TENURE: Residency Leases for Tiered LLM KV Caches

Amir Noohi
University of Edinburgh
Edinburgh, United Kingdom
amir.noohi@ed.ac.uk

Antonio Barbalace
University of Edinburgh
Edinburgh, United Kingdom
Antonio.barbalace@ed.ac.uk

Abstract

In long-context LLM serving, the key-value (KV) cache outgrows GPU memory. Production systems such as Mooncake and LMCache spill it to DRAM and NVMe SSDs, CXL memory adds a tier between them, and a tiering policy decides which blocks stay in fast memory. Deployed policies estimate future accesses from recency or frequency. For the KV cache this estimate is unnecessary, because the serving engine already knows which blocks each running and queued request will read, and it is costly, because every block it moves occupies the links that serve the cache. On a physical GPU/DRAM/CXL/NVMe host, 16 online policies serve 24–82% of *demanded* KV bytes, those a request waits on, from NVMe, and moves that leave placement unchanged still slow decoding.

Tenure replaces the estimate with a *residency lease*, a reference count on a request's blocks that is taken when the request is queued, dropped when it completes, and expires if its client fails. A leased block stays off NVMe but can move among the faster tiers. Because KV blocks never change once written, each keeps a valid copy on NVMe, so demotion moves no bytes and promotion reuses reads already in flight. Tenure is ~150 lines of user-space code that runs on unmodified Linux in standard cloud VMs. Across five workloads it serves no demanded bytes from NVMe, improves completion time 2.7× over the best baseline, and holds 99th-percentile time-to-first-token (TTFT) to 7–42 ms. In vLLM, the lease keeps NVMe off the TTFT path and cuts median TTFT by 1.30–1.60× against vLLM's own block pinning, which protects only the blocks of running requests from eviction.

CCS Concepts

• **Software and its engineering** → **Memory management**; • **Computer systems organization** → *Cloud computing*.

Keywords

cloud LLM serving, KV cache, memory tiering, CXL, placement policy

ACM Reference Format:
Amir Noohi and Antonio Barbalace. 2026. TENURE: Residency Leases for Tiered LLM KV Caches. In *ACM Symposium on Cloud Computing Volume 2 (SoCC '26 Vol 2.), November 18–20, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3828161.3857864



This work is licensed under a Creative Commons Attribution 4.0 International License.
SoCC '26 Vol 2., Singapore, Singapore
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2816-7/2026/11
https://doi.org/10.1145/3828161.3857864

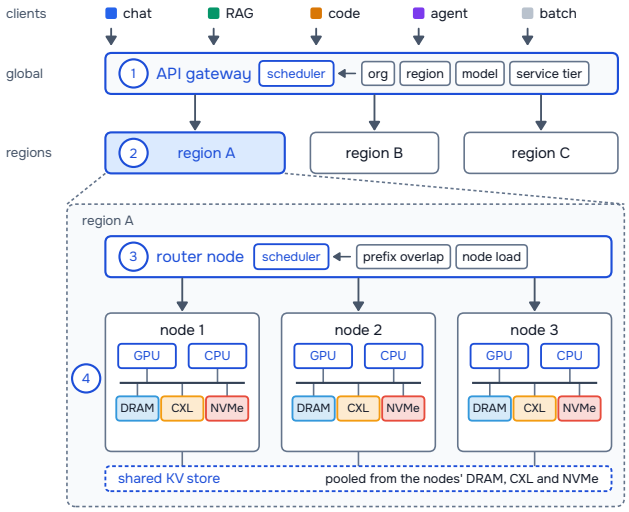

Figure 1: A request's path through a serving fleet.

1 Introduction

Motivation. Transformer inference caches the key and value vectors of every context token, the *KV cache*, and re-reads them for each generated token. Its footprint grows linearly in context length (24 GB at 128K tokens for a 34B-parameter model at FP16 [63]) and with the number of requests whose prefixes are retained for reuse. Production systems therefore extend the KV pool from GPU memory into DRAM and NVMe SSDs [59], and CXL-attached memory adds a tier between the two. Each tier holds bytes at a fraction of the price of the tier above, so where each block lives is a cost–latency decision the provider makes continuously (Figure 1). Serving engines manage the cache in fixed-size *blocks* named by content hashes, so requests that share a prefix share blocks (§2). When live KV exceeds fast-tier capacity (overcommit), some blocks must reside on NVMe, which on our host delivers 4.3 GB/s to the GPU against 27 GB/s from DRAM (Table 1). A *tiering policy* chooses those blocks, in the operating system [39, 73], in a KV-cache store [34, 59], or in the serving engine, as in vLLM, whose offloading tiers evict the least-recently-used (LRU) block [76]. A wrong choice is expensive, because a token cannot be emitted until every context block of its request has been read, so one NVMe-resident block stalls a step whose other reads finished in 1–2 ms.

The mismatch. Tiering policies were designed for applications whose future accesses are unknown, so they estimate them. NUMA balancing and TPP promote pages on recent hint faults [39, 73]; MEMTIS and HeMem sample accesses with precise event-based sampling (PEBS), a hardware performance counter [28, 60]; Colloid

balances measured access latency across tiers [77]; DAMON estimates the access rate of memory regions [54]. Each estimates the probability that a page is re-referenced soon, and each pays for the estimate twice, in sampling infrastructure and in the lag between a change in the working set and its detection. For KV blocks this probability need not be estimated. Under full attention, each *decode* step, which emits one token, reads the KV of all prior tokens, so a request with d steps left re-reads each of its context blocks exactly d more times. A queued request's blocks are named by the content hashes of its prompt [27, 93], known at enqueue, before its first read. The serving engine's scheduler therefore knows exactly the reuse that tiering policies estimate, from its queue of waiting requests and its batch of running ones. We derive two requirements. *(D1) Placement should consume the scheduler's declared reuse, not an estimate of it*, and *(D2) no placement action may serialize with a decode step.*

Why no existing interface suffices. Kernel interfaces cannot carry the schedule into placement. `move_pages` and `mbind` place a page once and record nothing afterwards, so a prefix that twenty live requests share looks the same as one no request needs. The `madvise` hints act only through reclaim [20], and the proposed `MADV_PROMOTE` was set aside because `move_pages` already exists [9]. DAMON's `DAMOS_MIGRATE_HOT` lets userspace request migration, but monitored access still decides which pages are hot [26]. The missing piece is a reference count on each block, held while a request needs the block and dropped when the request finishes. Serving engines know which blocks each queued request will read but do not count them. TensorRT-LLM's retention priorities expire after a fixed time-to-live (TTL) and govern only its GPU-to-host offload [51]; Mooncake's soft pins protect objects in its own pool [59]; vLLM's reference counts cover only running requests [27]. A queued request holds no count, yet the queue is where the tail of time-to-first-token (TTFT) is decided. Section 3 measures what this gap costs. Across 16 online policies, which rank blocks by the accesses they have already seen, 24–82% of *demanded* KV bytes, those a request is waiting on, come from NVMe at 3–5× overcommit, and migrations that leave placement unchanged still slow decoding.

Our work. Tenure is a ~150-line placement policy for the KV-cache pool of a serving engine. It runs in user space on an unmodified Linux kernel and needs no hardware performance counters, which cloud VMs may lack (§4.4). A *residency lease* is a reference count on each of a request's blocks, taken when the request enters the queue and again when it is admitted, and dropped when it completes, aborts, or is preempted. The policy keeps one invariant, that no leased block resides on NVMe (§4). A lease differs from vLLM's reference count, which we call a *block pin*, in three ways. It is taken while the request still waits. It keeps the block off NVMe but lets it move among VRAM, DRAM, and CXL, whereas a block pin keeps a block in its tier. It expires after a TTL, so a failed client cannot hold blocks forever. Three mechanisms make the invariant cheap. *Queue-time prefetch* reads a queued request's NVMe-resident blocks while the request ahead of it decodes, so their transfer stays out of its TTFT (§4.1). KV blocks are *sealed*, never written again once their token span fills, so each block keeps a permanent NVMe copy, a *shadow*, and demoting it moves zero bytes (§4.2). Promotion happens only when it costs one copy into CXL, after a demand read has already brought the block into host memory (*piggyback*) or during queue-time prefetch. When leases exceed fast-tier capacity, Tenure sheds the invariant in a defined order and counts each leased block forced to NVMe, a *spill* (§4.3). Every mechanism runs off the decode path, so D2 holds by construction. The same lease also runs in vLLM as a plugin of its KV-connector interface, through which vLLM loads and saves KV in an external store (§5).

Contributions.

- **Analysis** (§3). A study of 17 policies (16 plus Tenure) over five workloads on a physical GPU/DRAM/CXL/NVMe host, in a replay engine that issues the block reads of serving on the host's tiers without running a model and checks every run against eight invariants (§5). It finds (i) *placement blindness*, in which policies that disagree on every other metric all serve 24% or more of demanded bytes from NVMe, because none observes the queue, and (ii) a *churn tax*, in which migrations that provably leave placement unchanged slow decoding in proportion to the bytes they move, consistent with weighted-caching results in which miss count and cost diverge [1, 7, 90].
- **Design** (§4). The residency lease and its composition with the two migration mechanisms that sealed blocks enable, zero-byte demotion and piggybacked promotion.
- **Evaluation** (§6). In the replay engine, Tenure serves no demanded bytes from NVMe on any workload, improves completion time (*makespan*) 2.7× over the best baseline, and holds the 99th-percentile TTFT (p99) within 7–42 ms, results we test with a mechanism ablation, five repeated runs, and batched serving past the invariant's capacity. In vLLM 0.29, under the lease no request reads NVMe on its TTFT path, and median TTFT falls 1.30–1.60× against vLLM's own block pinning at 2–10× overcommit (§6.7).

2 Background

LLM serving. A *serving engine* such as vLLM or SGLang [27, 93] runs the model and manages its KV cache. A request supplies a prompt and streams back generated tokens in two phases. *Prefill* ingests the prompt, loading whatever prefix KV is already cached and computing the rest, and emits the first token; its latency is the *time-to-first-token* (TTFT). *Decode* then generates one token per step; with full attention each step re-reads the KV of the entire context, and the gap between tokens is the *time-between-tokens* (TBT). Service-level objectives constrain the tails of both, so we report medians (p50) and 99th percentiles (p99). The engine's *scheduler* keeps arriving requests in a *queue* and admits them into a *running batch* as slots free; a request leaves the batch when it emits its end-of-sequence (EOS) token or reaches its length limit. *Speculative decoding* [30] drafts several tokens cheaply, with a small draft model or from n-gram matches in the prompt, and the model then verifies all of them in one step, keeping the draft tokens it accepts and discarding the rest, together with the KV written for them. A run's *makespan* is its end-to-end completion time over the full request sequence.

