PAVE: Partial KV Cache Eviction for Fine-Grained KV Reclamation in LLM Serving

Seong Jang[1], Kihyun Kim[1], Jinwoo Kim[1], Min Heo[1], Dong Li[2], Youngjae Kim[1,†]
[1]Sogang University, Seoul, Republic of Korea, [2]University of California, Merced, CA, USA
{jseong, kion777, jinwookim, heomin02, youkim}@sogang.ac.kr, dli35@ucmerced.edu

***Abstract*—Large language model (LLM) serving systems commonly rely on *request-level preemption* to resolve GPU memory pressure. However, reclaiming memory at request granularity forces the runtime to evict an entire KV cache even when only a few additional KV blocks are needed. We identify this inefficiency as *over-eviction*, which unnecessarily increases recomputation, degrades latency, and leads to time-per-output-token (TPOT) spikes and service-level objective (SLO) violations. We present PAVE, a fine-grained KV reclamation mechanism that replaces request-level preemption with block-level reclamation. Instead of evicting an entire request, PAVE reclaims only the KV blocks needed to satisfy the current memory shortage and selectively restores them upon resumption. PAVE requires changes only to the preemption path of an existing runtime while preserving its continuous batching and scheduling mechanisms. This design allows PAVE to operate independently of scheduler decisions and to be readily integrated into existing serving systems. We implement PAVE on vLLM and evaluate it with Qwen3-14B on the real-world Azure Conversation workload and a synthetic prefill-heavy workload. PAVE eliminates over-eviction across both workloads, improves TPOT-based SLO satisfaction by up to $1.87\times$ over vLLM, and achieves the lowest mean TPOT across all request arrival rates, while reducing pure-recompute step latency from over 300ms to roughly 113ms.**

***Index Terms*—LLM Inference, KV Cache Management, Scheduling, Preemption, Service Level Objective**

I. Introduction

Large language models (LLMs) are increasingly being deployed in a wide range of online services, including conversational AI, retrieval-augmented generation (RAG), multimodal reasoning, and agentic workflows [1]–[4]. These applications are evolving beyond short, single-prompt interactions toward workloads that leverage long and complex contextual information. Consequently, LLM serving systems must process long-context requests under high concurrency while maintaining low token generation latency to satisfy per-token service-level objectives (SLOs) required by interactive services.

Modern LLM serving runtimes employ continuous batching to reduce request admission delay and improve serving throughput under online workloads [5]–[7]. Continuous batching dynamically updates the batch at every iteration, removing completed requests and immediately admitting waiting requests. Unlike conventional static batching, it does not delay the execution of newly arrived requests until all requests in the batch finish generation. As a result, serving runtimes can sustain high request concurrency. However, high concurrency also requires maintaining more per-request runtime state in GPU memory. Among these states, the key-value (KV) cache is repeatedly accessed during token generation and grows proportionally with sequence length. As recent LLM workloads increasingly leverage long prompts and accumulated context, the KV cache footprint per request continues to grow. Under a fixed GPU memory budget, this trend rapidly reduces the available memory headroom and further exacerbates KV memory pressure in serving runtimes.

As memory headroom diminishes, serving runtimes employ preemption to avoid out-of-memory (OOM) conditions by temporarily suspending selected requests and reclaiming their KV state [6]. The freed KV blocks are then reallocated to other requests. In recomputation-based preemption, the reclaimed KV state must be restored through recomputation when the preempted request is resumed. Consequently, the recovery overhead is largely determined by the amount of KV state reclaimed at the time of preemption.

Motivation. Existing preemption mechanisms operate at request granularity. Under memory pressure, request-level preemption [6] evicts the entire KV state of a selected request, even when the scheduler only needs a few additional KV blocks to continue execution. This granularity mismatch can reclaim substantially more KV state than necessary. We refer to such unnecessary reclamation beyond the actual memory shortage as *over-eviction*.

Over-eviction directly increases the cost of recomputation-based preemption. Unnecessarily evicted KV blocks must be recomputed later, wasting GPU compute needed for active decoding. As a result, over-eviction can increase time-per-output-token (TPOT) and lead to higher service latency and SLO violations (§III).

Limitations of SOTA Approaches. Existing approaches do not directly control this reclamation granularity. Prediction-based scheduling reduces the likelihood of preemption by estimating request lengths or memory demand [8], victim-selection policies choose requests with lower recovery costs [9], and restoration optimization techniques reduce the cost of recovering evicted KV state [10], [11]. These techniques mitigate when preemption occurs, which request is selected, or how efficiently evicted KV state is restored.

However, because request-level preemption still reclaims the selected request's KV state as a whole, existing approaches cannot control the amount of KV state reclaimed at the moment of preemption. Consequently, they do not address the root cause of over-eviction: the mismatch between the

†Y. Kim is the corresponding author.

actual KV shortage and the coarse granularity of request-level reclamation.

Our Approach. We present **PA**rtial K**V** Cache **E**viction (PAVE), a fine-grained KV reclamation mechanism for online LLM serving. PAVE decouples preemption from request-level eviction: instead of reclaiming the entire KV state of a victim request, it evicts only the tail KV blocks needed to satisfy the instantaneous memory shortage. The partially evicted request is suspended and later resumes by recomputing only the evicted blocks.

