths. For a given context length, multiple hardware configurations may satisfy the SLO requirements. Furthermore, the recomputation curves for nearly all GPUs (except the H200) intersect the restoration-time distributions. This suggests that, depending on the storage and GPU configuration, either recomputation or restoration may provide better performance (*i.e.*, a trade-off). These observations highlight the need for adaptive policies that dynamically balance recomputation and restoration based on hardware characteristics, context length, and application SLOs.

Case Study. As a motivating case study, we evaluated two GPUs, an H200 and an RTX A5000, together with a SATA

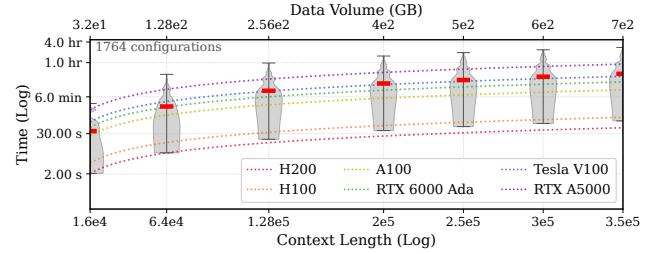


Figure 1: Performance modeling results for recomputation and restoration across 1,764 hardware configurations. Dashed lines show recomputation times for different GPUs; violin plots show restoration-time distributions (red lines denote medians). The x-axes represent session context length and corresponding KV-cache size; the y-axis shows completion time.

SSD, to examine the impact of restoration and recomputation on inference throughput and tail latency. Our analysis shows that the impact of restoration depends on the storage tiers and data-transfer paths. Restoration can increase end-to-end inference latency by up to 10× on high-performance GPUs, whereas it becomes nearly equivalent to recomputation on lower-performance GPUs. To serve the same model on both GPUs, we evaluated KV-cache restoration using the Llama-3.1-8B-Instruct model under constrained cache capacities (2GB HBM and 1GB DRAM) to force KV-cache offloading to disk. We executed twice a ShareGPT-based [45] chatbot comprising 200 sessions (~5 requests/session and ~1,000 input tokens/request): first to populate the cache hierarchy and trigger KV-cache evictions, and then to replay the same sessions and measure KV-cache restoration. During execution, we periodically sampled the system as sessions accumulated, increasing the aggregate context length from approximately 1.6×10^3 to 1.6×10^6 tokens. The reported context lengths represent the total KV-cache footprint across all sessions rather than the length of any individual conversation.

Figure 2 presents the results of this experiment. For the H200, restoration latency remains low at small context lengths because KV-cache data can be retrieved from GPU HBM. In contrast, the A5000 does not benefit from the same behavior and exhibits higher restoration latency. As context length increases, more KV-cache data resides on disk, causing restoration latency to grow substantially, and even reach minutes or hours. In contrast, recomputation latency is primarily determined by GPU compute capability, remaining relatively low on the H200 while increasing on the A5000. These results show that storage performance can become a major bottleneck for long-context inference, making recomputation a more attractive alternative on GPUs with high

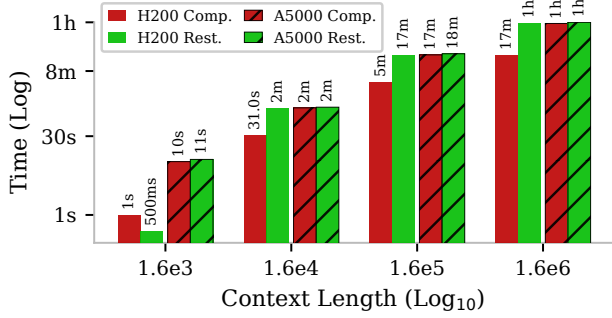


Figure 2: Processing time of different context lengths when the data is computed on GPU versus when it is restored from SSD.

compute throughput. In contrast to the H200, the lower compute throughput of the A5000 makes KV-cache restoration and recomputation comparable in execution time, making both viable alternatives that produce the same outputs.

