Towards a Solution to the Management Scaling Paradox in Distributed LLM Inference

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

Bita Asoodeh
University of Edinburgh
United Kingdom
bita.asoodeh@ed.ac.uk

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

Abstract

Disaggregated LLM inference separates prefill and decode across nodes, relying on cross-node KV cache reuse (prefix caching) to reduce time-to-first-token (TTFT). Yet existing systems manage this cache entirely in user space, atop the Operating Systems (OS) kernel that already handles memory mapping, page tables, and RDMA registration. We identify a *management scaling paradox*: user-space overhead from coordination RPCs, per-chunk locking, redundant memory copies, and RDMA transfer fragmentation grows to 79% of achievable TTFT as the cache warms, nearly doubling latency when caching should help most. We formalize this paradox as a TTFT decomposition model and validate it on the state-of-the-art LMCache + Mooncake stack atop vLLM. We then present **RMC** (Remote Memory for Cache), a kernel-space framework that extends OS-provided shared memory (`/dev/shm`) to cluster-wide scope via content-addressed Virtual Memory Regions (VMRs), where any node computes the address of any cached chunk by hashing its token content, requiring no coordination. Across four workloads, RMC achieves 1.1–1.3× TTFT reduction over LMCache + Mooncake and up to 2.1× over full recompute.

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **Computing methodologies → Distributed artificial intelligence**; • **Networks → Programming interfaces**.

Keywords: KV cache management, LLM serving, Disaggregated inference, Operating systems, RDMA, Prefix caching, Distributed shared memory

ACM Reference Format:
Amir Noohi, Bita Asoodeh, and Antonio Barbalace. 2026. Towards a Solution to the Management Scaling Paradox in Distributed LLM Inference. In *Sixth European Workshop on Machine Learning and Systems (EuroMLSys '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3805621.3807658

EuroMLSys '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2605-7/2026/04
https://doi.org/10.1145/3805621.3807658

1 Introduction

In autoregressive LLM serving, each token's Key and Value (KV) projections are cached so that previously computed attention states can be reused across decoding steps [5, 25]. **Prefix caching**, reusing KV cache from tokens shared across requests, reduces redundant computation, with production workloads exhibiting 45–71% cache hit rates [5, 19, 22, 31]. In **disaggregated serving** [20, 22, 33], where prefill and decode run on separate nodes, cached KV data must be transferred across the network, making cache management efficiency a direct determinant of time-to-first-token (TTFT).

Existing systems manage KV cache entirely in user space. Our instrumented profiling of the LMCache + Mooncake stack [5, 22], the state-of-the-art for cross-node KV cache reuse atop vLLM [12], reveals that user-space overhead from coordination RPCs, per-chunk locking, redundant memory copies, serialized hashing under Python's Global Interpreter Lock (GIL), and RDMA transfer fragmentation grows to 79% of achievable TTFT as the cache warms, nearly doubling latency when caching should help most (§3). The root cause is redundant software layers: user-space stacks replicate memory allocation, eviction, and transfer registration that the Operating System (OS) kernel already performs; they cannot bypass the kernel, so they add coordination on top of it. In disaggregated serving, this coordination must span multiple nodes: user-space stacks maintain distributed hash tables, metadata RPCs, and per-node buffer management to track and transfer cached KV chunks across the network.

This raises a natural question: if the OS kernel already manages memory, page tables, and RDMA registration on every node, can it provide the distributed caching layer directly? We observe two KV cache properties that make this feasible: (1) per-token cache occupation is *fixed* by model architecture, so memory regions have a known size at deployment time; and (2) cache index is *determined by token content*, meaning the same token sequence always produces identical KV tensors, so a deterministic hash of tokens can serve as both the cache index and the location of that chunk across the cluster. Together, these eliminate the need for metadata servers, distributed hash tables, or coordination RPCs: any node can independently locate any cached chunk by hashing its tokens. The kernel already provides the necessary primitives (`mmap`, page-fault handling, and RDMA memory registration), and vLLM

already uses `/dev/shm` for local KV cache sharing; the natural extension is cluster-wide. We introduce the idea of **RMC** (**R**emote **M**emory for **C**ache), which extends the Partitioned Global Address Space (PGAS) model with content-addressed Virtual Memory Regions (VMRs), memory regions whose address and location are derived from the token content they cache, and validate it with a prototype. Our contributions:

- Instrumented profiling of LMCache + Mooncake quantifying user-space management overhead and the management scaling paradox (§3).
- RMC: content-addressed PGAS with zero-coordination chunk addressing, write-once semantics that eliminate locking, and single-operation RDMA transfers (§4).
- Evaluation across four representative workloads showing 1.1–1.3× TTFT reduction over LMCache + Mooncake and up to 2.1× over full recompute (§5).