PAVE builds on two mechanisms. First, it allows the runtime to reclaim a subset of a request's KV blocks rather than treating the request as an indivisible eviction unit. Second, it determines how many tail blocks to reclaim at each preemption point, matching the reclaimed capacity to the current allocation demand. Together, these mechanisms reduce unnecessary KV loss, lower recomputation during restoration, and mitigate TPOT degradation under continuous batching.

We implement PAVE in vLLM and evaluate it with Qwen3-14B on two workloads: a real-world conversational workload derived from Azure traces and a prefill-heavy synthetic workload. We compare against default vLLM and SRF, a cost-aware preemption victim-selection policy inspired by INFERMAX [9]. Across both workloads, PAVE eliminates measured over-eviction from request-level reclamation and reduces recompute step latency from over 300ms to 113ms. These reductions improve TPOT-based SLO attainment by up to $1.87\times$ over default vLLM and yield the lowest mean TPOT across request arrival rates.

II. Background

A. LLM Inference and KV Cache

Inference in transformer-based LLMs [12] consists of two phases: *prefill* and *decode* [6], [7]. To avoid redundant computation over previously processed tokens, LLM serving systems maintain a Key-Value (KV) cache for each request in GPU memory.

The KV cache stores the Key (K) and Value (V) tensors computed at each transformer layer and reuses them in subsequent attention operations. During the prefill phase, the entire input prompt is processed in a single forward pass. The resulting K and V tensors for all prompt tokens are computed and stored in the request's KV cache. Because prefill is dominated by large matrix operations, it is generally compute-bound.

During the decode phase, new tokens are generated autoregressively, one token at a time. At each decoding step, the model retrieves the previously computed K and V tensors from the KV cache to perform attention and appends the newly generated token's K and V tensors to the cache. Due to these repeated KV cache accesses, decode is generally memory-bound, and the KV cache footprint of each request grows as generation progresses.

B. PagedAttention and Continuous Batching

Modern LLM serving systems employ *PagedAttention* [6] and *continuous batching* [5] to improve GPU utilization and serving throughput. Together, these techniques provide the foundation for dynamically managing the KV caches of multiple requests within limited GPU memory.

PagedAttention stores a request's KV cache in fixed-size KV blocks rather than in a contiguous memory region. Each request maintains a block table that maps logical token positions to physical KV blocks. This design allows the serving system to allocate only the additional KV blocks required as a request's sequence grows. When a request completes or is preempted, the KV blocks occupied by that request are returned to the free block pool.

Continuous batching allows newly arrived requests to join the batch while existing requests continue executing decode steps. At each iteration, the scheduler constructs a batch from runnable requests and expands their KV caches within GPU memory as needed. Both the prefill phase of new requests and the decode phase of existing requests require free KV blocks. Consequently, as request concurrency and sequence lengths grow, contention for the shared free block pool becomes increasingly severe.

C. KV Cache Pressure and Request-level Preemption

When the free KV block pool becomes depleted, the scheduler may be unable to allocate the KV blocks required for either the prefill phase of a new request or the decode step of an existing request. To prevent out-of-memory (OOM) conditions, the serving system preempts one or more running requests and reclaims their KV blocks. vLLM, a representative continuous-batching LLM serving system, addresses this situation through *request-level preemption* [6], [13]. The scheduler selects a victim request and removes it from the running batch. Various victim-selection policies have been proposed, including Last-In-First-Out (LIFO) and Shortest-Request-First (SRF), with vLLM adopting the LIFO policy by default [6], [9].

The serving system then reclaims all KV blocks belonging to the victim request and returns them to the free block pool. The reclaimed blocks are subsequently reused for KV cache expansion during decode or for admitting new prefill requests.

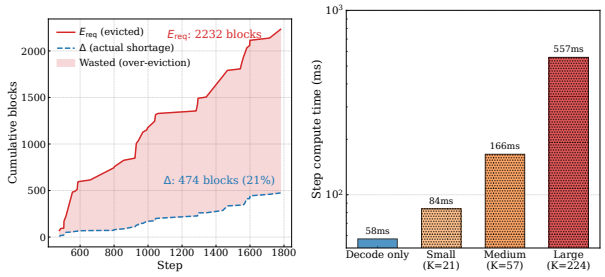
When a preempted request is scheduled again, its lost KV state must be restored. Two representative restoration mechanisms are swap and recompute. Swap preserves the KV cache by offloading it to CPU memory and later restoring it to GPU memory, but incurs CPU–GPU data transfer overhead [6]. Recompute, in contrast, discards the KV cache entirely and restores it by re-executing the request from the beginning when the preempted request is rescheduled, thereby incurring additional GPU computation overhead.

In this paper, we focus on recomputation-based preemption, which is the default mechanism adopted by vLLM and modern serving systems [14].

III. Motivation

A. Over-eviction in Request-level Preemption

Request-level preemption alleviates GPU memory pressure by reclaiming all KV blocks occupied by a victim request. However, the additional KV blocks required by the scheduler to continue execution may be substantially fewer than the victim request's total KV footprint. This discrepancy arises



(a) (b)

Fig. 1: Impact of request-level preemption under the Azure workload at 2.5 requests/s (RPS). (a) Mismatch between reclaimed (E_{req}) and required (Δ) KV blocks: Cumulative reclaimed KV blocks versus cumulative KV shortages during execution. (b) Restoration latency as a function of the number of recomputed KV blocks (K).

from the coarse-grained reclamation granularity of request-level preemption.