KV blocks. Serving systems manage KV state in fixed-size blocks indexed by content hash, so shared prefixes map to shared blocks [27]. A block is *sealed* (immutable) once its token span is full. We use 24 MB blocks, the KV of one 128-token span of a 34B

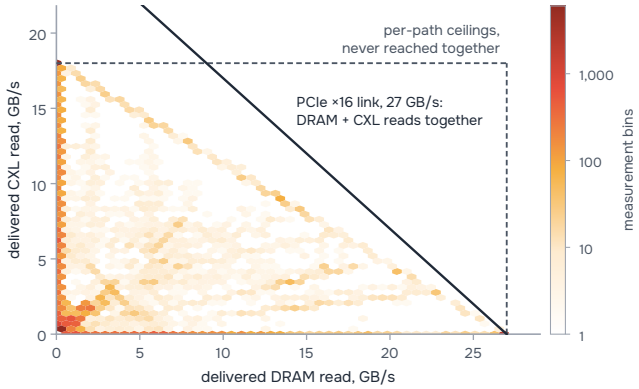

Figure 2: Delivered DRAM vs. CXL read bandwidth per time bin, all 85 study runs.

grouped-query-attention (GQA) model at FP16 (Code Llama 34B [63], with 48 layers of 8 KV heads of 128 dimensions); Llama-3-70B [19] yields ~40 MB and multi-head latent attention (MLA) models such as DeepSeek-V3 [11] ~9 MB, scaling the problem without changing its shape. An engine keeps a block from being evicted while requests use it. vLLM keeps a reference count on each block and evicts only blocks whose count is zero [27]; LMCache offers the same protection through explicit pin and unpin calls, and Mooncake through soft pins that expire after a TTL [34, 59]. We call this protection *block pinning* and a held reference a *block pin*; a pinned block cannot be evicted from its tier. Block pinning acts on KV blocks and locks no pages, so it is unrelated to OS memory pinning (`mlock`) and to page-locked DMA buffers.

Tiered memory. A modern inference node locally exposes up to four memory levels: GPU VRAM, DRAM, directly attached CXL memory, and an NVMe SSD. Linux manages the middle tiers as NUMA nodes: NUMA-balancing tiering promotes pages on hint faults, reclaim demotes cold pages downward, and DAMON schemes migrate monitored regions [54, 73]. Userspace software can place its own pages today on specific NUMA nodes with `mmap`, `mbind`, and `move_pages`, the system calls Tenure builds on (§4.4).

Memory pressure. Tiering only matters under memory pressure, which clouds create by overcommitting hosts [8, 86]. We therefore cap the KV pool's usable capacity in each fast tier, at 4 GB of VRAM, 8 GB of DRAM, and 8 GB of CXL, matching the DRAM cap, and back it with an NVMe tier large enough to hold every working set, so that our 53–108 GB working sets run at 3–5× overcommit. The caps are deliberate scaling, not the machine's size: what carries over to production is the ratio, which is the one a node reaches when its KV pool outgrows DRAM (§7 discusses this scaling). Our working sets are 0.06–0.25 GB per request (Table 2), so 10,000 retained requests occupy 0.6–2.5 TB, 1–10× the DRAM of a 256–512 GB host. We also report 4–9× without the CXL tier (§7) and 2–10× in vLLM (§6.7). Table 1 gives each tier's installed capacity before our caps, its idle access latency (one dependent cache-line load, from a GPU thread for VRAM and a CPU core for DRAM and CXL, and one 4 KiB read for NVMe), its read bandwidth to the GPU, and the time to move one 24 MB block; NVMe writes run at 2.2–5.0 GB/s. One topological fact shapes everything: every

Table 1: Measured tiers (RTX 3070; Samsung CXL 1.1+ Type-3; Samsung 990 EVO Plus).

tier	capacity	latency	GB/s to GPU	ms/block
VRAM	8 GB	295 ns	~405	0.06
DRAM	512 GB	137 ns	27	0.89
CXL	128 GB	277 ns	18	1.33
NVMe	1 TB	58 μs	4.3	5.58

byte that reaches the GPU, from DRAM, CXL, or NVMe, crosses the same PCIe Gen4×16 link. The GPU, the DRAM tier and the NVMe drive sit on the first socket. CXL memory is one Samsung CXL 1.1+ Type-3 expander, attached without a switch to a ×8 link of the second socket, so GPU reads of it also cross the inter-socket link; they still reach the device's own 18 GB/s, which CPU loads on its socket also measure. The tiers therefore do not add bandwidth: CXL contributes capacity, not throughput, and a byte read from NVMe occupies the shared path about six times longer than a byte from DRAM. We verified these upper bounds empirically in Figure 2, which reports the DRAM and CXL bandwidth delivered in each time bin while running the 17 tiering policies over five workloads (§3); no bin exceeds the shared link.

3 Analysis: Tiering Meets the KV Cache

We first measure how much it costs that tiering policies estimate a reuse pattern the serving engine already knows (§1). Our replay engine (§5) replays the memory traffic of serving token by token on our host: every block access is a `cudaMemcpy`, or an O_DIRECT `pread` that bypasses the page cache, from the tier that holds it. It models *KV-offload streaming*, in which the context that does not fit in GPU memory is read from the host tiers at every decode step [65]; §7 discusses engines that decode from GPU memory. Throughout, *demanded* bytes are bytes a prefill or decode step is waiting on; prefetch and migration traffic are accounted separately (§5). We evaluated 17 policies: 16 online policies from prior work, drawn from the six estimator families of Table 3, and Tenure itself. The 16 are `lru`, `fifo`, `autonuma` (Linux NUMA balancing [73]), `tpp` [39] and `tmts` [14] for recency; `lfu`, `memtis` [28] and `hemem` [60] for frequency; `w-interleave`, Linux's weighted interleave [57], for bandwidth; `arc` [40] and `lirs` [22] for reuse distance; `lecar` [74] for learned policies; and `evolved`, `vulcan`, `funsearch` and `value-density`, four ranking formulas found by automated search in the manner of FunSearch and Vulcan [15, 62], each reimplemented over KV blocks in our engine (§7). The block-pinning row of Table 3 (vLLM's reference counts, the block pinning of §2) enters as an ablation variant (§6.2) and as the baseline in vLLM (§6.7). Five workloads cover reuse concentration, the share of accesses that hit the hottest 1% of blocks, from 0.04 to 0.67 (Table 2; Figure 3). Three come from public request logs of production services. conversation (multi-turn chat) and toolagent (tool-using agents) are traces of the Kimi chatbot, served by Mooncake [59], and swissai takes 1,000 Qwen3-32B requests from the SwissAI serving trace [88]. Two are synthetic. rag prepends passages retrieved from a shared corpus to each question, and uniform, our control, asks about equally popular documents, so no block is hot. The spread matters: an estimator

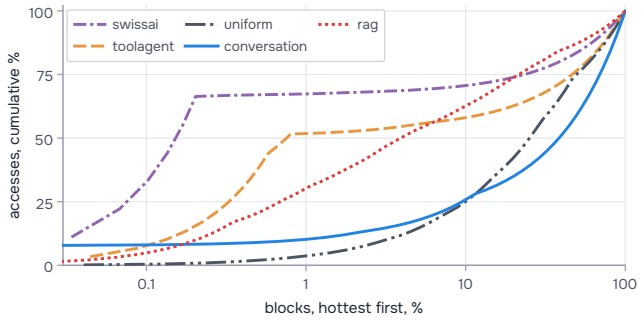

Figure 3: Access concentration of the five workloads.

Table 2: Workloads (WS = working set; reuse = share of accesses to the hottest 1% of blocks).

workload	blocks	accesses	WS (GB)	reqs	reuse
swissai	2 950	45 200	69	1 000	.67
toolagent	2 250	43 425	53	400	.52
uniform	2 472	45 000	58	1 000	.04
conversation	4 169	46 188	98	400	.10
rag	4 592	90 000	108	1 000	.30

Table 3: Estimator families and the input each ranks by.

family (examples)	input	uses schedule?
recency (`lru`, `tpp`)	access order	no
frequency (`lfu`, `memtis`)	access count	no
bandwidth (wt. interleave)	static ratio	no
reuse-dist. (`arc`, `lirs`)	inter-ref. gap	no
learned (`lecar`)	expert regret	no
automated search (`vulcan`, `funsearch`)	searched formula	no
block pinning (vLLM refcounts)	active set	running only

family can look adequate on concentrated reuse (swissai) and collapse on capacity-bound traces (conversation, rag), so all findings below are reported per workload.

3.1 Placement blindness

Figure 5(a) shows the problem for `lru` on one workload (swissai, reuse 0.67, the highest): execution alternates between NVMe-bound phases and short fast bursts. The demanded-byte numbers behind it are uniform across families: online policies from four different families each stream ~24% of *demanded* KV bytes from NVMe, whose peak read bandwidth is 4.3 GB/s (Table 1; full matrix in Figure 9). The convergence across families is the substantive result: recency, frequency, interleaving, and reuse-distance estimators produce different rankings on nearly every other metric, yet coincide within 0.2 percentage points here, because the miss set is decided by information none of them considers, namely which prefixes the *next* requests will read, rather than by estimator quality. Subfigure (a) also shows where the time goes: a decode token is a barrier over its context blocks, so one NVMe-resident block (5.6 ms alone, ~10 ms when two reads share the drive) stalls a token whose other reads finished in 1–2 ms, idling the fast path behind an NVMe drive

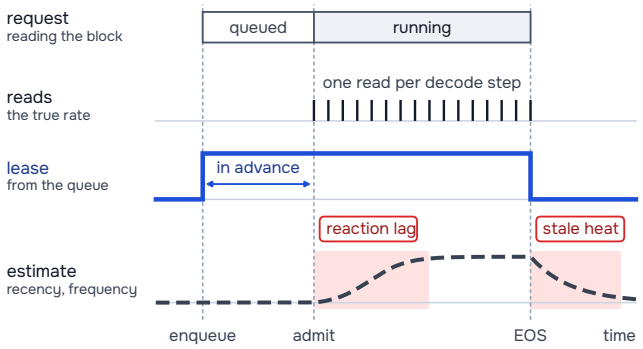

Figure 4: A block's re-reference rate vs. its lease count and a recency/frequency estimate.

serving one or two outstanding reads. The NVMe drive is the bottleneck and is still underutilized. The two problems are distinct and Figure 5 shows both, for every policy in the study: fast-path *underutilization* (subfigure b: delivered DRAM+CXL is 3% of the shared link under `lru` on swissai and below 11% for every online policy, because NVMe reads bottleneck token processing) and *bad placement* (subfigure c: every online policy's TTFT p99 sits at 59–481 ms, because a queued request's blocks are still on NVMe when its prefill begins).

Figure 4 shows why no estimator can address this problem. A block's true re-reference rate changes only at events of the serving engine's scheduler: it jumps when a request naming the block enters the queue or is admitted, and it drops to zero when the last request using the block finishes with its end-of-sequence (EOS) token. The lease count changes at exactly these events, because the engine increments and decrements it there. An estimate built from recency or frequency instead rises only after the block has already been accessed several times, which we call *reaction lag*, and stays high after EOS, when the block is already dead, which we call *stale heat*. The misses of Figure 5 occur inside those two windows.