4 Policy Design

In this work, we propose an *IO-aware* KV-cache management policy that adaptively decides whether KV-cache data should be restored from the storage hierarchy or recomputed. The policy is applied only during restoration operations, while the cache system always attempts to preserve KV-cache entries during normal execution.

We assume GPU-resident blocks require no restoration and that CPU DRAM is used only as a transient transfer buffer. In this work, we apply the policy at the disk tier, where its impact is most pronounced and can be most clearly observed; extending it to CPU DRAM is left as future work. Upon a restore request, the cache system uses metadata to identify cached and missing KV-cache blocks. Missing blocks are recomputed. For blocks that have been evicted to the disk tier, the cache system must decide whether each block should be restored from disk or recomputed on the GPU. The goal of the policy is to balance the workload between storage restoration and GPU recomputation while avoiding resource over-utilization and request queuing.

For the remainder of this section, we assume a GPU recomputation throughput of X bytes/s, a storage restoration throughput of Y bytes/s, a target request rate of r requests/s, a P95 latency target of p seconds, a restore request containing N KV-cache blocks of size B bytes each, and M cache-miss blocks that must be recomputed because they are unavailable in the storage hierarchy.

Since restoration and recomputation can proceed in parallel, the request completion time is determined by the slower

of the two paths. Of the N requested blocks, M must be recomputed unconditionally because they are cache misses. We let k additional cache-hit blocks be reassigned to recomputation, so recomputation handles $N_c = M + k$ blocks and restoration handles $N_s = N - M - k$. Because the two paths run in parallel, their maximum completion time is minimized when the paths finish together; setting $T_c = T_s$ and solving for k yields $k = \frac{NX}{X+Y} - M$, which we clamp to the feasible range $0 \leq k \leq N - M$. Therefore, the optimal allocation is obtained by balancing the restoration and recomputation workloads such that their completion times are equal:

$$k = \max \left(0, \min \left(N - M, \frac{NX}{X + Y} - M \right) \right),$$

where k denotes the number of cache-hit blocks reassigned from restoration to recomputation. After computing the optimal value of k , the policy derives the number of blocks to recompute ($N_c = M + k$) and restore ($N_s = N - M - k$). The resulting allocation is then validated against the throughput and tail-latency objectives separately. The recomputation time, restoration time, and request completion time are defined as

$$T_c = \frac{N_c B}{X}, \quad T_s = \frac{N_s B}{Y}, \quad T_{\text{req}} = \max(T_c, T_s).$$

The tail-latency constraint requires the request completion time to remain below the target P95 latency $T_{\text{req}} \leq p$. The throughput constraint is modeled as a resource-utilization constraint. At a target request rate of r requests/s, the recomputation workload must not overload the GPU, and the restoration workload must not overload the storage tier, $rT_c \leq 1$ and $rT_s \leq 1$. If all constraints are satisfied, the allocation is accepted. Otherwise, the request cannot simultaneously satisfy the throughput and tail-latency objectives under the current storage and compute capabilities. In such cases, additional resources, a relaxed SLO, or a different cache placement strategy may be required. The policy then operates in a best-effort mode by selecting the allocation that minimizes the maximum constraint violation.

5 Evaluation

Results. To evaluate the proposed policy, we used the simulator described in Section 3 and measured the success rate of satisfying both throughput and tail-latency objectives. Figure 3 compares the proposed policy against restoration and recomputation across six GPU models, ordered from the slowest (a) to the fastest (f). For each evaluated configuration, we determine whether both objectives are satisfied. The success rate is defined as the fraction of configurations that meet both objectives; higher success rates indicate more effective approaches.