2 Background

2.1 KV Cache in Disaggregated LLM Serving

During inference, LLMs store Key and Value projections from prior tokens in a *KV cache* to avoid recomputation during generation [12, 25]. The cache size per token is fixed by model architecture [5, 12]; for Llama-3.1-8B [1] (32 layers, 8 KV heads, 128-dim, BF16) this yields 128 KB/token, or 32 MB per 256-token chunk.

LLM inference has two phases [30, 33]: **prefill** processes the entire prompt in one pass, producing the KV cache (compute-bound), and **decode** generates tokens one at a time (memory-bandwidth-bound [20]). The key metric is **time-to-first-token (TTFT)**, the delay until the first output token [22]. **Disaggregated serving** [20, 33] separates prefill and decode onto specialized nodes, but the entire KV cache must now reach the decode node before generation begins (§3).

2.2 Prefix Caching

Production workloads exhibit 45–71% prefix caching ratios [5, 19, 22, 31] from shared system prompts, few-shot examples, and RAG contexts. **Prefix caching** stores and reuses KV data for previously seen token sequences in a prompt, eliminating redundant prefill and reducing TTFT. vLLM's PagedAttention [12] and SGLang's RadixAttention [32] implement local prefix caching at 256-token chunk boundaries, where each chunk is identified by a hash of the cumulative token sequence. Since vLLM is a multi-process Python application, the prefix cache hash table resides in the scheduler process [12], serializing all cache lookups under Python's GIL.

Single-node caching is inherently limited: if the KV cache is evicted under memory pressure, or the request is routed to a different node, the prefix must be recomputed from scratch or retrieved from a slower storage tier (e.g., disk).

2.3 State-of-the-Art

Extending prefix sharing across nodes in a disaggregated cluster requires three capabilities: (1) *cache lookup and management*, mapping token sequences to cached KV chunks and deciding what to cache, where, and when to evict; (2) *cross-node location tracking*, maintaining a cluster-wide directory so any node can locate cached prefixes; and (3) *high-throughput data transfer*, moving multi-megabyte KV chunks between nodes with minimal latency.

No single system provides all three. vLLM [12] and SGLang [32] handle only local caching; CacheGen [14] and CacheBlend [29] reduce prefill cost but provide no cross-node lookup. The most complete cross-node solution today combines:

- **LMCache** [5] provides cache identification and multi-tier management. Each 256-token chunk is identified by a cryptographic hash (`sha256_cbor`) and can be stored across GPU, CPU DRAM, local disk, or remote storage tiers. LMCache integrates directly into vLLM's KV connector interface, handling prefix matching, chunk-level eviction, and tier promotion.
- **Mooncake** [22] provides location tracking and RDMA-based transfer. Its centralized metadata service (Conductor) maintains a cluster-wide directory of cached prefixes, and its transfer engine (NIXL) manages RDMA buffer registration, connection pooling, and data movement between nodes.

Together, layered atop vLLM's disaggregated prefill-decode coordination, LMCache and Mooncake form the state-of-the-art stack for cross-node KV cache reuse, deployed in production by Moonshot AI for their Kimi service [22]. Critically, all existing systems (vLLM, SGLang, LMCache, Mooncake, CacheGen, CacheBlend) operate entirely in user space with explicit coordination for cross-node sharing. As we show in §3, this architectural choice introduces management overhead that grows with cache hit rate, precisely when caching should deliver the greatest benefit.

3 Motivation

In disaggregated LLM serving, a KV cache miss forces full prefill computation, while a cache hit only requires an RDMA transfer, **22× cheaper** on our target hardware (§5). Yet our profiling reveals that the state-of-the-art cross-node caching stack (§2) introduces coordination overheads that largely overlap the savings.

Figure 1 shows TTFT for 20 requests on a disaggregated prefill-decode cluster (Llama-3.1-8B-Instruct [1], two RTX 4090 nodes, 200 Gbps RDMA), progressively warming the cache from 0% to 80% hit rate. We compare three regimes. *Full recompute* (no caching, colocated) pays full prefill every request (~853 ms). *LMCache + Mooncake* performs disaggregated caching with the full user-space stack (hashing, RPCs, PCIe staging, RDMA transfers). *Computation + communication only* is derived by instrumenting the caching stack to isolate GPU prefill time for uncached chunks plus RDMA wire transfer time for cached chunks, stripping all user-space management (hashing, RPCs, staging, locking). The latter

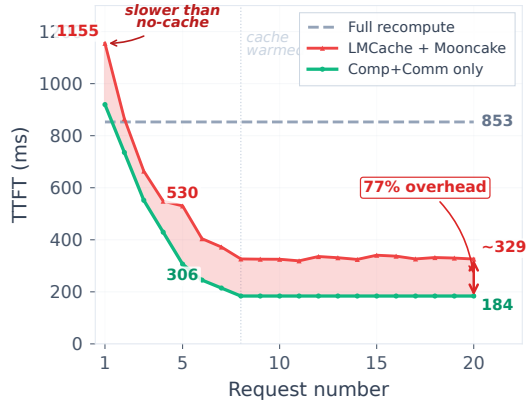

Figure 1. Progressive TTFT (Llama-3.1-8B, 7,680 tokens, 80% hit, 200 Gbps RDMA). Shaded gap: user-space management overhead, growing to 79% of achievable TTFT.

drops to ~184 ms once warm, a 4.6× reduction. However, LMCache + Mooncake adds ~145 ms of management overhead on average, growing from 26% of computation + communication TTFT at request 1 to **79%** once converged, capping the ideal 4.6× speedup at ~2.6×.