Insight #1. *Placement blindness is independent of the estimator family. Policies that disagree on everything else agree on stranding demanded KV on NVMe, because no quantity inferred from past accesses can reveal which blocks a queued request will read.*

3.2 The churn tax

Placement quality is not the only cost. Policies that react quickly migrate often, and the bytes those migrations read and write cross the same links as demand reads. Across the 16 online policies, migrations correlate with decode latency (r=0.66 pooled; 0.61–0.85 within each workload; Figure 6). Correlation pools confounds, so we isolate causality with a controlled injection: into `lfu`, which rarely migrates, we add dummy migrations, random *same-tier* block pair-swaps that move data but provably leave placement unchanged, and sweep the injection rate. Placement neutrality is verified directly: the NVMe demand-read share is constant to the decimal at every injection rate (24.2% on swissai, 72.3% on conversation). TBT nonetheless rises steeply, from 21 to 210 ms on swissai and 99 to 450 ms on conversation (Figure 7): data movement alone, with

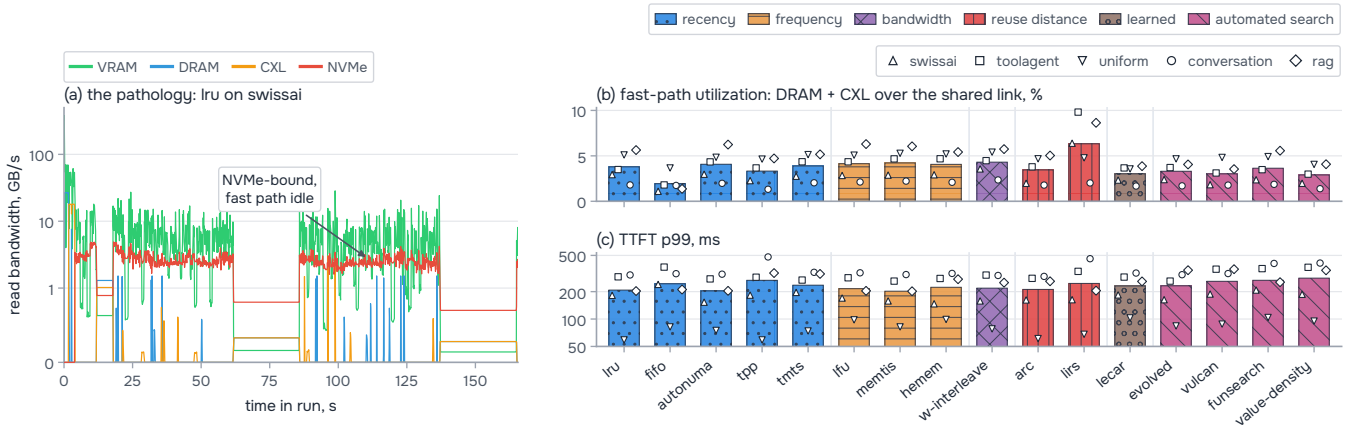

Figure 5: The 16 online policies: (a) read bandwidth by tier, `lru` on swissai; (b) fast-path utilization (bars = mean, markers = workloads); (c) TTFT p99.

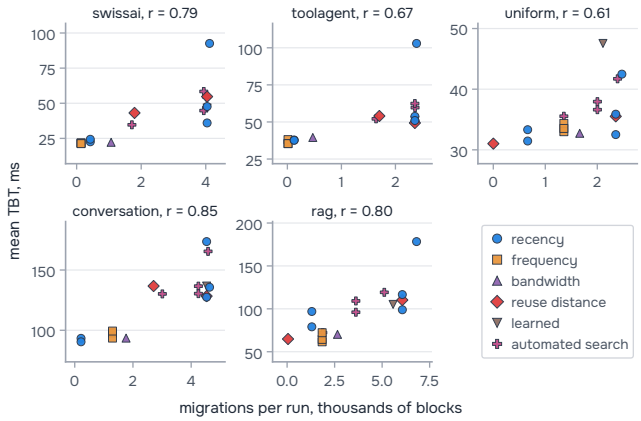

Figure 6: Migrations vs. mean TBT, per workload.

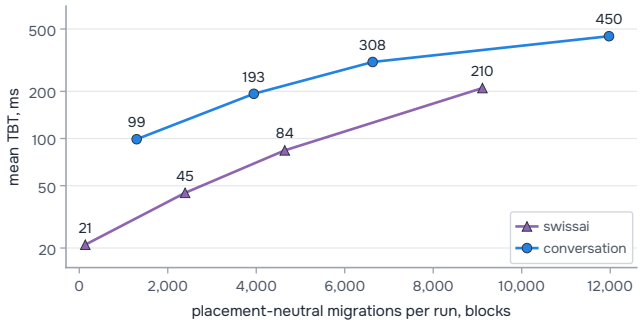

Figure 7: TBT vs. rate of dummy same-tier migrations injected into `lfu`.

placement quality held fixed, degrades decode latency. Stable frequency policies (~929 moves per run) hold TBT near 51 ms; the recency policies that reshuffle placement after every shift in access order (`lru`, `fifo` and `tpp`, ~3942 moves) pay ~88 ms.

Insight #2. *Migration is not a panacea; it is a tax on decode. A tiering policy for the KV cache must move blocks rarely, or make moves cost no link time, because every byte a migration moves delays decode; the objective is bytes over slow links plus migration cost, not miss count.*

Blindness demands that placement consume declared reuse (D1), and the churn tax demands that the moves placement makes cost almost nothing (D2).

4 Tenure Design

Tenure manages a pool of sealed KV blocks across multiple tiers. Each block carries three bytes of policy metadata: a 16-bit saturating lease count and a shadow-validity flag (§4.2). Like any tiered pool, Tenure also records each block's tier and its slot in that tier; a block's NVMe copy needs no recorded address, because block b always occupies slot b of one preallocated file (§4.2). The serving engine drives placement through two calls at three points, `lease` at enqueue and at admission and `release` at completion, each given a list of blocks. The serving engine already computes their arguments: the content-hash block list of each request, and per-block reference counts for the requests in its running batch [27]. The policy maintains one invariant:

> *I1: no block b with* lease$[b] > 0$ *resides on NVMe, except during declared saturation (§4.3).*

I1 operationalizes D1 (consume declared reuse); the transition costs specified below operationalize D2 (never serialize with decode). I1 constrains only the slowest tier. Leased blocks migrate freely among VRAM, DRAM, and CXL under the same frequency-ranked *passive layout* that orders unleased blocks (§4.2). I1 rests on no prediction. Full-attention decode re-reads a request's whole context at every step, so every leased block will be read again once per remaining decode step, and the set of leased blocks is exact at every instant. Figure 8(a) gives the state diagram; Table 4 the pseudocode.

4.1 Residency leases

A lease is a per-block reference count: a prefix shared by k live requests carries count k, and the count saturates rather than wraps.

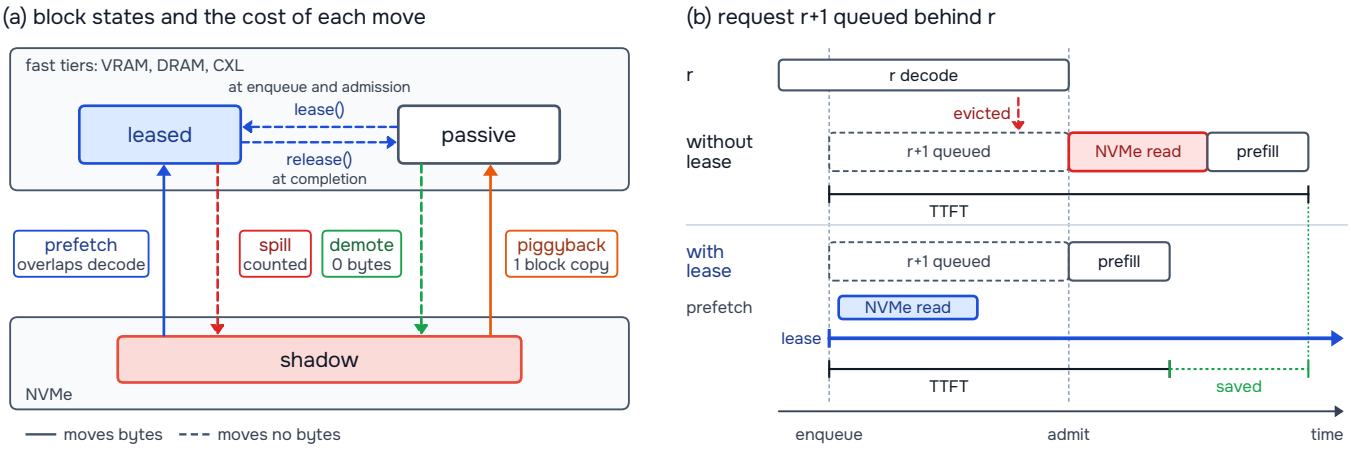

Figure 8: (a) Block states and move costs; (b) request r+1 queued behind r, with and without the lease.

Acquisition happens twice. At *admission*, the running request's context is counted; vLLM's block manager already keeps this count, `ref_cnt`, on the blocks of running requests, which is the block pin of §2. At *enqueue*, the request's blocks are counted while it still waits in the queue and holds no execution resources; this is the half that does not exist in current engines and that carries the tail benefit. Release is bound to the request lifecycle: completion, client abort, and preemption all decrement (a preempted request re-acquires on resumption), and a TTL bounds leakage from hung clients. Unlike a block pin, which holds a block in its tier until the request finishes, a lease keeps the block only off NVMe, lets it move among VRAM, DRAM, and CXL as the passive layout ranks it, and expires after its TTL. Because taking a lease costs nothing but holding leased blocks off NVMe takes fast-tier capacity, leases are a resource: within one serving engine, admission control bounds active contexts and therefore bounds the leased footprint; cross-tenant arbitration is out of scope here and discussed in §4.4. The lease needs only a waiting queue and a running batch, so it applies with or without prefill–decode disaggregation [55, 96].

Enqueue-time acquisition converts queue lookahead into transfer overlap. When request r+1 enters a queue of depth 1, its NVMe-resident blocks are promoted into CXL concurrently with r's decode; the transfer is I/O accounted in wall time, but it occupies the NVMe link during a window in which prefill is not waiting on it. Tenure prefetches only the head-of-queue request: deeper entries have a full predecessor-service-time budget remaining, so prefetching them earlier buys nothing and risks eviction before use. At queue depth 0 the mechanism is inert and Tenure falls back to admission-time leases; queue depths of 1 or more are precisely where TTFT tails form.

4.2 Shadows and piggyback

Sealed KV blocks are write-once. Tenure exploits this with a *shadow* discipline: block b's NVMe copy always sits in slot b of a preallocated file (slot(b) = b), so the shadow store needs no allocation map, no compaction, and no coherence protocol: a block written to NVMe once has a valid copy there for its lifetime (once set, shadow[b]=1 never clears). Demoting a shadowed block then only updates the record of its tier and slot, and zero bytes cross any link. Linux's swap cache follows the same rule for clean pages, dropping a page without writing it back when its swap copy is still valid; sealed blocks are always clean, so Tenure applies the rule unconditionally. This is non-exclusive tiering in the sense of NOMAD [80] with the invalidation path deleted rather than handled. The one mutable block per sequence (the unsealed tail, appended each token) is excluded from the shadow discipline and pinned in VRAM until sealed, costing B×24 MB of VRAM at batch depth B. The shadow discipline therefore requires the serving engine to seal a block only from verified tokens, those the model has accepted. Speculative decoding (§2) is the one common case in which an engine writes KV for tokens it may discard. A block that holds unverified draft tokens is not yet sealed, so a rejected draft is rolled back before its block seals and never reaches a sealed block. vLLM caches a block only once its tokens are final, and §6.7 checks the guarantee with speculation enabled. A draft model with its own KV cache is one more leased tenant.