Let Δ denote the number of additional KV blocks required for the currently running requests to proceed to the next execution step at a given preemption event. In request-level preemption, if a victim request r_i is selected and occupies B_i KV blocks, the number of reclaimed KV blocks becomes B_i rather than the actual shortage Δ. Consequently, when $B_i > \Delta$, the system reclaims more KV blocks than are necessary to resolve the current memory pressure.

In this paper, we define such unnecessary reclamation of KV blocks as *over-eviction*. More generally, when multiple victim requests are selected, let $\mathcal{V}$ denote the set of victim requests. The total number of KV blocks reclaimed by request-level preemption is then given by:

$$E_{\text{req}} = \sum_{r_i \in \mathcal{V}} B_i \tag{1}$$

The number of over-evicted KV blocks is given by:

$$O = \max(0, E_{\text{req}} - \Delta) \tag{2}$$

In recomputation-based preemption, reclaimed KV state must be regenerated through recomputation when the request resumes. Consequently, KV blocks reclaimed due to over-eviction translate directly into unnecessary recomputation overhead. Therefore, minimizing preemption overhead requires not only selecting appropriate victim requests, but also precisely controlling the amount of KV state reclaimed during preemption.

B. Quantifying Over-eviction and Its Impact

To quantify both the extent of over-eviction and its performance impact, we instrumented the vLLM serving runtime to record preemption events. Experiments were conducted using the Qwen3-14B model [15] and an Azure conversational workload [16] generated according to the setup described in Section V. The arrival rate was set to 2.5 requests per second (RPS), representing a load condition under which preemption is repeatedly observed on the target model and hardware configuration. Fig. 1 illustrates the extent of over-eviction and its performance impact across the instrumented preemption events. We make two key observations.

Observation 1: Request-level preemption over-evicts KV blocks. Fig. 1a compares the cumulative KV block shortage observed across scheduling steps with the cumulative number of KV blocks actually reclaimed by request-level preemption. The results show that request-level preemption consistently reclaims more KV blocks than required. Across all reclaimed blocks, only about 21% contributed to resolving actual KV block shortages. In one representative preemption event, 257 KV blocks were reclaimed to satisfy a shortage of only four KV blocks.

These results confirm that over-eviction is not merely a theoretical artifact but a prevalent phenomenon in real vLLM serving runtimes. Since over-evicted KV state must be restored through recomputation upon request resumption, over-eviction directly translates into additional restoration overhead.

Observation 2: Over-eviction amplifies recomputation and serving overhead. Fig. 1b shows the relationship between the number of KV blocks requiring recomputation (K) and the resulting recomputation latency, based on the recompute events observed in Fig. 1a.

The results show that recomputation latency increases with the number of KV blocks that must be recomputed. In particular, when $K = 224$, it reaches as high as 557ms for restoring the evicted KV state. This result confirms that KV blocks unnecessarily reclaimed due to over-eviction directly translate into additional recomputation work during request restoration.

In continuous batching, recomputation and decode computation share the same scheduling step, which completes only after all requests finish. Thus, recomputation-heavy preempted requests can become the critical path, delaying other decode requests. Consequently, over-eviction increases both restoration cost and per-step decode latency.

Takeaway. These results show that preemption overhead depends not only on preemption frequency, but also on the amount of KV state evicted per event. Therefore, the coarse reclamation granularity of request-level preemption is a key source of recomputation overhead.

C. Limitations of Existing Approaches

The above analysis shows that request-level preemption overhead is influenced not only by the occurrence of preemption events but also by the amount of KV blocks reclaimed at each event. However, existing LLM serving and KV-cache management techniques have focused on memory-pressure-aware scheduling, cost-aware victim selection, or optimizing the restoration and reuse of evicted KV state, rather than controlling the reclamation granularity itself [8]–[11], [17].

While these approaches can reduce the frequency of preemption events or the restoration cost after preemption, they do not address the fundamental limitation that KV state is reclaimed at request granularity regardless of the actual memory shortage at the time of preemption. Consequently, they do not directly address over-eviction caused by the mismatch between the number of KV blocks required and the number of KV blocks actually reclaimed.

Prediction-based scheduling. Prediction-based scheduling techniques estimate output length or memory demand to adjust

request ordering or admission timing [8], [17]. These approaches can delay memory pressure or reduce the frequency of preemption events. However, due to bursty arrivals, variability in output length, or prediction errors, preemption may still occur, and the amount of KV blocks reclaimed at each event is ultimately determined by the runtime reclamation policy. In other words, while scheduling decisions can influence the likelihood of preemption, they do not control the amount of KV state reclaimed once preemption occurs.

Victim-selection policies. INFERMAX selects victim requests with low restoration cost to reduce the recomputation overhead of request-level preemption [9]. However, this approach optimizes only the choice of victims and does not directly control the amount of KV blocks reclaimed during preemption. In request-level preemption, the entire KV footprint of the selected victim is reclaimed, which can result in E_{req} exceeding the actual shortage Δ. Therefore, while victim selection can mitigate preemption cost, it cannot fundamentally address over-eviction induced by request-level reclamation granularity.