3.1 The Management Scaling Paradox

In existing systems, the measured TTFT for a request with C fixed-size chunks at cache hit rate f decomposes as:

$$T_{\text{ttft}} = \underbrace{(1-f)Ct_{\text{pfill}}}_{\text{prefill}} + \underbrace{Ct_{\text{stage}}}_{\text{PCIe staging}} + \underbrace{fCt_{\text{mgmt}}}_{\text{software overhead}} \quad (1)$$

This models exclusively the per-request prefill-side latency before the first token (TTFT). Here t_{pfill} is per-chunk GPU prefill compute cost, t_{stage} is per-chunk GPU→CPU memory copy cost (in user-space stacks such as LMCache, KV data is staged through CPU-side buffers before RDMA transfer, incurred for every chunk whether cached or not), and t_{mgmt} is per-chunk software overhead (hashing, locking, coordination RPCs, and network round-trips to the metadata service) incurred for cached chunks; freshly computed chunks also pay management cost during the store path, but this is subsumed by t_{pfill} which dominates. As $f \to 1$, the prefill term vanishes while the staging cost remains constant and the management term grows with the number of cached chunks: every cached chunk must be staged through CPU memory, acquire locks under the GIL, synchronize with the coordinator, and pass through user-space memory management. Note that Eq. 1 is purposely a simplified model. For example, the behavior at the very first request (f=0, cold start) is not accurately modeled: disaggregated caching is slower than colocated recompute because the staging and coordination costs are incurred with zero cache benefit, as confirmed in our evaluation (§5). We define the **management scaling paradox**: *in any disaggregated caching system where per-object management cost is non-zero, the fraction of latency consumed by management grows monotonically with cache hit rate, because useful computation shrinks while management overhead grows.* Eq. 1 provides a general diagnostic: when $t_{\text{mgmt}} > t_{\text{pfill}}$, additional caching degrades rather than improves effective throughput.

Our measurements confirm this: at convergence (80% hit), the computation + communication baseline drops to ~184 ms, yet LMCache + Mooncake measures ~329 ms. The ~145 ms gap is pure management overhead, amounting to 77–79% of the achievable TTFT depending on the request. The ideal 4.6× speedup from caching is capped at ~2.6×.

3.2 Root Cause: Redundant User-Space Layering

We found that all these overheads stem from treating KV cache management exclusively as a *user-space application problem*: hash tables in Python dictionaries, metadata in RPC servers, memory in user-space allocators with per-object locking. This is not unique to LMCache and Mooncake: KVDirect [28] found that actual data transfer accounts for only 13.2% of total KV cache transfer duration, with 87% consumed by synchronization, and FlowKV [13] reports that for example discontinuous allocation forces redundant operations dominating transfer time. Beyond overhead, user-space scheduling introduces stale snapshots: cache availability is checked once at dispatch and frozen, so concurrent requests redundantly recompute shared chunks, a thundering-herd pattern that grows with parallelism.

The OS kernel is already involved. These user-space layers build the distributed KV caching layer atop the OS kernel. Every RDMA transfer requires kernel-registered memory regions; every allocation goes through the kernel page allocator; every CPU-side buffer is backed by kernel-managed virtual pages. Bypassing the OS kernel is not the answer: kernel bypass (e.g., DPDK [6], SPDK [24]) eliminates the kernel from the *network data path*, but in a dynamic multi-user system the *memory control path* must pass via the OS kernel, which cannot be bypassed. Because user-space stacks redundantly replicate *memory management* that the kernel already performs, the question is whether the kernel's existing involvement can be made direct, providing the distributed caching layer itself.

From local shared memory to cluster-wide. The answer is hinted by how KV cache is already shared *within* a single node. vLLM uses `/dev/shm`, the kernel's POSIX shared memory facility, to share KV cache between processes on the same machine [12]. The kernel manages this through page tables: no user-space RPC, no explicit serialization, and no per-object locking. The natural extension is to add RDMA-backed remote page faults, turning node-local shared memory into a cluster-wide in-kernel Partitioned Global Address Space (PGAS), a programming model where a single virtual address space is partitioned across nodes so that any node can access any region through a uniform address [21].