Promotion never initiates a slow-tier read of its own. It occurs on two occasions. *Piggyback*: when a prefill or decode step reads a block from NVMe because its request needs it now (a demand read), the read lands first in an aligned staging buffer in host memory, a bounce buffer, as O_DIRECT requires; keeping the block in CXL then costs one copy of its 24 MB from that buffer, issued after the token completes. *Queue-time prefetch*: as above. Both paths select victims with the same routine, and both must handle the same race: victim selection and data movement run concurrently on the replay engine's parallel I/O threads (§5), so a block one thread is promoting must not be selected as a victim by another before its copy completes. Tenure resolves this with a reserve-then-issue protocol: the placement transaction (victim chosen, the tier-and-slot records of the promoted block and the victim swapped) executes serially and first; the data transfers it decides on are then issued in parallel. Because the placement state is committed before any I/O begins, a concurrent scan observes post-transaction residency and cannot double-allocate a slot.

Table 4: Complete placement pseudocode.

```
on enqueue(r):
  for b ∈ ctx(r): lease[b]++
  if r is head of queue:
    for b ∈ ctx(r) with tier[b]=NVMe:
      v ← victim(); commit placement; issue I/O
                              (overlaps predecessor decode)
on demand read of b from NVMe:   (first touch only)
  serve read; v ← victim(); commit; write b to v's slot
          (+1 block copy into CXL, after token completes)
victim():                                  (victim order)
  prefer: lease=0 ∧ shadow=1        (demote = remap, 0 B)
  then: lease=0, coldest first          (one shadow write)
  last: lease>0, ascending (count, freq); spills++
on release(r): for b ∈ ctx(r): lease[b]−−
  (no eager motion; cold blocks demote lazily, by remap)
```

Victim selection is a two-pass linear scan over CXL-resident blocks, in the victim order of §4.3; with C blocks in CXL it is $O(C)$ per admission batch ($C \leq 342$ here, microseconds in practice) and amortizes over the batch. Blocks with lease$[b]$=0 follow a *passive layout* ranked by cumulative access frequency, the ranking that caused the least migration in our study (§3), so cross-request-popular prefixes ascend once and are not reshuffled by arrival order.

4.3 Saturation semantics

At batch depth B with mean context P blocks, the leased footprint is $\approx BP \times 24$ MB and exceeds fast capacity at $B \approx 64$ for our conversation workload. I1 is then unsatisfiable, and *how* a policy sheds an unsatisfiable invariant is part of its specification. Tenure spills along the victim order of Table 4: unleased-and-shadowed blocks (free), then unleased by ascending frequency (one shadow write), and only then leased blocks by ascending (lease count, frequency), so a prefix held by twenty requests outlives one held by one, and the blocks of the least-shared, least-reused request go to NVMe first. Every leased spill increments an exported counter, which turns saturation from a latency anomaly into a capacity metric an operator can shed load or scale out on. At full saturation (every resident block leased) Tenure degenerates, by construction, to its passive frequency layout plus the spill order; §6.4 measures that regime.

4.4 Cloud deployment on stock Linux

Tenure runs without kernel modification, deploying a new placement policy as user-space software, much as NFV deploys new network protocols as software functions [25]. The pool is `mmap`ed per tier (DRAM and CXL are NUMA nodes, so placement uses `mbind` at allocation and `move_pages` for migration), and NVMe I/O is O_DIRECT against a preallocated file whose block-ID-indexed layout *is* the shadow store. No PEBS, uncore access, or page-table scanning is required, so Tenure deploys inside standard cloud VMs, where sampling-based tiering (MEMTIS/HeMem/Colloid-class) may lack the hardware counters it relies on. A kernel-native lease would need a home for the count (`struct page` has none), mlock-style accounting against holders that never release it, and a cross-cgroup spill order.

5 Implementation

We built two prototypes of Tenure. The first runs its ∼150-line policy inside a replay engine, which we use for the study of §3 and for §6.1–§6.6; the second runs the lease inside vLLM, for the end-to-end evaluation of §6.7.

Replay engine. The replay engine is ∼500 lines of CUDA/C++ for each of its two modes, serial and batched. It runs no model. It replays a trace's block reads on the tiers of Table 1 and times each read, and every policy, Tenure included, plugs into it and decides where each block lives. The *serial mode* replays token by token: a decode step issues its request's P context-block reads together over 16 CUDA streams (VRAM/DRAM/CXL via `cudaMemcpyAsync`) and 16 OpenMP I/O threads (NVMe via O_DIRECT `pread` into 4 KiB-aligned bounce buffers), and completes when the slowest read lands, as an attention step cannot start until all of its context blocks are loaded. The threads keep up to 16 of a step's reads in flight at once; a single 24 MB read already reaches the drive's 4.3 GB/s (Table 1), so an asynchronous interface such as `io_uring` would save CPU time but change neither the bandwidth nor which tier serves a read. The *batched mode* implements closed-loop continuous batching: an iteration reads the deduplicated union of all B active contexts, slots refill from the queue on completion, and TTFT is clocked from admission. All 17 policies execute on one shared runtime (identical epoch re-ranking, identical migration engine, identical I/O paths), so differences between policies reflect their placement decisions alone, not differences of implementation. Tenure adds its 3 bytes/block of metadata and the routines of Table 4; each mechanism sits behind an independent flag (`--shadows`, `--piggy`, `--qpre`, `--pinctx`), which is what makes the ablation of §6.2 a controlled experiment.

We verify every run after it completes. Each run emits per-tier bandwidth timelines, latency percentiles, and migration and spill counters, and is checked against eight invariants: read shares sum to 100%; bytes equal accesses×block size; each tier's demand-read bandwidth integrates to its demand bytes within 1%; throughput stays within the measured link bandwidths; no request finishes faster than its P blocks can cross the 27 GB/s link; residency slots are unique and within capacity; I/O errors are zero; and no leased block is demand-read from NVMe (I1). Counters of issued `pread` bytes and of paid promotions cross-check the event stream.

vLLM. Our second prototype runs Tenure in vLLM 0.29 [27] as a plugin of its KV-connector interface, about 400 lines of Python, both baselines included; vLLM itself is unmodified. The connector manages a host store of two tiers, a capacity-bounded pool of page-locked DRAM over one O_DIRECT NVMe file per block, and maps the lease onto connector hooks. When a request enters vLLM's queue (`on_new_request`), the connector leases the request's leading prompt blocks that the store holds and starts reading those on NVMe into DRAM. When the scheduler admits the request, a load copies the blocks into GPU memory before the forward pass that uses them. When vLLM reports the request finished by completion or abort (`request_finished`), the connector releases the lease. DRAM never evicts a leased block. Every block is written once to NVMe when first computed, so evicting it from DRAM moves no bytes (the shadow of §4.2), and a block read from NVMe stays in DRAM (piggyback). Blocks are named by a hash chained over their prefix's token IDs, so prompts that share a prefix share blocks. A

Table 5: Means over five workloads (wall = makespan, averaged over repeated runs where repeated).

policy	wall (s)	TTFT p99	TBT	NVMe
Tenure	**87**	**29 ms**	**19 ms**	**0%**
w-interleave	246	218 ms	52 ms	43%
hemem	249	223 ms	52 ms	43%
memtis	233	202 ms	50 ms	43%

load reads the request's host blocks with a pool of 8 threads, gathers them into one page-locked staging buffer and copies them to the GPU with one indexed copy per layer. Under the lease a load finds its blocks already in DRAM (§6.7), so the size of this pool is off the critical path. The store keeps DRAM and NVMe tiers, unleased DRAM blocks leave in LRU order, and every arriving request's stored blocks are prefetched.

The no-CXL and compute-overlap analyses of §7 come from separate runs of the replay engine, with `lru`, `lfu` and Tenure on traces generated with the statistics of Table 2 and the same audits, so we compare these runs with each other.

6 Evaluation

We evaluate Tenure first in the replay engine of §5, against all 16 online policies (§6.1–§6.6), and then end to end in vLLM, a production serving engine, where attention, continuous batching, compute–transfer overlap and the request lifecycle are vLLM's own and only the host store is ours (§6.7).

6.1 Headline results

Table 5 and Figure 9 give the full results: NVMe share, TTFT p99, mean TBT and makespan for every policy on every workload. Tenure is first among the 17 policies on every workload and metric in this regime. Mean makespan is 87 s, 2.7× better than the best baseline mean. TTFT p99 is 7–42 ms, 5.5–8.4× below the best baseline per workload. TBT is 2.6× below the best baseline mean, and NVMe demand reads are 0% on every workload. The tail structure follows (Figure 10): across all 85 policy–workload combinations every online policy's TTFT p99 sits well above its p50, because demotion victims pay the full NVMe penalty at prefill, while Tenure's p99 tracks its p50 (on uniform, both are 7 ms), since prefill residency is established before admission. Figure 12 shows where the time goes, for one policy from every estimator family. Every baseline is NVMe-bound for most of its run: each step waits on one or two NVMe reads, so the drive delivers 1–4 GB/s, mostly below its 4.3 GB/s peak, while DRAM and CXL deliver about 1 GB/s or less each. Tenure is faster because queue-time prefetch moves each queued request's blocks into CXL while its predecessor runs. Its steps then read from CXL at 7–10 GB/s. It still reads about 1 GB/s from NVMe, but every one of those reads is a prefetch for a queued request, and no step waits on it. It completes the swissai trace 3.5× sooner than `lru`. Figure 11 gives both margins per workload: makespan is 2.2–3.2× shorter than the best baseline's.

6.2 Ablation: mechanisms vs. semantics

The shadow and piggyback mechanisms work under any placement policy, so to attribute the gains we grant them to the baselines as well. The variants add Tenure's mechanisms one at a time to two baselines: `lfu`, and `pinlru`, which is LRU with the blocks of running requests pinned, as vLLM does. Table 6 gives every variant's results per workload; Figure 14 summarizes them, and its shape is the finding: no variant shortens both bars except Tenure. *Mechanisms carry throughput and decode latency*: `lfu` with shadows and piggyback reaches 46 s, comparable to Tenure's 47 s, and its mean TBT is about half of Tenure's on every workload (6–20 vs. 11–32 ms): Tenure pays for its own prefetch and piggyback traffic during decode, the churn mechanism of Insight #2 applied to our own system. Its tail, however, stays near 163 ms, 6× worse, because promotion still happens after the demand read that paid the NVMe price. *Queue-time prefetch carries the tail*: the lease-free variant (+ queue-prefetch) reaches 28 ms, within 1 ms of Tenure, on every workload. *The lease makes prefetch effective*: without a lease, prefetched blocks are evicted before use and re-fetched, costing 84 s of makespan against Tenure's 47 s; the reference count is what lets declared prefetch survive contention. Block pinning of active contexts alone, which vLLM's reference counts provide, is the weakest variant (210 s), since without shadows every demotion of an unpinned block still writes the whole block to NVMe. Adding a mechanism does not always help: queue-prefetch on top of `pinlru` *worsens* rag by 61%, because unprotected prefetch evicts useful residents. No variant can reproduce Tenure's combination of makespan, tail, and zero demand reads; Tenure's contribution is the composition. The tail mechanism itself (queue prefetch) is CachedAttention's [17], made safe by counting.