KV restoration optimization. KV-cache restoration or reuse techniques reduce swap or recomputation overhead incurred when preempted requests resume execution by storing or reusing evicted KV state [10], [11]. However, these approaches optimize the restoration path after KV state has already been evicted. Therefore, while they can reduce post-preemption restoration cost, they do not eliminate the root cause of over-eviction, where unnecessary KV blocks are reclaimed during preemption.

As a result, existing approaches mitigate preemption overhead by either reducing the frequency of preemption events, selecting lower-cost victims, or accelerating the restoration of evicted KV state. However, they all commonly fail to directly address the granularity mismatch between the actual shortage Δ and the amount of KV state reclaimed under request-level preemption, E_{req}. Therefore, reducing preemption overhead requires a runtime memory-management mechanism that goes beyond scheduling or post-preemption restoration and adjusts the KV reclamation granularity according to the actual memory shortage.

IV. Design of PAVE

In this section, we present the design of PAVE, a block-granular reclamation mechanism for reducing over-eviction in online LLM serving. PAVE preserves the existing serving runtime's admission, batching, and decode execution, and replaces only the preemption path triggered by KV memory shortage. Instead of evicting an entire victim request, PAVE reclaims only the number of tail KV blocks needed to resolve the current shortage, while preserving the remaining prefix KV state for later restoration.

A. PAVE *Design Overview*

Fig. 2 illustrates how PAVE handles memory shortage through block-level reclamation instead of request-level preemption. While the baseline approach returns the entire KV footprint of a victim request to the free block pool, PAVE incrementally reclaims only tail KV blocks until the shortage Δ is satisfied. The reclaimed blocks are immediately returned

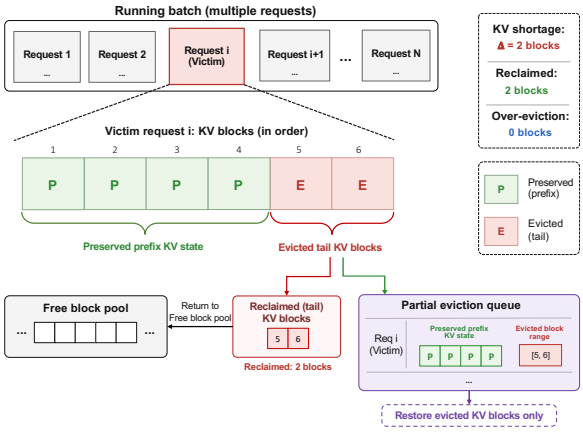


Fig. 2: Overview of block-level KV reclamation in PAVE, which reclaims only the required number of tail KV blocks and preserves the prefix KV state for selective restoration.

to the free block pool, while the remaining prefix KV state and the evicted block range are recorded in a partial eviction queue. In this way, PAVE resolves memory pressure while preserving reusable portions of a request's execution state for subsequent restoration and reuse.

The preemption path of PAVE consists of three components. *Block-level KV reclamation* reclaims tail KV blocks according to the size of the memory shortage. *State management for partially-evicted requests* separates requests with partially evicted KV state from the running queue and tracks both the preserved prefix state and the evicted range. *Selective restoration* restores only the evicted range when the request is resumed, thereby avoiding full request-level recomputation. Algorithm 1 summarizes this workflow, and the following sections describe each component in order.

B. *Block-level KV Reclamation*

Block-level KV reclamation is the core preemption mechanism of PAVE. In conventional request-level preemption, once a victim request is selected, its entire KV state is reclaimed, meaning that the amount of reclaimed memory is determined by the victim request's KV footprint rather than the actual memory shortage. PAVE refines this all-or-nothing reclamation strategy to operate at block granularity.

When memory shortage occurs, the scheduler selects a victim request from either the running queue or the partial eviction queue. PAVE then reclaims the KV blocks of the victim request incrementally, rather than removing the entire KV state at once, starting from the tail KV blocks until the required KV shortage is resolved. Each reclaimed block is immediately returned to the free block pool and can be reused for allocations required by the current decode step or newly admitted requests.

PAVE adopts tail-first reclamation to preserve prefix KV state. By reclaiming tail KV blocks first, the preserved prefix KV state remains intact, allowing the system to restore only the evicted KV state when the request resumes.

Algorithm 1 PAVE: Block-level KV Reclamation

Require: Running queue $\mathcal{R}$, waiting queue $\mathcal{W}$, partial eviction queue $\mathcal{P}$, free block pool $\mathcal{F}$

while serving **do**
  Allocate KV blocks for the next tokens of requests in $\mathcal{R}$
  if KV block shortage occurs **then**
    $\Delta \leftarrow$ number of additional KV blocks required
    while $\Delta > 0$ **do**
      $r \leftarrow$ SelectVictim($\mathcal{R} \cup \mathcal{P}$)
      while $\Delta > 0$ and r has reclaimable KV blocks **do**
        $b \leftarrow$ SelectTailBlock(r)
        Reclaim KV block b from request r
        Return b to free block pool $\mathcal{F}$
        Record b as an evicted KV block of r
        $\Delta \leftarrow \Delta - 1$
      end while
      if r has remaining valid KV blocks **then**
        Move r to partial eviction queue $\mathcal{P}$
        Record the preserved prefix KV state of r
      else
        Move r to waiting queue $\mathcal{W}$
      end if
    end while
  end if
  if no KV block shortage and enough free KV blocks are available for restoration **then**
    $r \leftarrow$ SelectRestoreCandidate($\mathcal{W} \cup \mathcal{P}$)
    Restore evicted KV blocks of r via recomputation
    Move r back to running queue $\mathcal{R}$
  end if
  Execute one model step for the requests in $\mathcal{R}$
end while

This approach allows the reclamation amount to be adjusted according to the memory shortage. As a result, execution can proceed without evicting the entire KV state of a victim request, while the preserved prefix KV state can be reused when the request resumes. Consequently, block-level KV reclamation reduces over-eviction and decreases the amount of KV state that must be recovered through recomputation.