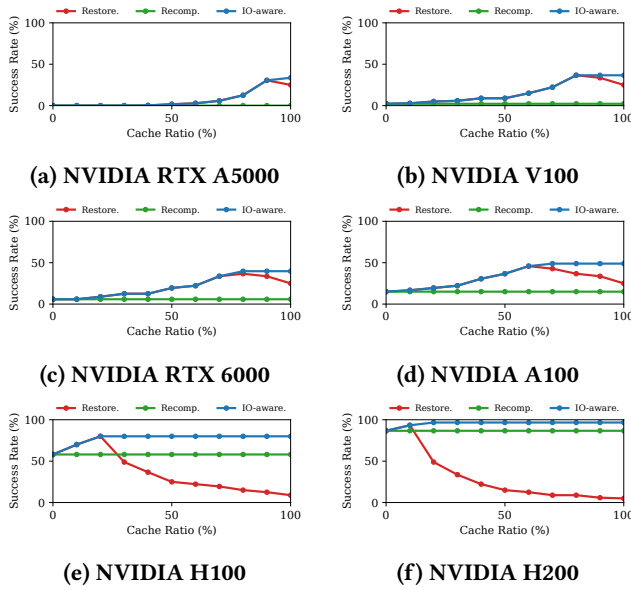


Figure 3: Effectiveness of the proposed policy compared with restoration and recomputation across cache-ratio configurations, ordered from slower to faster GPUs. Success rate represents the fraction of scenarios that satisfy both throughput and tail-latency objectives; higher values are better.

Figure 3e illustrates the trade-off between restoration and recomputation across different cache-hit ratios. The cache-hit ratio represents the fraction of KV-cache blocks that can be restored from storage. For example, a 25% cache-hit ratio means that 25% of the required KV-cache blocks are restored, while the remaining 75% are recomputed. As shown, when the cache-hit ratio is below 25%, restoration (red line) improves the success rate by up to 25% compared to recomputation (green line). However, as the cache-hit ratio increases, a larger fraction of KV-cache blocks is restored, increasing restoration overhead and reducing its effectiveness. In contrast, the proposed I/O-aware KV-cache management policy (blue line) adapts to these changing conditions and maintains up to a 10× improvement in effectiveness by selectively leveraging the GPU’s computational capability when recomputation is more efficient than restoration. Figure 3f demonstrates that this transition occurs at lower cache-hit ratios for high-performance GPUs such as the H200, further increasing the effectiveness of our proposed policy.

As shown in Figures 3a–d, slower GPUs often struggle to satisfy both objectives because of their limited recomputation throughput, causing recomputation success rates to remain below 20% and restoration success rates below 40%. Nevertheless, the blue line (our policy) consistently achieves the highest success rate by dynamically selecting

between restoration and recomputation. As the cache-hit ratio increases, the policy can leverage more restoration opportunities, further improving its effectiveness across most evaluated configurations.

Overhead Analysis. We measured the cost of the decision itself over 20.2 million timed scheduling decisions in our simulator. The proposed policy averages $5.07\mu\text{s}$ per decision. The scheduler sustains $\approx 2 \times 10^5$ decisions/s; concretely, scheduling 10^6 real-world requests costs $\approx 5.1\text{s}$ (versus 0.23s for the fixed policies). Crucially, in a deployed system the parameters this scheduler reads already exist in the cache subsystem and the inference engine, so no additional state is required; the only cost incurred is this computation overhead.

6 Discussion

Conclusion. Existing LLM inference cache systems lack an adaptive mechanism for balancing KV-cache restoration and recomputation. To address this limitation, we propose an I/O-aware KV-cache management policy that maximizes inference throughput while increasing the success rate of satisfying both throughput and tail-latency constraints. According to our experiments, the proposed policy can achieve up to a 2–10× performance improvement.

Ongoing and Future Work. As future work, we plan to implement the proposed policy in a production-grade LLM-serving system and evaluate it under a diverse set of workloads, GPUs, and storage configurations. While this paper relies on simulation to study the trade-off between KV-cache restoration and recomputation, a real-world implementation would enable us to quantify runtime overheads, validate the accuracy of our performance models, and measure end-to-end inference performance under realistic operating conditions. We also plan to investigate how data placement, cache-hit ratios, caching policies, and access patterns affect policy effectiveness in dynamic multi-tier storage hierarchies. Finally, we intend to extend our work to distributed inference environments, where KV-cache data may be shared, migrated, or restored across multiple nodes, introducing new challenges and opportunities for adaptive-cache management for LLMs.