6.3 Variance across runs

We ran the swissai comparison of Tenure and `hemem` five times: Tenure takes 50.9±5.4 s and `hemem` 105.0±1.8 s (mean ± 95% t-interval). Repeated runs also exposed a measurement hazard. One early run of a write-heavy policy finished far sooner than that policy's mean over repeated runs, because the drive's SLC write cache, the fast region an SSD absorbs writes into until it fills, kept its state from the run before. We therefore interleave runs of different policies, run the closest comparisons several times, and report the hazard here. Three runs each of the noisiest online measurements bound their spread: the widest, `arc`'s makespan on uniform, varies by ±158 s, less than the 185 s by which every baseline trails Tenure.

6.4 Concurrency at the invariant's limit

The replay's batched mode (§5) serves B concurrent requests in a closed loop, with TTFT clocked from admission. Where the leased footprint fits, Tenure holds NVMe demand reads near zero at every depth (swissai B=16: 0% vs. 64% for `hemem`) with iteration latency (ITL) several times lower (72 vs. 330 ms). Lease spills begin before full saturation (69 spills already at conversation B=16), and the pinned unsealed tails cost B×24 MB of VRAM (1.5 GB at B=64). At B=64 on conversation the leased footprint (21.2 GB) exceeds the 20 GB fast tiers; the invariant is arithmetically unsatisfiable, and Tenure degrades to 35% NVMe with 159 counted spills, against 76% for `hemem`. At full lease saturation the advantage narrows to

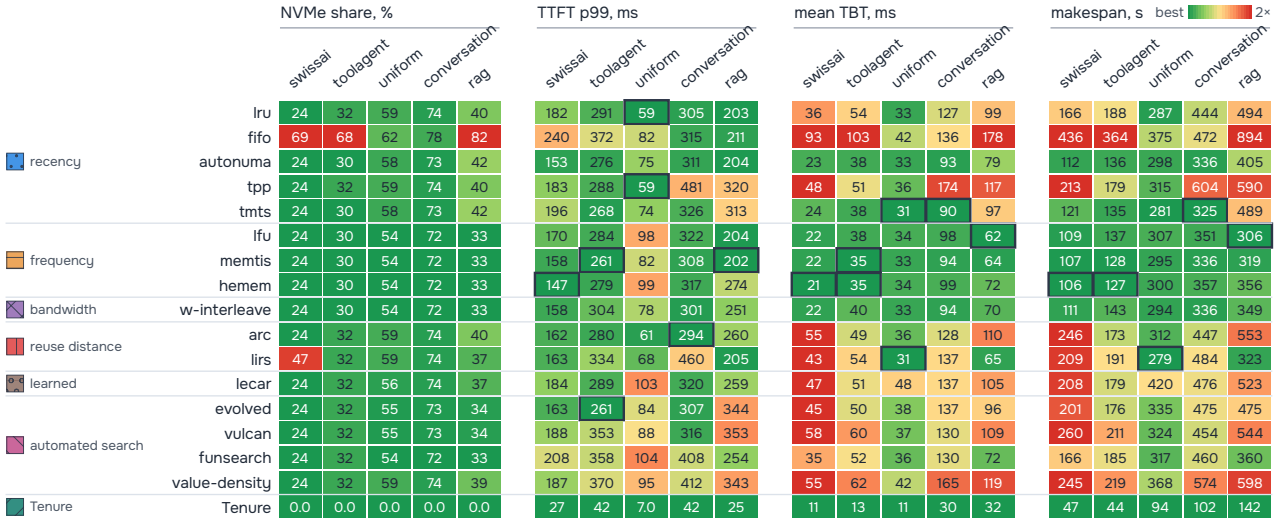

Figure 9: All 17 policies on all five workloads (shade = gap to the best baseline, outlined).

Table 6: Mechanism variants: makespan (wall, s) and TTFT p99 (ms).

	swissai		toolagent		uniform		conversation		rag	
variant	wall	p99	wall	p99	wall	p99	wall	p99	wall	p99
`lfu` + shadows	109	154	126	276	255	60	333	286	319	211
+ piggyback	46	163	44	276	93	77	105	327	137	212
+ queue-prefetch	84	28	72	42	164	7	209	43	221	25
`pinlru` (vLLM pins)	210	139	164	281	316	60	483	341	488	201
+ queue-prefetch	207	27	135	42	285	7	387	43	785	25
TENURE (full)	**47**	**27**	**44**	**42**	**94**	**7**	**102**	**42**	**142**	**25**

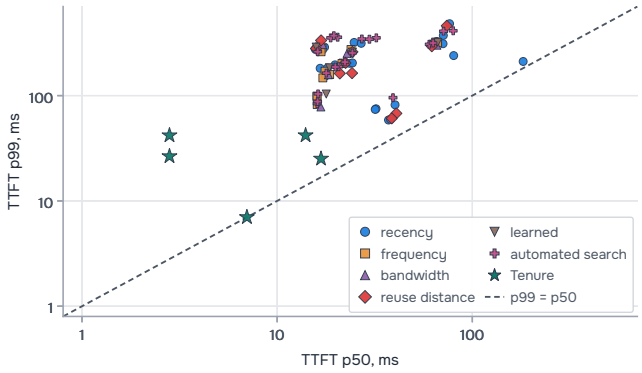

Figure 10: TTFT p50 vs. p99 (stars = TENURE).

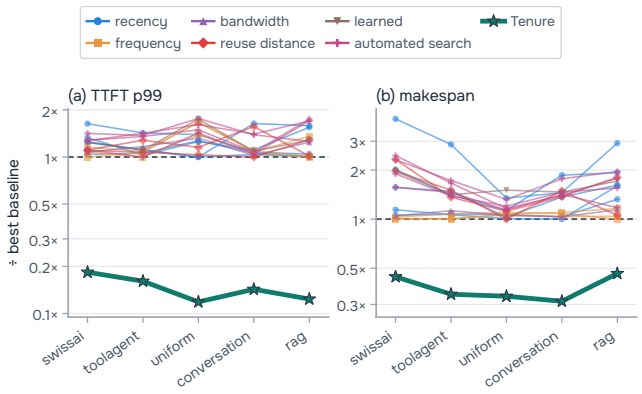

Figure 11: TTFT p99 and makespan relative to the best baseline, per workload.

the passive layout and victim order, exactly as §4.3 predicts, and the spill counter makes the transition observable (Table 7).

6.5 Decode length and the fast-tier bottleneck

We swept decode length by replaying each request's context D times, $D \in \{4, 16, 64\}$ on swissai. TENURE's NVMe demand reads stay at 0% and its TTFT p99 stays flat at every D, but its makespan grows (45/101/273 s). Long decode makes the workload fast-tier-bandwidth-bound: the 8-block context fits in VRAM, yet TENURE's mechanisms promote to CXL, leaving 26% of reads on the 18 GB/s tier rather than VRAM's ~405 GB/s. The remedy follows our thesis: the active context is *declared* maximally hot, so its lease can target VRAM rather than merely keep it off NVMe.

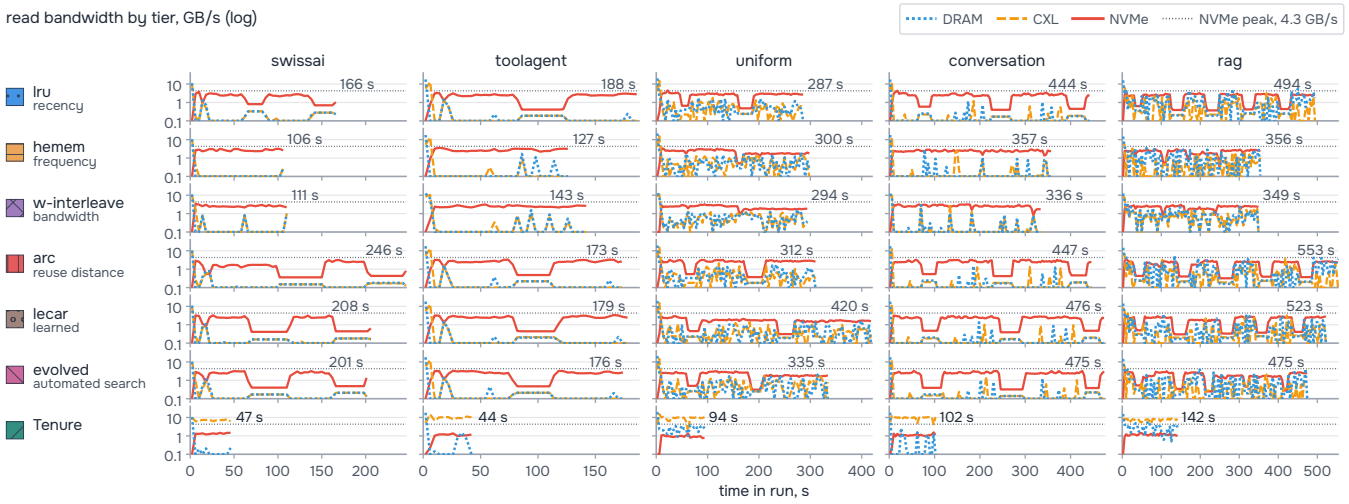

Figure 12: Per-tier read bandwidth (4 s means), one policy per estimator family vs. TENURE.

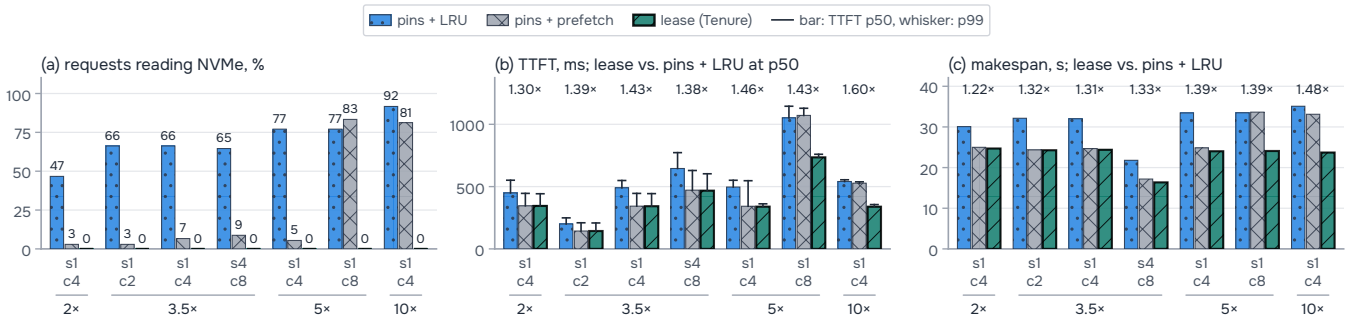

Figure 13: vLLM runs by overcommit, running limit s and clients c.