C. State Management for Partially-Evicted Requests

Block-level KV reclamation produces partially-evicted requests in which only a subset of KV blocks is reclaimed. Such requests cannot be directly scheduled in either the waiting queue or the running queue. A waiting request refers to one that has not yet constructed valid KV state, and thus differs fundamentally from a partially-evicted request. In contrast, the running queue assumes that all requests have complete KV state required for decoding, and therefore cannot accommodate requests with incomplete KV state.

To address this issue, PAVE introduces a partial eviction queue. Requests from which some KV blocks have been reclaimed are migrated from the running queue to the partial eviction queue. This queue maintains both the preserved prefix KV state and the range of evicted blocks for each request. In this way, PAVE preserves the remaining KV state of partially evicted requests while maintaining the execution-state invariant of the running queue.

Requests in the partial eviction queue may be subject to further reclamation if memory pressure persists. In this case, PAVE additionally reclaims the remaining tail KV blocks of the same request to resolve the memory shortage. When all KV blocks have been removed, no partially preserved state remains, and the request is moved to the waiting queue. This design allows the reclamation scope within a single request to be progressively expanded according to the degree of memory pressure.

D. Selective Restoration

Selective restoration is the process of returning partially-evicted requests to the running queue. When memory pressure is relieved and the existing serving runtime can admit new requests into the running batch, the runtime considers requests in both the waiting queue and the partial eviction queue as candidates for admission. If a selected request is in a partially-evicted state, PAVE reuses the preserved prefix KV state and restores only the evicted KV range via recomputation before readmitting the request into the running queue.

By preserving the prefix KV state during preemption, PAVE restricts the restoration scope to the evicted tail KV range. The restoration cost is therefore determined not by the total KV footprint of the request, but by the number of KV blocks reclaimed during preemption. This selective restoration lowers the recomputation overhead when a request resumes and mitigates the latency impact of recomputation on decode requests within the same scheduling iteration.

V. Evaluation

A. Experimental Setup

We implement PAVE in the v1 engine of vLLM v0.16.0 using the setup in Table I. PAVE preserves vLLM's continuous batching and KV block allocator, but replaces request-level preemption under memory pressure with block-level KV reclamation. We disable prefix caching and CUDA Graph to isolate recomputation and scheduling effects. All experiments use Qwen3-14B [15], a grouped-query attention model [18].

Baselines. We compare PAVE against three request-level reclamation baselines that vary either the scheduling policy or the preemption victim-selection policy.

- **vLLM.** The default vLLM policy uses FCFS scheduling and selects preemption victims in a Last-In-First-Out (LIFO) order, evicting the entire KV state of each selected request.
- **SRF.** SRF uses the same FCFS scheduler as vLLM but replaces LIFO victim selection with a cost-aware policy inspired by INFERMAX [9]. It prioritizes requests with smaller KV footprints, which have lower recomputation cost after preemption. Since INFERMAX is not publicly available, we implement the described victim-selection policy directly in vLLM.
- **SOF.** SOF replaces FCFS scheduling with an oracle Shortest-Output-First scheduler, which admits requests with

TABLE I: Experimental setup.

GPU		**Server**	
Model	NVIDIA L40S	CPU	2× Xeon Gold 6548Y+
FP16	91.6 TFLOPS	Memory	1 TiB DDR5
VRAM	48 GB	NUMA	2 nodes
Bandwidth	864 GB/s	PCIe	Gen4 ×16 (32 GB/s)
Software		**Model**	
OS/CUDA	Ubuntu 24.04 / 12.8	Name	Qwen3-14B
Framework	PyTorch 2.8.0	Attn.	GQA (8 KV heads)
Engine	vLLM 0.16.0		

TABLE II: Workload characteristics.

Workload	Input/Output Length	Note
Azure	1,155 / 211	real conversation trace
Prefill-Heavy[1]	2,048 / 128	synthetic, fixed length

[1] Fixed input/output lengths (2,048/128) following the code-analysis category of the MLPerf Client benchmark [19].

shorter output lengths first. Victim selection remains identical to vLLM, i.e., LIFO request-level preemption. This baseline tests whether PAVE remains effective beyond FCFS scheduling.

Workloads. We evaluate each system under varying request arrival rates, with requests injected according to a Poisson arrival process. We use two workloads summarized in Table II: Azure Conversation [16], a real-world online serving trace, and Prefill-Heavy, a synthetic fixed-length workload with long inputs and short outputs. Prefill-Heavy stresses preemption recovery because each request maintains a large KV footprint, making over-eviction effects easier to isolate.