What is needed to be extended. Prior RDMA-based remote memory systems (Infiniswap [9], Leap [15], AIFM [23], FaRM [7]) provide remote paging at 4 KB–2 MB granularity, far below the 32 MB that matches a KV cache chunk, and none support

content-addressed addressing, where the memory address and location of an object are derived solely from a hash of its content, analogous to how distributed hash tables (e.g., Chord) resolve object location from content keys without a central directory. RMAI [16] demonstrated coarse-grained VMRs via an in-kernel PGAS for MoE expert loading, but uses model-architecture-based addressing and cannot identify chunks by token content. Two properties of KV cache make content-addressed kernel-level caching feasible:

1. **Fixed occupancy per token.** Cache occupancy per token is determined entirely by model architecture (§2), so memory region sizes are fixed at deployment time and no dynamic allocation or sizing is needed.
2. **Content-determined cache index.** The same token sequence always produces identical KV tensors, an observation exploited by vLLM [12] and SGLang [32] for local prefix caching. A deterministic hash of token content uniquely identifies each chunk without requiring a metadata server.

Together, these properties mean that *any node can independently compute the location of any cached KV chunk*, with no coordination server, no distributed hash table, and no network round-trips. With cache addressing in the kernel, where RDMA registration, page-fault handling, and memory mapping are native primitives, the PCIe staging, software coordination overhead, and stale-snapshot hazard are all eliminated: chunks become visible cluster-wide as soon as they are written, and an in-kernel fault handler resolves remote accesses without crossing the user–kernel boundary.

4 Design

RMC provides the cluster-wide distributed cache argued for in §3: a Partitioned Global Address Space (PGAS) where each 256-token KV cache chunk maps to a fixed-size Virtual Memory Region (VMR) at a deterministic address computed from token content. All nodes (coordinator, prefill, and decode) map the same PGAS region via kernel-level `mmap`, so identical virtual addresses resolve to the same logical VMR slot on every node. Figure 2 shows the end-to-end architecture.

4.1 Content-Addressed PGAS

The central idea of RMC is that the virtual address of any VMR is a deterministic function of token content:

$$\texttt{vmr\_addr} = \texttt{BASE} + \left(\text{xxHash64}(\text{tok}[0{:}b]) \bmod N_s\right) \times S_{\text{vmr}} \tag{2}$$

where b is the chunk boundary (a multiple of 256), N_s is the total number of VMR slots, and S_{vmr} is the VMR size (32 MB for Llama-3.1-8B). All parameters are fixed at model deployment time and identical across all nodes. Computing a VMR address requires a single xxHash64 evaluation [4] on 1 KB of token data plus a modulo and an offset addition (the BASE address), which is constant-time per chunk regardless of cluster size, with no coordination server or distributed hash table.

Correctness follows from determinism (identical tokens produce bit-identical KV tensors, so no false matches) and prefix dependency (the KV cache at boundary b depends on all tokens $[0{:}b]$, so no missed matches). Hash collisions are negligible ($< 10^{-9}$ at 100K active chunks with 64-bit hashes) and, when they occur, fall back to prefill, producing correct results, never stale data.

4.2 Chunk-Level VMRs

Each VMR is a contiguous memory region holding the KV cache for one 256-token chunk across all transformer layers (32 MB for Llama-3.1-8B). Because all VMRs for a given model are identically sized, allocation is fragmentation-free: slots are never split, merged, or partially occupied.

The contiguous layout enables single-operation RDMA transfer of an entire chunk without staging buffers, scatter-gather lists, or per-layer iteration, in contrast to the fragmented transfers used by existing stacks (§5.2).

Write path and coherence. After computing a chunk's KV cache content, the prefill node writes the data into the corresponding VMR via `mmap()` and broadcasts its owner node ID to the corresponding entry in the block directory (the shared lookup structure described in §4.3) on every other node using one-sided RDMA writes. After the broadcast, all nodes know which node owns the chunk and can issue RDMA reads to fetch it. VMRs follow write-once semantics: once computed, a chunk is immutable until evicted. Because each block-directory entry is a single 8-byte owner ID and aligned stores are atomic on x86-64, no locking is required. If two nodes independently compute the same chunk, determinism guarantees bit-identical results, making concurrent writes safe without application-level locks. On eviction, the kernel broadcasts a zero owner ID to all remote block directories, invalidating the entry cluster-wide.

4.3 Prefix Matching and Scheduling

Given a request with T tokens, RMC finds the longest cached prefix and the best target node in a single pass. For each of the $C = \lfloor T/256 \rfloor$ chunks, it computes the VMR address via Eq. 2 and reads the corresponding entry in a shared *block directory*, a compact array in shared memory accessible from both user space and the kernel, where each 8-byte entry holds the owner node ID (zero if unoccupied). For example, 1M VMR slots require only 8 MB of directory space, fitting comfortably in shared memory on each node. The longest cached prefix is the last chunk with a non-zero owner node ID; the target node is $\text{argmax}_n \text{hits}(n)$.