Acknowledgments

We thank the anonymous HotStorage reviewers for their valuable feedback. We also thank Danny Harnik, Thomas Parnell, Alexander Joukov, and Tahsin Ahmed for their contributions. This work was made possible in part thanks to IBM support; a SUNY/IBM Alliance award; and NSF awards CNS-2106263, CNS-2106434, CNS-2524208, CNS-2214980, and OAC-2230078.

References

- [1] Amey Agrawal, Nitin Kedia, Anmol Agarwal, Jayashree Mohan, Nipun Kwatra, Souvik Kundu, Ramachandran Ramjee, and Alexey Tumanov. 2025. On Evaluating Performance of LLM Inference Serving Systems. *arXiv preprint arXiv:2507.09019* (2025).
- [2] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369* (2023).
- [3] BIZON Tech. 2026. GPU Benchmarks. <https://bizon-tech.com/gpu-benchmarks>. Accessed: 2026-05-13.
- [4] Zhen Cao, Geoff Kuenning, and Erez Zadok. 2020. Carver: Finding important parameters for storage system tuning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 43–57.
- [5] Shiyang Chen, Rain Jiang, Dezhi Yu, Jinlai Xu, Mengyuan Chao, Fanlong Meng, Chenyu Jiang, Wei Xu, and Hang Liu. 2024. Kvdirect: Distributed disaggregated llm inference. *arXiv preprint arXiv:2501.14743* (2024).
- [6] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. 2025. {IMPRESS}: An {Importance-Informed} {Multi-Tier} prefix {KV} storage system for large language model inference. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 187–201.
- [7] Yihua Cheng, Kuntai Du, Jiayi Yao, and Junchen Jiang. 2024. Do large language models need a content delivery network? *arXiv preprint arXiv:2409.13761* (2024).
- [8] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffanetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. 2024. Llm-inference-bench: Inference benchmarking of large language models on ai accelerators. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1362–1379.
- [9] Kexin Chu, Zixu Shen, Sheng-Ru Cheng, Dawei Xiang, Ziqin Liu, and Wei Zhang. 2025. Mcam: Efficient llm inference with multi-tier kv cache management. In *2025 IEEE 45th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 571–581.
- [10] Kexin Chu, Zixu Shen, Dawei Xiang, and Wei Zhang. 2025. Safekv: Safe kv-cache sharing in llm serving. In *Machine Learning for Computer Architecture and Systems 2025*.
- [11] Shengkun Cui, Archit Patke, Hung Nguyen, Aditya Ranjan, Ziheng Chen, Phuong Cao, Gregory Bauer, Brett Bode, Catello Di Martino, Saurabh Jha, et al. 2025. Story of Two GPUs: Characterizing the Resilience of Hopper H100 and Ampere A100 GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1145–1164.
- [12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.
- [13] Tyler Estro, Mário Antunes, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Guiding simulations of multi-tier storage caches using knee detection. In *2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.
- [14] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. 2020. Desperately seeking... optimal {Multi-Tier} cache configurations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.
- [15] FS. 2025. Understanding PCIe Versions, Lanes, and Slot Types Explained. <https://www.fs.com/blog/understanding-pcie-versions-lanes-and-slot-types-explained-28862.html>. Accessed: 2026-05-13.
- [16] Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. 2025. Niyama: Breaking the silos of llm inference serving. *arXiv preprint arXiv:2503.22562* (2025).
- [17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
- [18] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. 2024. Mélange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527* (2024).
- [19] IBM. 2026. Accelerating AI inference with IBM Storage Scale. <https://research.ibm.com/blog/accelerating-ai-inference-with-ibm-storage-scale>. Accessed: 2026-06-09.
- [20] Chaoyi Jiang, Lei Gao, Hossein Entezari Zarch, and Murali Annavaram. 2024. Efficient llm inference with i/o-aware partial kv cache recomputation. *arXiv preprint arXiv:2411.17089* (2024).
- [21] Shuwei Jin, Xueshen Liu, Qingzhao Zhang, and Z. Morley Mao. 2025. Compute Or Load KV Cache? Why Not Both? *arXiv:2410.03065 [cs.LG]* <https://arxiv.org/abs/2410.03065>
- [22] Rupinder Kaur, Arghavan Asad, Seham Al Abdul Wahid, and Farah Mohammadi. 2025. A survey of advancements in scheduling techniques for efficient deep learning computations on GPUs. *Electronics* 14, 5 (2025), 1048.
- [23] Minsu Kim, Seongmin Hong, RyeoWook Ko, Soongyu Choi, Hunjong Lee, Junsoo Kim, Joo-Young Kim, and Jongse Park. 2025. Oaken: Fast and efficient llm serving with online-offline hybrid kv cache quantization. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 482–497.
- [24] 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*. 611–626.
- [25] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.
- [26] Ang Li. 2019. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswhitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 94–110.
- [27] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. 2024. Llm inference serving: Survey of recent advances and opportunities. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.
- [28] Qingyuan Li, Ran Meng, Yiduo Li, Bo Zhang, Liang Li, Yifan Lu, Xiangxiang Chu, Yerui Sun, and Yuchen Xie. 2023. A speed odyssey for deployable quantization of llms. *arXiv preprint arXiv:2311.09550* (2023).
- [29] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. 2024. Evaluating quantized large language models. *arXiv preprint arXiv:2402.18158* (2024).
- [30] Yuhao Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. 2025. Lmcache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665* (2025).
- [31] Yuhao Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. 2024. Cachegen: Kv cache compression and streaming