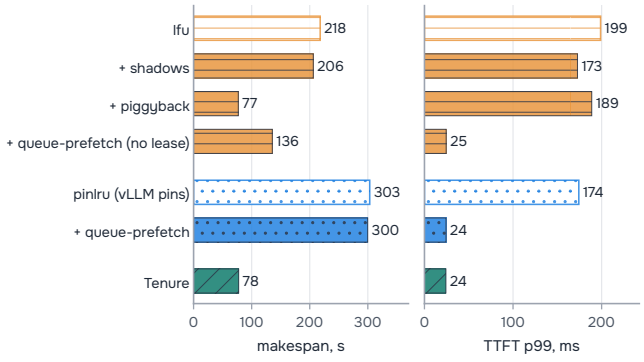

Figure 14: Ablation, geometric means over five workloads.

6.6 Overheads

Shadows take NVMe space equal to the trace's working set (53–108 GB), written once (60 GB on swissai, charged to makespan), and wear the drive at production write rates (§7). Queue-time prefetch reads 58–164 GB per trace from NVMe, 0.75–1.5 working sets because a block evicted between requests is fetched again, all off the latency path. Each admission updates the lease counts of the request's blocks and runs two linear scans over CXL-resident blocks (microseconds, §4.2); the state is 3 bytes per block, and no hardware counters are sampled.

Table 7: Batched serving: NVMe demand share (%) and ITL (ms) vs. batch depth.

		B=4		B=16		B=64	
workload	policy	NVMe%	ITL	NVMe%	ITL	NVMe%	ITL
swissai	TENURE	0	16	0	72	4	455
	hemem	48	87	64	330	70	1389
conv.	TENURE	0	63	2	281	35	4366
	hemem	76	501	78	1639	76	5656
uniform	TENURE	0	27	0	102	4	405
	hemem	55	122	55	436	56	1635

6.7 End-to-end evaluation

Setup. vLLM serves Llama-3.2-3B-Instruct [19] at FP16 on the RTX 3070 of Table 1, with its weights in GPU memory and vLLM's own GPU prefix cache off, so every prefix hit is loaded from the host store. Its KV heads have the per-layer shape of the 34B model of

§2 (8 heads of 128 dimensions) over 28 layers, so a 128-token block holds 14 MiB of KV. A prefix-reuse stream first stores 40 prompts of 2,048 tokens (640 blocks, 8.75 GiB). Each of the 240 measured requests then appends a unique 32-token tail to one of them, drawn uniformly with a fixed seed, and generates 4 tokens. The DRAM tier holds 2.5 GiB, a 3.5× overcommit inside the replay's range, and we also run 2×, 5× and 10×. GPU memory holds about one request's KV. vLLM runs at most $s \in \{1, 4\}$ requests at a time, and $c \in \{2, 4, 8\}$ concurrent clients set how many requests wait in its queue. Clients stream every request and record TTFT. Percentiles exclude the first 8 requests, which fill the empty DRAM tier; makespan covers all 240. Every policy sees the same request stream, once per configuration. All runs share one store, fully written before the first run.

Baselines. *Pins + LRU* is vLLM's own discipline (Table 3): its reference counts pin the blocks of running requests (block pinning, §2), its offloading tiers evict in LRU order by default [76], and LMCache offers the same block pinning through its pin and unpin calls [34]. A request's stored blocks are pinned in DRAM when the scheduler admits it, and unpinned blocks leave in LRU order. *Pins + prefetch* adds the lease's enqueue-time prefetch without the lease, as CachedAttention's look-ahead does [17], so a prefetched block may be evicted before its request runs. Both share the connector's store, shadow and piggyback with Tenure, so the three differ only in whether a queued request holds a claim.

Results. Under pins + LRU, 47–92% of requests read NVMe while they load, and under the lease none does (Figure 13(a)), because a queued request's blocks reach DRAM while its predecessors run (§4.1). At s=1, c=2 the median host-side fetch of a request's 16 blocks falls from 47.0 to 0.2 ms, while the copy into GPU memory takes 21 ms under both. The lease cuts median TTFT by 1.30–1.60×, p99 by 1.20–1.56× and makespan by 1.22–1.48× in every configuration (Figure 13(b, c)), and at s=1, c=4 its median stays at 339–345 ms from 2× to 10× overcommit. While DRAM has room for the queued requests' blocks, pins + prefetch comes within 1% of the lease's median, since the enqueue-time prefetch is what moves those blocks off NVMe, though 3–9% of its requests still find a prefetched block evicted. When the queued blocks fill DRAM (5× with 8 clients, 10× with 4), LRU evicts most prefetched blocks before their requests run, so 81–83% of its requests read NVMe and the lease is 1.46–1.56× faster at the median and 1.40× on makespan. Median TBT is 18.8 ms under pins and 17.5 ms under the lease, because vLLM decodes from GPU memory. The gain is on the TTFT path of §6.1; the decode-side churn tax of §3 belongs to KV-offload streaming, which the replay measures.

Speculative decoding. We reran the connector with vLLM's n-gram speculation, which drafts 4 tokens from the prompt, on prompts that repeat a 256-token pattern so that drafts find matches. Each of 40 requests generates 64 tokens over 64-token blocks, and every run starts from an empty store. Of 2,220 draft tokens, 291 were rejected. Under both the lease and pins + LRU, the generated text, the set of 320 blocks committed to the store and each block's checksum match the run without speculation, and no block identity was committed twice with different content. The blocks the connector stores are prompt blocks, and blocks sealed during decode follow the rule of §4.2, which vLLM enforces.

Scope. These runs serve Llama-3.2-3B on one 8 GB GPU from a DRAM and NVMe store, with prefill-heavy requests. vLLM runs one to four requests at a time, so the queue the lease protects is a few requests deep, which is one reason the makespan gain is smaller than that of Tenure over `pinlru` in the replay (Table 6). The leases fit in DRAM in every run, and no request is preempted.

7 Discussion and Limitations

Regime. The TBT and makespan wins are specific to KV-offload streaming, where context is read from the tiers at every step [65]; VRAM-resident decode absorbs per-token traffic, leaving TTFT as the tiering-sensitive path. The replay charges I/O at full price, so serving engines that overlap transfer with compute see smaller absolute gaps, though not better placement. In vLLM, which overlaps them, the lease still keeps NVMe off the TTFT path (§6.7). **Capacity, bandwidth and topology.** The lease needs only that a leased block sit in some tier faster than NVMe and that a queued context load within its predecessor's service time, so a deployment enters through capacity and delivered bandwidth. On our host CXL adds capacity but no bandwidth to the GPU (§2), because DRAM alone fills the 27 GB/s link that CXL reads also cross (Figure 2). A 24 MB block takes 1.33 ms from it against 0.89 ms from DRAM and 5.58 ms from NVMe (Table 1), and its capacity raises the batch depth at which every lease fits in fast memory. With our caps the leases exceed the 20 GB of fast tiers at B=64 on conversation (§6.4); without the 8 GB CXL tier they would exceed them at roughly half that depth. Access latency (277 ns) is under 0.03% of a block transfer, so a switch's extra hop does not matter, and interleaving devices adds bandwidth up to the GPU's link. Pooling a device across hosts divides its bandwidth among them, and a leased block helps only if it loads faster than from NVMe, so each host's share must stay above NVMe's 4.3 GB/s. Our device delivers about 18 GB/s to concurrent CPU readers on its socket. With 2 readers each gets about 9 GB/s and a block loads in 2.7 ms, under NVMe's 5.58 ms; with 4 the slowest gets 3.3 GB/s and needs 7.3 ms. A pool therefore scales with the number of devices, each backing about two hosts, not with the capacity of one device. Without CXL the lease still holds, with DRAM as its landing tier. In the no-CXL replay runs of §5, at 4–9× overcommit on swissai, toolagent, uniform and rag, Tenure serves 0.0% of demanded bytes from NVMe where `lru` serves 25–75%, and the vLLM study has no CXL tier at all. **Attention structure.** The lease is exact under full attention. Under importance- or window-based attention (H2O [92], Quest [71], IMPRESS [4]) a step reads a query-dependent subset, and the lease becomes a conservative minimum tier within which an estimate can still rank blocks. **Fetch vs. recompute.** Engines may recompute prefill rather than fetch cached KV [35]. Recomputing one 128-token block of the 34B model costs about 20 ms on an H100-class GPU ($2\times34\cdot10^9\times128$ FLOP at an achieved 400 TFLOP/s), against 0.89, 1.33 or 5.58 ms to load it from DRAM, CXL or NVMe, so even an accelerator twice as fast leaves NVMe below break-even. A faster accelerator also leaves less compute for a load to hide behind. A prefix hit is loaded, not recomputed, so its load overlaps only the compute of the request's new tokens. We modelled each prefill's TTFT as the later of its slowest hit-block load, measured in the no-CXL replay runs of §5, and the compute of its new blocks at Q ms each. At Q=20, placement sets 46% and 43% of `lru`'s mean TTFT on uniform and rag against 4% and 1% of Tenure's; at Q=10, a GPU twice as fast, 68% and 69% against 8% and

3%. The TTFT mechanism holds either way: VRAM-resident serving still loads context on a prefix hit, which queue-time leases move off the latency path. **Baseline fidelity.** The policies named after published systems, such as `memtis`, `hemem` and `tpp`, receive exact, free access information per block but synchronous epoch migration; MEMTIS and HeMem themselves migrate asynchronously. Each reimplements only that system's placement rule over KV blocks in our replay engine; the systems target other hierarchies (HeMem targets PM–DRAM, and NOMAD has no NVMe tier). The blindness finding is robust to this, since it concerns which blocks sit on NVMe when a queued request arrives; churn magnitudes are specific to our setup, which the injection experiment's within-setup design controls. Asynchronous migration moves the same bytes over the same PCIe link at a different moment, so it changes when decode pays the churn tax, not whether it pays it. Object granularity (24 MB) gives each estimate far more accesses per unit than kernels sample per page, so our results apply to object-level pool management, and to kernel policy only through what each policy observes. **Scale.** Capacities are scaled against the host's measured link speeds, so migration volume relative to capacity is inflated; finding directions are theory-consistent, magnitudes are regime-specific. On datacenter PCIe GPUs the VRAM-to-host ratio is more extreme than ours (H100: ~3350 vs. ~55 GB/s [50]), strengthening the placement finding; NVLink-C2C [49] and other cache-coherent heterogeneous hosts [85] and enterprise SSDs compress it. A production shadow stream also adds drive writes. And because new blocks keep arriving in production, the store must reuse the NVMe slots of blocks that leave the cache for good, and should first TRIM each freed slot (NVMe's Deallocate command), so the drive stops copying the slot's dead data during garbage collection.