Each lookup requires a single hash and one local directory read. Because the block directory resides in shared memory, the coordinator and vLLM's scheduler read it directly without a metadata service or RPCs. Ownership is propagated by one-sided RDMA writes, so consistency is eventual: a stale-zero entry causes unnecessary recomputation and a stale-owner

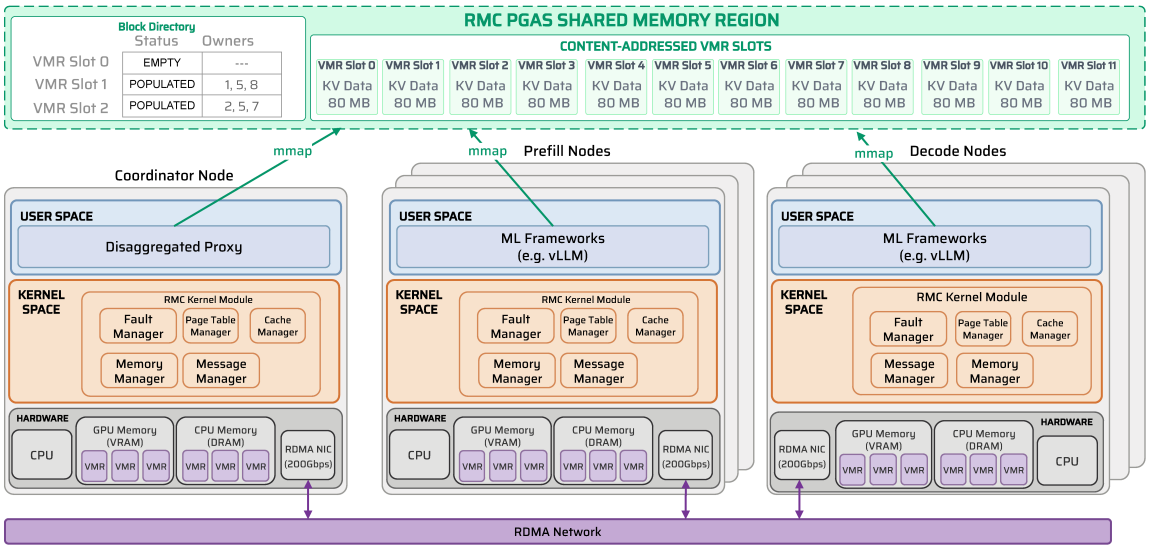

Figure 2. RMC architecture.

triggers fallback to prefill; neither produces incorrect results, since recomputed KV data is bit-identical to evicted data.

4.4 OS Kernel-Level Operation

As argued in §3, extending `/dev/shm`-style shared memory to cluster scope requires OS kernel involvement. RMC registers the entire PGAS region with the RDMA NIC once at deployment; subsequent VMR transfers use one-sided operations without user-kernel transitions or intermediate buffer copies.

VMR-granularity faulting. Standard remote memory systems [9, 15] fault at 4 KB granularity, so fetching a 32 MB chunk triggers ~8,000 individual faults. RMC handles the entire chunk in a single fault (one trap, one RDMA operation, one TLB batch update), eliminating three orders of magnitude of per-page overhead. At system level, coarse-grained VMR faulting reduces total page faults by up to 80% and TLB shootdowns by 4× [16].

Transparent access. User-space serving systems access KV cache through memory-mapped VMR regions. A page fault on an unmapped VMR triggers the kernel fault handler, which reads the owner node ID from the block-directory entry for that VMR, issues a one-sided RDMA read to the owner, populates local pages, and resumes the faulting process transparently. From vLLM's perspective, cached KV data appears in CPU-pinned memory accessible to the GPU on first access, requiring no modification to the serving system.

Implementation. We implement RMC as a Linux kernel module building on the in-kernel PGAS infrastructure of RMAI [16], adding ~2,000 lines of C. While RMAI provides the base PGAS and VMR faulting mechanisms, RMC contributes content-addressed addressing (Eq. 2), write-once coherence without locking (§4.2), and the block-directory scheduling design (§4.3). VMRs use lazy page allocation, so only populated VMRs consume physical memory; the OS kernel maintains an LRU list across all slots and reclaims pages under memory pressure; evicted entries are invalidated cluster-wide via one-sided RDMA writes that zero the corresponding block-directory entries. This eliminates application-level eviction logic. We integrate with vLLM [12] via a custom `CacheEngine` that replaces the default PagedAttention KV cache backend with memory-mapped access to the PGAS region (~200 lines of Python).

5 Evaluation

We evaluate RMC against the combined LMCache + Mooncake stack [5, 22], the state-of-the-art for cross-node KV cache reuse atop vLLM's disaggregated serving. We focus on time-to-first-token (TTFT), the primary SLO metric for interactive LLM applications [20, 22, 33], as it directly determines perceived responsiveness and is dominated by prefill and cache transfer latency in disaggregated deployments.