Metrics. We report latency, preemption behavior, and SLO attainment.

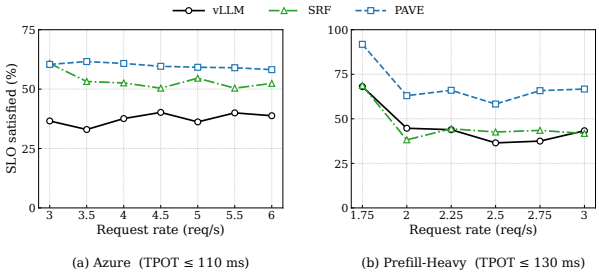
- **Normalized latency.** End-to-end request latency divided by the number of generated output tokens.
- **TTFT.** Time from request arrival to the first generated token.
- **TPOT.** Average per-token generation latency after the first output token.
- **Preemption and over-eviction.** We measure the number of preemption events and over-evicted KV blocks. Over-evicted blocks are computed at each preemption point as the gap between the number of reclaimed KV blocks and the number of KV blocks required to sustain generation, as defined in (2).
- **SLO satisfaction rate.** The fraction of requests satisfying the workload-specific TPOT SLO. TPOT is used because over-eviction-induced recomputation delays decode-phase token generation [7]. For each workload, we set the threshold to the default vLLM p_{95} per-request TPOT under low load, rounded up to the nearest 5 ms [20].

B. SLO Attainment

Fig. 3 reports the fraction of requests that satisfy the TPOT SLO as the request arrival rate increases. Across both Azure and Prefill-Heavy, PAVE sustains higher SLO attainment than request-level reclamation baselines.

On Azure, PAVE achieves up to $1.87\times$ higher SLO attainment than vLLM at 3.5 RPS and up to $1.18\times$ higher attainment than SRF at 4.5 RPS (Fig. 3a). These gains show that reducing the amount of KV state reclaimed during preemption translates into more stable decode-phase token generation under load.

PAVE also sustains higher SLO attainment on Prefill-Heavy. PAVE improves SLO attainment by up to $1.75\times$ over vLLM at 2.75 RPS and by up to $1.65\times$ over SRF at 2.0 RPS (Fig. 3b). In this workload, SRF approaches the behavior of vLLM because requests have similar KV footprints, leaving limited room for a victim-selection policy to reduce recovery cost. In contrast, PAVE reduces preemption overhead by changing



(a) Azure (TPOT ≤ 110 ms) (b) Prefill-Heavy (TPOT ≤ 130 ms)

Fig. 3: SLO attainment under varying request rates on Azure and Prefill-Heavy workloads.

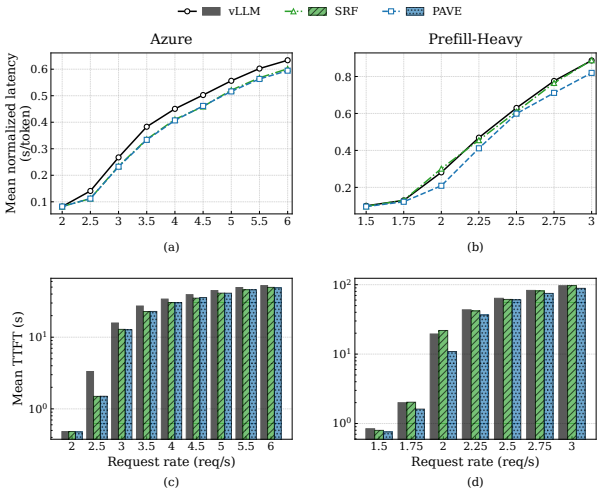


Fig. 4: End-to-end serving performance of Qwen3-14B under the Azure and Prefill-Heavy workloads. (a) and (b) show normalized latency, while (c) and (d) show TTFT.

the granularity of reclamation itself, rather than relying on heterogeneity among victim candidates.

Overall, these results indicate that improving victim selection alone is insufficient to preserve TPOT SLO attainment under memory pressure. PAVE addresses a complementary source of preemption overhead by reclaiming KV blocks in proportion to the instantaneous memory shortage.

C. E2E Performance

Fig. 4 reports normalized latency and TTFT as the request arrival rate increases. When preemption is rare, PAVE closely matches the request-level baselines. As memory pressure increases, however, PAVE increasingly outperforms them by reducing the amount of KV state that must be recomputed after preemption.

On Azure, PAVE achieves up to $1.26\times$ lower mean normalized latency and up to $2.21\times$ lower mean TTFT than vLLM, both at 2.5 RPS. On Prefill-Heavy, PAVE reduces mean normalized latency and mean TTFT by up to $1.35\times$ and $1.79\times$, respectively, compared to vLLM at 2.0 RPS. These gains come from avoiding unnecessary KV eviction,

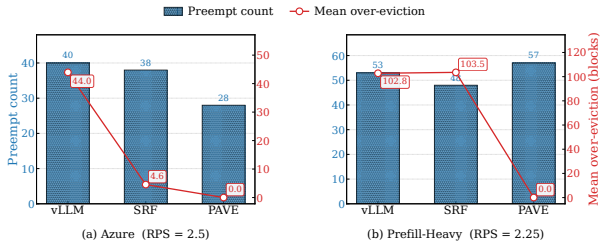
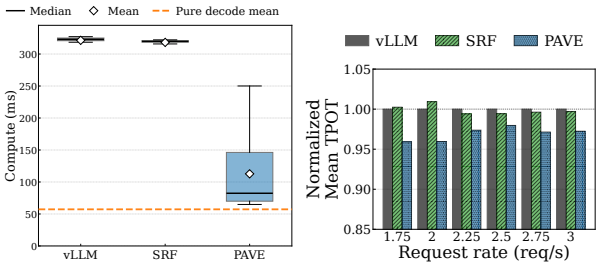


Fig. 5: Preemption frequency and average over-eviction under high-load conditions. Bars indicate the number of preemption events, while lines represent the mean number of over-evicted KV blocks per preemption.