8 Related Work

Kernel and CXL tiering infers hotness or access impact in software (TPP [39], MEMTIS [28], HeMem [60], Colloid [77], TMTS [14], NOMAD [80], SOAR [33], MTM [61], Chrono [58], DAMON [26, 54], weighted interleave [57]) or in the CXL device (NeoMem [97], M5 [70]). Cloud hosts expose pool memory to guests as CPU-less NUMA nodes (Pond [31]) or tier it beneath them in hardware (Memstrata [95]); other work rethinks tiering for data centers [87] and speeds up guests' nested address translation [48]. CXL pools serve as shared caches [38] and address spaces [84], VMs borrow remote memory [8], and in-kernel remote shared memory [46], OS-level memory middleware [47] and kernel-level KV sharing across a cluster [45] extend AI's memory past one host. None consumes a declared, counted, releasable reuse contract. **Application-informed memory** is the lineage we extend: DBMIN's counted, released locality sets [6]; TIP [56]; X-Mem's allocation-time tags [13]; AIFM's releasable dereference scopes [64]; cross-layer hints [75]; user-programmable kernel schedulers [43]. **KV-cache systems.** Besides the block pins of §2, also held by Dynamo's KV block manager [52], Mooncake Store grants a *lease* that blocks eviction for a few seconds while a lookup's transfer completes [41]. Agents' KV is retained between tool calls for a TTL in GPU memory [32] or with load-dependent expiry in host memory [5]. Queue-aware designs come closest. CachedAttention [17] prefetches queued sessions and evicts by look-ahead in its two-level store (an ablation variant, §6.2); PCR prefetches queued RAG requests' KV from SSD [79], PEEK shields the GPU-cached blocks of queued requests [81], and concurrent work reads KV from NVMe while requests wait in vLLM [24]. Unlike all of these, a Tenure lease spans enqueue to completion, counts its holders, expires, sets a minimum tier across four tiers, and orders spills by holder count. We compare it with vLLM's own block pinning (§6.7) because LMCache and Mooncake bring their own pools and transports. Resident-KV-Claims [67, 68] declares a releasable reuse contract for pool admission, without tier placement, enqueue-time start or cheap migration. Other systems estimate or schedule: HiCache tiers KV across GPU, host and storage [82]; a Bayesian reuse predictor drives six tiers in projection [16]; IMPRESS, KVFlow and TensorRT-LLM place KV by estimated importance, agent-step distance and TTL-expiring priority [4, 51, 53]; Strata optimizes transfers [83], Pensieve retains multi-turn KV [91], RAGCache places documents by cost [23], InfiniGen, ShadowKV and ArkVale fetch predicted subsets of offloaded KV [3, 29, 69], HCache restores KV from hidden states [18], FlexGen schedules offline [65], Kareto sizes tiers [94], and learned work targets eviction [42]. Other work evaluates CXL memory as a KV store [21, 72], compares KV-management methods [37] and characterizes production reuse [44, 78]. **Learned and searched policies**: LeCaR [74], LIRS [22], ARC [40], Kleio [12], Sibyl [66], IDT [2], ArtMem [89]; heuristics from automated search (FunSearch [62], Vulcan [15]). Concurrent simulations also find that the choice among estimators matters little, as 14 replacement policies barely beat LRU on prefix reuse [36] and tier capacity outweighs policy across GPU, CPU and SSD [10]. No prior study we know of compares these families with classical tiering on four-tier hardware under serving metrics. Weighted caching [1, 7, 90] anticipates the churn tax of §3.

9 Conclusion

Tiered-memory placement has been built on estimation because applications were assumed opaque. In LLM serving, attention re-reads every context block at each decode step, and the queue names each request's blocks before its first read, so the reuse a tiering policy estimates is already known. Tenure carries it into placement as residency leases, taken at enqueue and released with the request, and pairs them with zero-byte demotion and piggybacked promotion. It removes demanded NVMe reads in replay and keeps NVMe off the TTFT path in vLLM. Where an application already computes the reference pattern a memory manager estimates, the profitable interface is a counted, releasable declaration, backed by movement cheap enough to act on it.

References

[1] Pei Cao and Sandy Irani. 1997. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems (USITS 97)*. USENIX Association, Monterey, CA.

[2] Juneseo Chang, Wanju Doh, Yaebin Moon, Eojin Lee, and Jung Ho Ahn. 2024. IDT: Intelligent Data Placement for Multi-tiered Main Memory with Reinforcement Learning. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24)*. Association for Computing Machinery, New York, NY, USA, 69–82. doi:10.1145/3625549.3658659

[3] Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. 2024. ArkVale: Efficient Generative LLM Inference with Recallable Key-Value Eviction. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37. Curran Associates, Inc., Vancouver, Canada, 113134–113155. https://proceedings.neurips.cc/paper_files/paper/2024/hash/cd4b49379efac6e84186a3ffce108c37-Abstract-Conference.html

[4] 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)*. USENIX Association, Santa Clara, CA, 187–201. https://www.usenix.org/conference/fast25/presentation/chen-weijian-impress

[5] Minseo Choi and Ananya Joshi. 2026. Adaptive KV Retention for LLM Agents at Human-Approval Timescales. arXiv:2608.30830 [cs.OS]

[6] Hong-Tai Chou and David J. DeWitt. 1986. An evaluation of buffer management strategies for relational database systems. *Algorithmica* 1, 1-4 (1986), 311–336. doi:10.1007/bf01840450

[7] Marek Chrobak, Howard J. Karloff, T. H. Payne, and Sundar Vishwanathan. 1991. New Results on Server Problems. *SIAM Journal on Discrete Mathematics* 4, 2 (1991), 172–181. doi:10.1137/0404017

[8] Ho-Ren Chuang, Karim Manaouil, Tong Xing, Antonio Barbalace, Pierre Olivier, Balvansh Heerekar, and Binoy Ravindran. 2023. Aggregate VM: Why Reduce or Evict VM's Resources When You Can Borrow Them From Other Nodes?. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 469–487. doi:10.1145/3552326.3587452

[9] Jonathan Corbet. 2024. *Memory-Management: Tiered Memory, Huge Pages, and EROFS (MADV_DEMOTE/MADV_PROMOTE)*. Retrieved October 2026 from https://lwn.net/Articles/984839/

[10] Srikanta Datta Tumkur, Jay Iyer, Mehar Simhadri, Sai Pavan Kumar, Sai Kapil Kumar, and Ramesh Nampelly. 2026. Where Should the KV Cache Live? Placement Policies Across GPU, CPU, and SSD for Long-Lived Sessions. arXiv:2609.16215 [cs.AI]

[11] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL]

[12] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 37–48. doi:10.1145/3307681.3325398

[13] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/2901318.2901344

[14] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 727–741. doi:10.1145/3582016.3582031

[15] Rohit Dwivedula, Divyanshu Saxena, Sujay Yadalam, Eric Hayden Campbell, Daehyeok Kim, and Aditya Akella. 2025. Vulcan: Instance-specialized, Verifiable Systems Heuristics Through LLM-driven Search. arXiv:2512.25065 [cs.OS]

[16] Sanjeev Rao Ganjihal. 2026. Predictive Multi-Tier Memory Management for KV Cache in Large-Scale GPU Inference. arXiv:2604.26968 [cs.AR]

[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 *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 111–126. https://www.usenix.org/conference/atc24/presentation/gao-bin-cost

[18] Shiwei Gao, Youmin Chen, and Jiwu Shu. 2025. Fast State Restoration in LLM Serving with HCache. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 128–143. doi:10.1145/3689031.3696072

[19] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI]

[20] Dave Hansen and Ying Huang. 2021. *mm/migrate: Demote Pages during Reclaim, Linux 5.15, Commit 26aa2d199d6f (MADV_PAGEOUT Excluded via no_demotion)*. Retrieved October 2026 from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=26aa2d199d6f

[21] Kwan Huen, Gongjin Sun, Mehdi Nik, Caroline Kahn, Elham Movahedian, Senthil Murugesapandian, Michael Wasef, Shuyu Lyu, Dongyang Li, Edmund Au, Brian Luu, Jason Scott, Ranjitha Gopalakrishna, Namita Kumar, Amir Beygi, Vitorio Cargnini, and Ramdas Kachare. 2026. LLM KV Cache Storage Using CXL Memory. *IEEE Computer Architecture Letters* 25, 2 (2026), 319–322. doi:10.1109/lca.2026.3720952

[22] Song Jiang and Xiaodong Zhang. 2002. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*. Association for Computing Machinery, New York, NY, USA, 31–42. doi:10.1145/511334.511340

[23] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. arXiv:2404.12457 [cs.DC]

[24] Joseph Kanichai, Tiziano De Matteis, and Animesh Trivedi. 2026. Building py-kvcache: A Performance Characterization of External KV Caching for vLLM with NVMe SSDs. arXiv:2609.11744 [cs.DC]

[25] Abbas Khater, Seyed Amirmasoud Noohi, Massoud Reza Hashemi, and Zeinab Zali. 2024. An NFV-Based Framework for Autonomous Deployment of New Protocols in SDN Networks. *IEEE Access* 12 (2024), 148727–148740. doi:10.1109/access.2024.3476244

[26] Honggyu Kim. 2024. *DAMON-Based Tiered Memory Management for CXL Memory (DAMOS_MIGRATE_HOT/COLD), Linux 6.11*. Retrieved October 2026 from https://docs.kernel.org/mm/damon/design.html

[27] 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 '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[28] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 17–34. doi:10.1145/3600006.3613167

[29] 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)*. USENIX Association, Santa Clara, CA, 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[30] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, Honolulu, HI, 19274–19286. https://proceedings.mlr.press/v202/leviathan23a.html

[31] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[32] Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Huanchen Zhang, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. 2025. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. arXiv:2511.02230 [cs.OS]

[33] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 731–747. https://www.usenix.org/conference/osdi25/presentation/liu

[34] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. 2025. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. arXiv:2510.09665 [cs.LG]

[35] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 38–56. doi:10.1145/3651890.3672274

[36] Yiyu Liu, Minlan Yu, and Juncheng Yang. 2026. When Fancy Eviction Fails: Rethinking Cache Replacement For LLM Prefix Reuse. arXiv:2609.28870 [cs.DC]

[37] Oteo Mamo, Olga Kogiou, Hyunjin Yi, and Weikuan Yu. 2026. Comparative Characterization of KV Cache Management Strategies for LLM Inference. arXiv:2604.05012 [cs.AR]

[38] Karim Manaouil, Ji Zhang, Yang Zhe, Zhou Xing Wang, Shai Aviram Bergman, and Antonio Barbalace. 2025. Storage cluster for persistency, CXL pools for caching. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25)*. Association for Computing Machinery, New York, NY, USA, 22–29. doi:10.1145/3723851.3723859

[39] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[40] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. USENIX Association, San Francisco, CA, 115–130.