- for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 38–56.
- [32] LLM-d. 2026. LLM-d KV Cache. <https://github.com/llm-d/llm-d-kv-cache>. Accessed: 2026-05-13.
- [33] Oteo Mamo, Olga Kogiou, Hyunjin Yi, and Weikuan Yu. 2026. Comparative Characterization of KV Cache Management Strategies for LLM Inference. *arXiv preprint arXiv:2604.05012* (2026).
- [34] Tiyasa Mitra, Ritika Borkar, Nidhi Bhatia, Ramon Matas, Shivam Raj, Dheevatsa Mudigere, Ritchie Zhao, Maximilian Golub, Arpan Dutta, Sailaja Madduri, et al. 2025. Beyond the buzz: A pragmatic take on inference disaggregation. *arXiv preprint arXiv:2506.05508* (2025).
- [35] Isa Muradli. 2025. Insights into GPUDirect Data Transfer through NIXL Benchmarking. In *2025 IEEE International Conference on eScience (eScience)*. IEEE.
- [36] Quan Nguyen-Tri, Mukul Ranjan, and Zhiqiang Shen. 2025. Attention is all you need for kv cache in diffusion llms. *arXiv preprint arXiv:2510.14973* (2025).
- [37] NVIDIA. 2023. NVIDIA GPUDirect Storage. <https://docs.nvidia.com/cuda/archive/11.4.2/pdf/GDS.pdf>. Accessed: 2026-05-13.
- [38] NVIDIA. 2026. NVIDIA GPUDirect RDMA. https://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf. Accessed: 2026-05-13.
- [39] NVIDIA. 2026. NVIDIA NVLink. <https://www.nvidia.com/en-in/data-center/nvlink>. Accessed: 2026-05-13.
- [40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.
- [41] Rajeev Patwari, Ashish Sirasao, and Devleena Das. 2025. Forecasting LLM inference performance via hardware-agnostic analytical modeling. *arXiv preprint arXiv:2508.00904* (2025).
- [42] Nearchos Potamitis, Lars Henning Klein, Chongyang Xu, Attreyee Mukherjee, Bardia Mohammadi, Niket Tandon, Laurent Bindschaedler, and Akhil Arora. 2025. Cache saver: A modular framework for efficient, affordable, and reproducible LLM inference. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*.
- [43] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading more storage for less computation—a {KVCACHE-centric} architecture for serving {LLM} chatbot. In *23rd USENIX conference on file and storage technologies (FAST 25)*. 155–170.
- [44] Zebin Ren. 2025. An I/O Characterizing Study of Offloading LLM Models and KV Caches to NVMe SSD. In *Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*.
- [45] ShareAI Lab. 2023. ShareGPT-Chinese-English-90k: A Bilingual Chinese-English Human-Machine Dialogue Dataset. <https://huggingface.co/datasets/shareAI/ShareGPT-Chinese-English-90k>. Hugging Face dataset repository (Accessed: 2026-06-09).
- [46] Ying Sheng. 2023. Flexgen: High-throughput generative inference of large language models with a single GPU. In *International Conference on Machine Learning (ICML)*.
- [47] Zhiyuan Shi, Qibo Qiu, Feng Xue, Zhonglin Jiang, Li Yu, Jian Jiang, Xiaofei He, and Wenxiao Wang. 2026. HeteroCache: A Dynamic Retrieval Approach to Heterogeneous KV Cache Compression for Long-Context LLM Inference. *arXiv preprint arXiv:2601.13684* (2026).