(a) Pure recompute step latency (b) Mean TPOT

Fig. 6: Impact of recomputation on decode latency under the Prefill-Heavy workload. (a) Compute-time distribution of scheduling steps involving recomputation, compared to pure decode steps. (b) Mean TPOT normalized to vLLM across different request rates.

which lowers the recomputation cost incurred when preempted requests resume.

The comparison with SRF highlights the workload dependence of victim-selection policies. On Azure, SRF performs close to PAVE because request lengths are heterogeneous, allowing SRF to reduce recovery cost by selecting smaller-footprint victims. On Prefill-Heavy, however, requests have similar KV footprints, leaving little room for victim selection to reduce preemption cost. In this setting, PAVE achieves up to $1.44\times$ lower mean normalized latency and up to $2.01\times$ lower mean TTFT than SRF at 2.0 RPS. We further analyze this behavior in Section V-D.

D. Preemption and Over-Eviction Analysis

Fig. 5 compares preemption frequency and per-preemption over-eviction across systems to explain the workload-dependent behavior observed in the previous section. The analysis focuses on why SRF is competitive on the Azure workload but loses its advantage on the Prefill-Heavy workload.

Azure workload. In Fig. 5a, vLLM performs 40 preemptions at RPS=2.5 and over-evicts 44.0 KV blocks per preemption on average. SRF performs a similar number of preemptions (38), but reduces average over-eviction to 4.6 blocks. This shows that, when request lengths are diverse, selecting shorter victims can substantially reduce the cost of request-level reclamation. This explains why SRF achieves end-to-end latency comparable to PAVE on the Azure workload.

PAVE further reduces the number of preemptions to 28 and eliminates over-eviction, with 0.0 over-evicted blocks per preemption. Unlike SRF, which only changes victim selection, PAVE directly controls the number of KV blocks reclaimed according to the memory shortage.

Prefill-Heavy workload. Fig. 5b shows the limitation of victim selection under the Prefill-Heavy workload. At RPS=2.25, vLLM over-evicts 102.8 KV blocks per preemption on average, and SRF shows a similar value of 103.5 blocks. Since requests have large and similar input contexts, victim candidates have comparable KV footprints; selecting the shortest request therefore does not substantially reduce request-level reclamation. By contrast, PAVE keeps over-eviction at 0.0 blocks per preemption despite triggering the largest number of preemptions (57). This is because PAVE controls the reclamation amount directly, rather than relying on the availability of a short victim request.

E. Impact on Recompute and Decode Latency

We analyze how over-eviction affects recomputation cost and decode-time latency. Under the Prefill-Heavy workload, we measure scheduling steps that contain decode requests and recompute work, but no newly admitted prefill requests. This isolates the latency impact of recomputing evicted KV state.

Fig. 6a shows the distribution of compute time for scheduling steps that include recompute work. The average compute time of pure decode steps is approximately 57ms. In vLLM and SRF, request-level preemption requires recomputation of the full prior context of preempted requests. As a result, the recomputation workload increases and the mean compute time of steps that include recompute work exceeds 300ms.

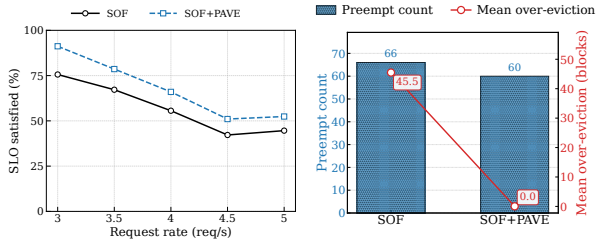
PAVE reduces the number of evicted KV blocks during preemption to match the actual memory shortage, thereby decreasing the amount of recomputation required upon readmission. As a result, the mean compute time of scheduling steps that include recompute work is reduced to approximately 113ms. This demonstrates that PAVE not only reduces over-eviction, but also directly mitigates recompute latency incurred when preempted requests resume execution.

This reduction also improves decode-time latency. When recompute work and decode requests are co-scheduled, decode requests cannot advance to the next token until recomputation finishes, so larger recompute work increases per-token latency. Fig. 6b shows that PAVE achieves lower normalized mean TPOT than vLLM and SRF across all RPS settings under the Prefill-Heavy workload, suggesting that limiting recomputation reduces decode-time interference.

Overall, PAVE reduces recomputation overhead by evicting less KV state during preemption, which shortens recompute-containing scheduling steps and reduces delay for co-scheduled decode requests. These results show that preemption overhead depends not only on preemption frequency, but also on the amount of KV state recomputed after each preemption.