Setup. Disaggregated prefill-decode cluster with NVIDIA RTX 4090 GPUs (24 GB), Intel i7-9700K CPUs, 32 GB DDR4, and BlueField-2 200 Gbps NICs, running Llama-3.1-8B-Instruct [1] (32 layers, 8 KV heads, 128-dim, BF16; 32 MB/chunk) on vLLM 0.15.1 with LMCache 0.3.14 and Mooncake 0.3.9. We evaluate four workloads modelling representative production patterns [22, 31], each with progressive cache warm-up (token counts are exact multiples of the 256-token chunk boundary):

- **W1** – Multi-turn Chat (2,048 tokens, 8 chunks, 50% hit)
- **W2** – RAG Retrieval (4,096 tokens, 16 chunks, 44% hit)
- **W3** – Code/Agent (6,144 tokens, 24 chunks, 54% hit)
- **W4** – Long Context (7,680 tokens, 30 chunks, 63% hit)

5.1 End-to-End TTFT

Table 1 reports converged TTFT across four regimes: full recompute (no caching), LMCache + Mooncake, RMC, and

Table 1. Converged TTFT (ms) at 200 Gbps. Speedups are RMC over baseline.

Workload	Full Recomp.	LMC+ MC	RMC	Comp+ Comm	vs LMC	vs Rec.
W1 (2,048, 50%)	231	174	**156**	152	1.1×	1.5×
W2 (4,096, 44%)	455	378	**319**	305	1.2×	1.4×
W3 (6,144, 54%)	691	488	**390**	366	1.3×	1.8×
W4 (7,680, 63%)	853	530	**401**	366	1.3×	2.1×

computation + communication only. The computation + communication lower bound is obtained by instrumenting the user-space LMCache + Mooncake code path with fine-grained timestamps to isolate only GPU prefill time (for uncached chunks) and RDMA wire transfer time (for cached chunks), excluding all management overhead. It is not a kernel-level system or a separate implementation, but a measured decomposition of the existing user-space stack that reveals the inherent cost floor. Figures 3–6 show how TTFT evolves per-request as the cache progressively warms.

Cold-start overhead. LMCache + Mooncake is *slower* than full recompute for the first several requests because the full recompute baseline runs colocated (single GPU, no network transfer), while disaggregated caching must pay staging, coordination, and transfer costs even at zero hit rate. Crossover occurs around request 3–4; RMC's lower overhead shifts it earlier.

Figures 3–6 show per-request convergence. Management overhead scales with cached chunk count: from 22 ms (W1, 4 chunks) to 164 ms (W4, 19 chunks), representing 14–45% of achievable TTFT. RMC's advantage widens from 1.1× (W1) to 1.3× (W3–W4) because it eliminates per-chunk coordination, hashing, and staging. At every workload, RMC converges to within 4–35 ms of computation + communication, confirming the management scaling paradox is a structural property of user-space cache management. Because per-chunk management cost (t_{mgmt} in Eq. 1) is independent of cluster size, the paradox predicts that RMC's advantage grows further with longer contexts and larger clusters, where more cached chunks amplify the overhead RMC eliminates. Our motivation experiment (Figure 1) demonstrates this at 80% converged hit rate; at full convergence the paradox is most acute, as the prefill term in Eq. 1 approaches zero and management overhead dominates entirely.

5.2 Overhead Source Analysis

We instrument both LMCache and Mooncake code paths to isolate the sources of the overhead gap.

Mooncake transfer engine (dominant). Each cross-node transfer traverses CPU-side memory stages on both the sender and receiver before and after reaching the wire. Our instrumentation measures 2 HTTP + 1 ZMQ coordination round-trips per transfer (100μs–5 ms). Mooncake further fragments each region into 64 KB slices, so a 32 MB chunk becomes 512 RDMA operations backed by 4 KB kernel pages, each with its own CQE, underutilizing NIC bandwidth [22, 28]. Every

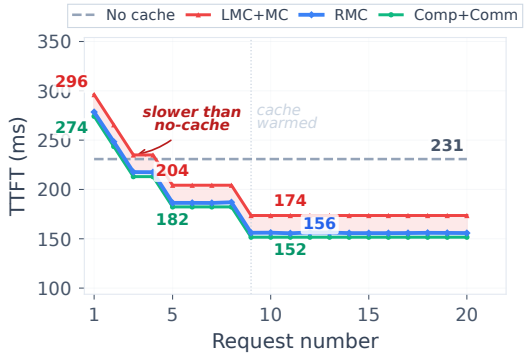

Figure 3. W1: Multi-turn Chat (2,048 tokens, 50% hit).

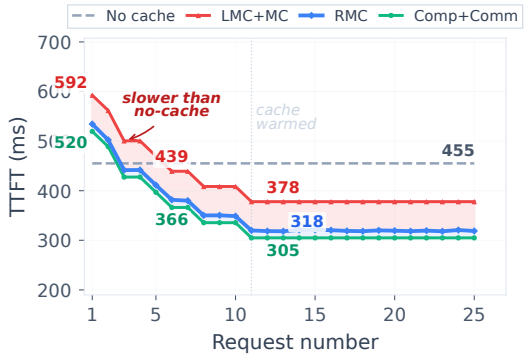

Figure 4. W2: RAG Retrieval (4,096 tokens, 44% hit).