- [48] He Sun, Li Li, Mingjun Xiao, and Chengzhong Xu. 2025. Breaking the Boundaries of Long-Context LLM Inference: Adaptive KV Management on a Single Commodity GPU. *arXiv preprint arXiv:2506.20187* (2025).
- [49] UserBenchmark. 2026. UserBenchmark. <https://www.userbenchmark.com>. Accessed: 2026-05-13.
- [50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- [51] VLLM. 2026. vLLM Documentation. <https://docs.vllm.ai/en/latest>. Accessed: 2026-05-13.
- [52] James Vo. 2024. Sparseaccelerate: Efficient long-context inference for mid-range gpus. *arXiv preprint arXiv:2412.06198* (2024).
- [53] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. {KVCACHE} Cache in the Wild: Characterizing and Optimizing {KVCACHE} Cache at a Large Cloud Provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 465–482.
- [54] Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. 2025. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. *arXiv preprint arXiv:2505.22618* (2025).
- [55] Yichun Xu, Navjot K Khaira, and Tejinder Singh. 2026. KV Cache Optimization Strategies for Scalable and Efficient LLM Inference. *arXiv preprint arXiv:2603.20397* (2026).
- [56] Huan Yang, Renji Zhang, Mingzhe Huang, Weijun Wang, Yin Tang, Yuanchun Li, Yunxin Liu, and Deyu Zhang. 2025. Kvshare: An llm service system with efficient and effective multi-tenant kv cache reuse. *arXiv preprint arXiv:2503.16525* (2025).
- [57] Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. 2026. Kvlink: Accelerating large language models via efficient kv cache reuse. *Advances in Neural Information Processing Systems* 38 (2026), 133797–133824.
- [58] Jiayi Yao. 2025. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the twentieth European conference on computer systems*.
- [59] Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, et al. 2026. Kvcomm: Online cross-context kv-cache communication for efficient llm-based multi-agent systems. *Advances in Neural Information Processing Systems* 38 (2026), 17882–17928.
- [60] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. 2024. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363* (2024).
- [61] Fei Zeng. 2025. H2O: Heterogeneity-Aware Hierarchical Orchestration for Memory-Efficient on-Device LLM Inference. *IEEE Transactions on Mobile Computing* (2025).
- [62] Qiuyang Zhang, Kai Zhou, Ding Tang, Kai Lu, Cheng Li, Zhenyu Yang, Peng Xu, and Jiguang Wan. 2026. ScoutAttention: Efficient KV Cache Offloading via Layer-Ahead CPU Pre-computation for LLM Inference. *arXiv preprint arXiv:2603.27138* (2026).
- [63] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.
- [64] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.
- [65] Xuanhe Zhou, Junxuan He, Wei Zhou, Haodong Chen, Zirui Tang, Haoyu Zhao, Xin Tong, Guoliang Li, Youmin Chen, Jun Zhou, et al. 2025. A survey of llm× data. *arXiv preprint arXiv:2505.18458* (2025).