F. Orthogonality to Scheduling Policies

In this section, we evaluate the scheduler-agnostic nature of PAVE by applying it to Shortest-Output-First (SOF), a non-FCFS oracle scheduler that prioritizes requests with shorter output lengths. SOF changes the request admission order, but



(a) SLO attainment across request rates.

(b) Preemption count and over-eviction at RPS=4.

Fig. 7: Applying PAVE on top of Shortest-Output-First (SOF) scheduling under the Azure workload. PAVE eliminates over-eviction and improves SLO attainment even with a non-FCFS scheduler, showing that it operates orthogonally to the scheduling policy.

it does not change the reclamation granularity. Thus, over-eviction remains unless PAVE is applied.

Fig. 7a shows the SLO satisfaction rate under varying request arrival rates in the Azure workload. SOF+PAVE achieves a higher SLO satisfaction rate than SOF alone across all arrival rates, with a maximum improvement from 75.6% to 91.2% (1.21×, RPS=3). This demonstrates that, even when the scheduling policy improves request admission ordering, reducing the amount of KV state unnecessarily reclaimed during preemption remains crucial for maintaining SLO performance.

Fig. 7b analyzes the preemption behavior at RPS=4. While both configurations trigger a similar number of preemptions (66 vs. 60), SOF alone over-evicts an average of 45.5 KV blocks per preemption, whereas SOF+PAVE eliminates over-eviction. Therefore, the performance improvement is primarily attributed to better reclamation granularity rather than a reduction in preemption frequency.

These results show that PAVE is not an FCFS-specific optimization, but a scheduler-agnostic KV reclamation mechanism applicable to different preemption-based scheduling policies.

VI. Conclusion

In this paper, we identified *over-eviction* as a hidden cost of request-level preemption in LLM serving systems. We presented PAVE, a block-granular KV reclamation mechanism that reclaims only the KV blocks needed to resolve memory shortage. Implemented in vLLM, PAVE eliminates over-eviction, reduces recomputation overhead, and improves TPOT-based SLO satisfaction by up to 1.87× over request-level preemption. These results show that reclamation granularity is a key design dimension for preemption-based LLM serving, independent of scheduling policy.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (RS-2024-00416666 and RS-2025-00564249) and by Institute of Information & communications Technology Planning & Evaluation (IITP) under the Artificial Intelligence Innovation Human Resources Development (IITP-RS-2026-25547954) grant funded by the Korea government (MSIT).

References

[1] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the Middle: How Language Models Use Long Contexts," *Transactions of the Association for Computational Linguistics (TACL)*, vol. 12, pp. 277–294, 2024.

[2] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 9459–9474, 2020.

[3] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual Instruction Tuning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, pp. 34892–34916, 2023.

[4] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in *International Conference on Learning Representations (ICLR)*, 2023.

[5] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "ORCA: A Distributed Serving System for Transformer-Based Generative Models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

[6] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proc. of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, 2023.

[7] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve," in *Proc. of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 117–134, 2024.

[8] R. Shahout, E. Malach, C. Liu, W. Jiang, M. Yu, and M. Mitzenmacher, "Don't Stop Me Now: Embedding Based Scheduling for LLMs," in *International Conference on Learning Representations (ICLR)*, 2025.

[9] K. Kim, J. Li, K. Hong, and A. Ailamaki, "Faster LLM Inference using DBMS-Inspired Preemption and Cache Replacement Policies." arXiv:2411.07447, 2025.

[10] S. Gao, Y. Chen, and J. Shu, "Fast State Restoration in LLM Serving with HCache," in *Proc. of the Twentieth European Conference on Computer Systems (EuroSys)*, pp. 128–143, 2025.

[11] C. Jiang, L. Gao, H. E. Zarch, and M. Annavaram, "KVPR: Efficient LLM Inference with I/O-Aware KV Cache Partial Recomputation," in *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19474–19488, 2025.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, pp. 5998–6008, 2017.

[13] Y. Xiang, X. Li, K. Qian, Y. Yang, D. Zhu, W. Yu, E. Zhai, X. Liu, X. Jin, and J. Zhou, "Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market," in *Proc. of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 1030–1045, 2025.

[14] vLLM Project, "Optimization and tuning — vLLM documentation." https://docs.vllm.ai/en/stable/configuration/optimization/, 2025. Accessed: 2026-06-06.

[15] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, *et al.*, "Qwen3 Technical Report," 2025. arXiv:2505.09388.

[16] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient Generative LLM Inference Using Phase Splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, IEEE, 2024.

[17] Y. Fu, S. Zhu, R. Su, A. Qiao, I. Stoica, and H. Zhang, "Efficient LLM Scheduling by Learning to Rank," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 37, pp. 59006–59029, 2024.

[18] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron, and S. Sanghai, "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints," in *Proc. of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, 2023.

[19] MLCommons, "MLPerf client benchmark." https://mlcommons.org/benchmarks/client/. Accessed: 2026-06-06.

[20] Y. Wang, Y. Chen, Z. Li, X. Kang, Y. Fang, Y. Zhou, Y. Zheng, Z. Tang, X. He, R. Guo, X. Wang, Q. Wang, A. C. Zhou, and X. Chu, "BurstGPT: A Real-World Workload Dataset to Optimize LLM Serving Systems," in *Proc. of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 5831–5841, 2025.