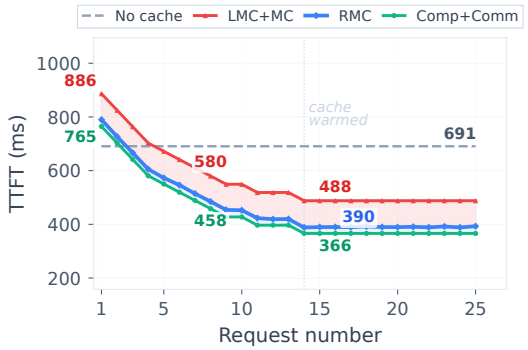

Figure 5. W3: Code/Agent (6,144 tokens, 54% hit).

cached chunk traverses this pipeline, so costs grow linearly with cached chunk count.

LMCache management layer (minor). LMCache [5] adds `sha256_cbor` hashing under the GIL (3×/chunk), 4N mutex locks per retrieve, and per-chunk `cudaStreamSync`, totaling ~4 ms for W4 (under 3% of the gap).

Table 2 quantifies the scaling: overhead grows from 22 ms (W1, 4 cached chunks) to 164 ms (W4, 19 cached chunks), confirming the management scaling paradox from §3. The near-linear scaling (~8.6 ms/chunk) confirms that cost is dominated by per-chunk operations (hashing, locking, RDMA

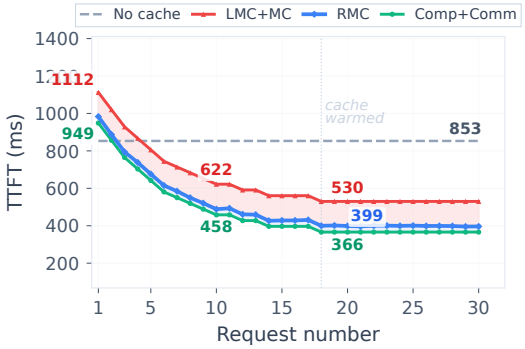

Figure 6. W4: Long Context (7,680 tokens, 63% hit).

Table 2. Management overhead across workloads. Overhead grows with cached chunk count, consuming 14–45% of achievable TTFT.

Workload	Hit%	Cached	Comp+Comm	Overhead	OH%
W1 (2,048, Chat)	50%	4	152 ms	22 ms	14%
W2 (4,096, RAG)	44%	7	305 ms	73 ms	24%
W3 (6,144, Code)	54%	13	366 ms	122 ms	33%
W4 (7,680, Long)	63%	19	366 ms	164 ms	45%

fragmentation), precisely what in-kernel single-operation transfers eliminate.

6 Discussion

Is the Kernel Fundamentally Necessary? A natural question is whether a well-engineered user-space system could eliminate most of the overheads without kernel modifications. We argue it cannot, for structural reasons. Even switching from Python to a hypothetical C/C++ implementation that avoids the GIL still must (1) cross the user–kernel boundary for every RDMA memory registration and page pinning, (2) maintain address-mapping data structures that duplicate the kernel's own page tables, and (3) use either `userfaultfd` (incurring a context switch per fault) or busy-polling (wasting CPU cycles) to detect remote accesses. An in-kernel handler resolves the fault, issues the RDMA read, and maps the page in a single trap without leaving kernel mode, demonstrated by several works [2, 3, 26, 27]. More fundamentally, the kernel is already in the path for every RDMA transfer and page allocation; a user-space layer that coordinates on top of these primitives is, by construction, adding work rather than replacing it. This mirrors a broader trend of relocating application-level coordination into infrastructure, as observed in programmable networking where NFV-based frameworks eliminate redundant protocol handling [11].

Trade-offs and Scope. Kernel-level implementation introduces engineering costs: kernel modules are harder to debug, bugs risk system-wide crashes, and the development cycle is slower. RMC also requires Linux with RDMA-capable NICs, limiting deployment to datacenter environments. We mitigate these risks through write-once VMR semantics (which eliminate most concurrency hazards) and by confining RMC to a loadable kernel module rather than modifying the core kernel. The kernel-level approach is justified specifically for disaggregated multi-node deployments. For single-node serving, the kernel already provides efficient local sharing via `/dev/shm`, and user-space prefix caching works well without cross-node coordination.

Generalization Beyond KV Cache. The content-addressed PGAS model generalizes to other distributed AI data with fixed-size, content-identified semantics: MoE expert parameters [8] (fixed-size tensors identified by expert index, as demonstrated in [16]), LoRA adapter weights [10] (identified by adapter name with fixed dimensions per layer), intermediate activations in pipeline parallelism (identified by layer and micro-batch), and gradient checkpoints during training. The core requirement is that data size is known at deployment time and content identity can be computed without coordination. NTSM [17] generalizes this principle into a unified OS-level memory framework that transparently extends node-local memory to cluster scope, removing the need for application-level distributed data management.