[41] Mooncake Project. 2026. *Mooncake Store Design: Leases, Soft Pins and Eviction*. Retrieved October 2026 from https://github.com/kvcache-ai/Mooncake/blob/main/docs/source/design/mooncake-store.md

[42] Luca Moschella, Laura Manduchi, and Ozan Sener. 2026. Learning to Evict from Key-Value Cache. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*. PMLR, Seoul, South Korea. arXiv:2602.10238 [cs.CL]

[43] Djob Mvondo, Antonio Barbalace, Jean-Pierre Lozi, and Gilles Muller. 2022. Towards User-Programmable Schedulers in the Operating System Kernel. In *11th Workshop on Systems for Post-Moore Architectures (SPMA 22)*. Rennes, France, 1–4. https://hal.science/hal-03750209

[44] William Nixon, Jon Durbin, Florian Standhartinger, Haryadi S. Gunawi, and Juncheng Yang. 2026. A Year in LLM Serving: Workload Evolution, Caching and Load-Balancing. arXiv:2608.13573 [cs.AI]

[45] Amir Noohi, Bita Asoodeh, and Antonio Barbalace. 2026. Towards a Solution to the Management Scaling Paradox in Distributed LLM Inference. In *Proceedings of the Sixth European Workshop on Machine Learning and Systems (EuroMLSys '26)*. Association for Computing Machinery, New York, NY, USA, 183–191. doi:10.1145/3805621.3807658

[46] Amir Noohi, Mostafa Derispour, and Antonio Barbalace. 2025. RMAI: Rethinking Memory for AI (Inference): In-Kernel Remote Shared Memory as a Software Alternative to CXL. In *Proceedings of the 5th Workshop on Machine Learning and Systems (EuroMLSys '25)*. Association for Computing Machinery, New York, NY, USA, 122–131. doi:10.1145/3721146.3721954

[47] Amir Noohi, Mostafa Derispour, Saba Ebrahimi, and Antonio Barbalace. 2026. NTSM: OS-Level Memory Middleware for Scaling AI Without Boundaries. In *Proceedings of the 27th ACM/IFIP International Middleware Conference (Middleware '26)*. Association for Computing Machinery, New York, NY, USA.

[48] Amir Noohi, Mostafa Derispour, Rodrigo C. O. Rocha, Antonio Barbalace, and Artemiy Margaritov. 2026. SagePTE: Co-locating Guest and Host PTEs to Accelerate Virtual Memory Translation. In *Proceedings of the 59th IEEE/ACM International Symposium on Microarchitecture (MICRO '26)*. IEEE, 1–16.

[49] NVIDIA. 2023. *NVIDIA Grace Hopper Superchip Architecture Whitepaper*. Retrieved October 2026 from https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/

[50] NVIDIA. 2023. *NVIDIA H100 Tensor Core GPU Datasheet*. Retrieved October 2026 from https://www.nvidia.com/en-us/data-center/h100/

[51] NVIDIA. 2025. *TensorRT-LLM KV-Cache Reuse: Token-Range Retention Priorities*. Retrieved October 2026 from https://nvidia.github.io/TensorRT-LLM/advanced/kv-cache-reuse.html

[52] NVIDIA. 2026. *Dynamo KV Block Manager (KVBM)*. Retrieved October 2026 from https://github.com/ai-dynamo/dynamo

[53] Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. 2025. KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows. In *Advances in Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., San Diego, CA. arXiv:2507.07400 [cs.DC]

[54] SeongJae Park. 2026. *DAMON: Data Access MONitor*. Retrieved October 2026 from https://docs.kernel.org/admin-guide/mm/damon/index.html

[55] 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) (ISCA '24)*. IEEE, Buenos Aires, Argentina, 118–132. doi:10.1109/isca59077.2024.00019

[56] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. 1995. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 79–95. doi:10.1145/224056.224064

[57] Gregory Price. 2024. *MPOL_WEIGHTED_INTERLEAVE, Linux 6.9*. Retrieved October 2026 from https://lwn.net/Articles/969379/

[58] Zhenlin Qi, Shengan Zheng, Ying Huang, Yifeng Hui, Bowen Zhang, Linpeng Huang, and Hong Mei. 2025. Chrono: Meticulous Hotness Measurement and Flexible Page Migration for Memory Tiering. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 835–853. doi:10.1145/3689031.3717462

[59] 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)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[60] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 392–407. doi:10.1145/3477132.3483550

[61] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 803–817. doi:10.1145/3627703.3650075

[62] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2024. Mathematical discoveries from program search with large language models. *Nature* 625, 7995 (2024), 468–475. doi:10.1038/s41586-023-06924-6

[63] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[64] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Virtual Event, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[65] 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 the 40th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, Honolulu, HI, 31094–31116. https://proceedings.mlr.press/v202/sheng23a.html

[66] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. 2022. Sibyl: Adaptive and Extensible Data Placement in Hybrid Storage Systems Using Online Reinforcement Learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 320–336. doi:10.1145/3470496.3527442

[67] Lukas Stepanek. 2026. Fail-Closed Lowering of Resident KV Claims onto LLM Serving Runtimes. arXiv:2606.01387 [cs.DC]

[68] Lukas Stepanek. 2026. Resident KV Claims: A Conformance Contract for Future Reuse under Active KV Pressure. arXiv:2605.24259 [cs.DC]

[69] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2025. ShadowKV: KV Cache in Shadows for High-Throughput Long-Context LLM Inference. In *Proceedings of the 42nd International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, Vancouver, Canada, 57355–57373. https://proceedings.mlr.press/v267/sun25b.html

[70] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 604–621. doi:10.1145/3676641.3711999

[71] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. Quest: Query-Aware Sparsity for Efficient Long-Context LLM Inference. In *Proceedings of the 41st International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, Vienna, Austria, 47901–47911. https://proceedings.mlr.press/v235/tang24l.html

[72] Yupeng Tang, Runxiang Cheng, Ping Zhou, Tongping Liu, Fei Liu, Wei Tang, Kyoungryun Bae, Jianjun Chen, Wu Xiang, and Rui Shi. 2024. Exploring CXL-based KV Cache Storage for LLM Serving. In *Machine Learning for Systems Workshop at NeurIPS 2024*. Vancouver, Canada. https://mlforsystems.org/assets/papers/neurips2024/paper17.pdf

[73] The Linux Kernel Documentation. 2026. *NUMA Balancing and Memory Tiering (numa_balancing=2, promotion rate limit)*. Retrieved October 2026 from https://docs.kernel.org/admin-guide/sysctl/kernel.html

[74] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/vietri

[75] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B. Gibbons, and Onur Mutlu. 2018. A Case for Richer Cross-Layer Abstractions: Bridging the Semantic Gap with Expressive Memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) (ISCA '18)*. IEEE, Los Angeles, CA, 207–220. doi:10.1109/isca.2018.00027

[76] vLLM Project. 2026. *KV-Cache Offloading: CPU and File-System Tiers with LRU or ARC Eviction (vllm/v1/kv_offload), v0.29*. Retrieved October 2026 from https:

//github.com/vllm-project/vllm

[77] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 79–94. doi:10.1145/3694715.3695968

[78] 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)*. USENIX Association, Boston, MA, 465–482. https://www.usenix.org/conference/atc25/presentation/wang-jiahao

[79] Wenfeng Wang, Xiaofeng Hou, Peng Tang, Hengyi Zhou, Jing Wang, Xinkai Wang, Chao Li, and Minyi Guo. 2026. PCR: A Prefetch-Enhanced Cache Reuse System for Low-Latency RAG Serving. arXiv:2603.23049 [cs.DC]

[80] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 19–35. https://www.usenix.org/conference/osdi24/presentation/xiang

[81] Bing Xie, Zhipeng Wang, Masahiro Tanaka, and Zhen Zheng. 2026. PEEK: Predictive Queue-Informed KV Cache Management for LLM Serving. arXiv:2607.02525 [cs.DC]

[82] Zhiqiang Xie. 2025. *SGLang HiCache: Fast Hierarchical KV Caching with Your Favorite Storage Backends*. Retrieved October 2026 from https://lmsys.org/blog/2025-09-10-sglang-hicache/

[83] Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. 2026. Strata: Hierarchical Context Caching for Long Context Language Model Serving. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 1–16. https://www.usenix.org/conference/osdi26/presentation/xie-zhiqiang

[84] Tong Xing and Antonio Barbalace. 2025. Rethinking Applications' Address Space with CXL Shared Memory Pools. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25)*. Association for Computing Machinery, New York, NY, USA, 52–59. doi:10.1145/3723851.3723858

[85] Tong Xing, Cong Xiong, Tianrui Wei, April Sanchez, Binoy Ravindran, Jonathan Balkind, and Antonio Barbalace. 2025. Stramash: A Fused-Kernel Operating System For Cache-Coherent, Heterogeneous-ISA Platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1172–1188. doi:10.1145/3676641.3716275

[86] Tong Xing, Cong Xiong, Chuan Ye, Qi Wei, Javier Picorel, and Antonio Barbalace. 2023. Maximizing VMs' IO Performance on Overcommitted CPUs with Fairness. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 93–108. doi:10.1145/3620678.3624649

[87] Tong Xing, Jiaxun Yang, Javier Picorel, and Antonio Barbalace. 2025. Rethinking Tiered Memory Management in Cloud Data Centers. In *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC '25)*. Association for Computing Machinery, New York, NY, USA, 74–87. doi:10.1145/3772052.3772213

[88] Xiaozhe Yao, Youhe Jiang, Ilia Badanin, Qinghao Hu, Robert Matthew Smith, Binhang Yuan, Imanol Schlag, Eiko Yoneki, and Ana Klimovic. 2026. OpenTela: Unifying Decentralized Computing Resources for Heterogeneous LLM Serving (Operational Systems). In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 1821–1838. SwissAI serving trace: https://huggingface.co/datasets/eth-easl/swissai-serving-trace. https://www.usenix.org/conference/osdi26/presentation/yao

[89] Xinyue Yi, Hongchao Du, Yu Wang, Jie Zhang, Qiao Li, and Chun Jason Xue. 2025. ArtMem: Adaptive Migration in Reinforcement Learning-Enabled Tiered Memory. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 405–418. doi:10.1145/3695053.3731001

[90] Neal E. Young. 1994. The k-Server Dual and Loose Competitiveness for Paging. *Algorithmica* 11, 6 (1994), 525–541. doi:10.1007/bf01189992

[91] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. Stateful Large Language Model Serving with Pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 144–158. doi:10.1145/3689031.3696086

[92] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 36. Curran Associates, Inc., New Orleans, LA, 34661–34710. https://proceedings.neurips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfcebb2827f8-Abstract-Conference.html

[93] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H. Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37. Curran Associates, Inc., Vancouver, Canada, 62557–62583. https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html

[94] Xianzhe Zheng, Zhengheng Wang, Ruiyan Ma, Rui Wang, Xiyu Wang, Rui Chen, Peng Zhang, Sicheng Pan, Zhangheng Huang, Chenxin Wu, Yi Zhang, Bo Cai, Kan Liu, Teng Ma, Yin Du, Dong Deng, Sai Wu, Guoyun Zhu, Wei Zhang, and Feifei Li. 2026. Adaptive Multi-Objective Tiered Storage Configuration for KV Cache in LLM Service. arXiv:2603.08739 [cs.AR]

[95] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

[96] 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)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[97] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. 2024. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) (MICRO '24)*. IEEE, Austin, TX, 1518–1531. doi:10.1109/micro61859.2024.00111