GPU-Direct RDMA. GPU-Direct RDMA [18] could in principle bypass the CPU entirely, enabling NIC-to-GPU DMA. However, integrating it with kernel-level VMR management is non-trivial: GPU-Direct requires pinning GPU memory and registering it with the NIC at fine granularity, which conflicts with the dynamic page-fault-driven VMR lifecycle; current GPU-Direct implementations also lack support for kernel-initiated RDMA from within a page-fault handler. We leave this integration as future work for a full system paper.

Fault Tolerance and Eviction. A node failure renders its VMRs inaccessible, but RMC's deterministic addressing simplifies recovery: any surviving node can recompute lost chunks from the original token sequences. Under memory pressure, the kernel evicts VMRs using LRU; the content-addressed structure also enables prefix-aware eviction that protects shared prefixes and evicts leaf chunks first.

7 Conclusion

We presented RMC, an in-kernel framework that rethinks KV cache management as an OS-level memory systems problem. By exploiting fixed per-token cache occupancy and content-determined cache index, RMC provides a content-addressed PGAS where any node computes the index and virtual address of any cached chunk without coordination, achieving 1.1–1.3× TTFT reduction over LMCache + Mooncake and up to 2.1× over full recompute. More broadly, the management scaling paradox (Eq. 1) confirms a general anti-pattern in disaggregated systems: when user-space stacks replicate functionality the kernel already provides, the resulting overhead scales with the metric being optimized. We believe this insight extends beyond KV cache to any distributed AI workload where data has fixed size and content-determined indexing and location, including MoE expert routing, LoRA

adapter serving, and pipeline-parallel activation transfer. As LLM context lengths and cluster sizes grow, the gap between user-space management cost and kernel-native operation will only widen, making OS-level redesign increasingly important for inference efficiency.

References

[1] Meta AI. 2024. The Llama 3 Herd of Models. arXiv:2407.21783

[2] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 645–659. doi:10.1145/3037697.3037738

[3] 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*. 469–487.

[4] Yann Collet. 2024. xxHash: Extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash

[5] LMCache Contributors. 2024. LMCache: Reducing Redundancy and Boosting Efficiency in KV-Cache Management for LLM Serving. https://github.com/LMCache/LMCache

[6] DPDK Project. 2024. Data Plane Development Kit (DPDK). https://www.dpdk.org/

[7] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*.

[8] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. arXiv:2101.03961 [cs.LG]

[9] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. 649–667.

[10] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *Proceedings of the 10th International Conference on Learning Representations (ICLR '22)*.

[11] 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.2024.3475028

[12] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. 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)*. 611–626. doi:10.1145/3600006.3613165

[13] Shiju Li, Wenxuan Bi, Jiajun Liu, Yuhui Li, and Peng Zuo. 2025. FlowKV: A Disaggregated Inference Framework for Large Language Models with Coupled Compute-Communication. arXiv:2504.03775 [cs.DC]

[14] 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. arXiv:2310.07240 [cs.LG]

[15] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*. 843–857.

[16] Amir Noohi, Mostafa Derispour, and Antonio Barbalace. 2025. RMAI: Rethinking Memory for AI (Inference). In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25)*.

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

[18] NVIDIA Corporation. 2024. GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/

[19] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Niccolini, Sachin Mehta, Christos Kozyrakis, Ravi Netravali, Omer Levy, and Syed Fahad. 2025. Marconi: Prefix Caching for the Era of Hybrid LLMs. In *Proceedings of the 8th Conference on Machine Learning and Systems (MLSys '25)*.

[20] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st International Symposium on Computer Architecture (ISCA '24)*.

[21] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. 2022. A Scalable Actor-based Programming System for PGAS Runtimes. arXiv:2107.05516 [cs.DC]

[22] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-Centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST '25)*. 155–172.

[23] 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)*. 315–332.

[24] SPDK Project. 2024. Storage Performance Development Kit (SPDK). https://spdk.io/

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems 30 (NeurIPS '17)*. 5998–6008.

[26] 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*. 52–59.

[27] 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. 1172–1188.

[28] Shuo Yang, Yifan Qiao, Kaiqi Chen, Xingda Wei, Yongji Wu, Yao Lu, Rong Chen, Lidong Zhou, and Yaliang Li. 2025. KVDirect: Distributed Disaggregated LLM Inference with Direct KV Cache Access. arXiv:2501.14743 [cs.DC]

[29] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*.

[30] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. 521–538.

[31] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Ming Tang, Fan Yang, Yong Li, Yuqing Yang, Lili Qiu, Mao Yang, and Lidong Zhou. 2025. Taming the Titans: Efficient LLM Inference Serving with Prefix Sharing. In *2025 USENIX Annual Technical Conference (USENIX ATC '25)*.

[32] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2025. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and*

Implementation (NSDI '25).

[33] 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)*.