Disaggregated LLM Serving with CXL Shared Memory KV Cache at Rack-Scale

Dongha Yoon

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Application

Sam H. Noh, Chair
Ali R. Butt
Huaicheng Li

May 8, 2026
Blacksburg, Virginia

Keywords: CXL, Disaggregated LLM Serving, KV Cache, Shared Memory

Disaggregated LLM Serving with CXL Shared Memory KV Cache at Rack-Scale

Dongha Yoon

(ABSTRACT)

Disaggregated LLM serving improves resource efficiency by separating prefill and decode across workers, but introduces a new bottleneck: key/value (KV) tensors generated during prefill must be transferred to decode workers, typically through the network stack. CXL memory offers an attractive alternative by enabling multiple hosts to share a large byte-addressable memory pool, allowing KV blocks to be published and consumed without network transfer. Realizing this vision is challenging because current CXL devices provide neither cross-host atomic operations nor full-device cache coherence, and expose only raw shared memory. This thesis presents TraCT, a rack-scale LLM serving system that uses CXL shared memory as both a network-free KV-transfer substrate and a rack-scale prefix-aware KV cache. TraCT overcomes three key challenges imposed by current CXL hardware: inter-node mutual exclusion without hardware atomics, correct visibility on non-coherent shared memory, and portable shared-memory data structures without shared pointers. To this end, it introduces a two-tiered lock, a software-managed coherence discipline, and a shared object directory. Implemented as a general-purpose CXL shared-memory library with a vLLM external KV-caching plugin, TraCT achieves up to 2.6x reduction in average time-to-first-token (TTFT), 3.0x lower P99 latency, and 1.9x higher peak goodput than RDMA-based baselines on a real-world multi-turn conversational workload. These results demonstrate that CXL shared memory is a practical and efficient substrate for KV transfer and prefix caching in rack-scale disaggregated LLM serving.

Disaggregated LLM Serving with CXL Shared Memory KV Cache at Rack-Scale

Dongha Yoon

(GENERAL AUDIENCE ABSTRACT)

When you ask an AI assistant a question, your request travels through a surprisingly complex pipeline inside a data center. Modern AI systems split the work across multiple specialized servers: one server processes your input and prepares a large block of intermediate data, and another server uses that data to generate your response token by token. For this hand-off to work, the intermediate data—which can reach hundreds of megabytes for a single conversation—must travel between the two servers before the response can begin. Today, this transfer happens over a high-speed data center network. Data is packaged, sent through a network card, routed across cables and switches, and unpacked on the other end. Even inside a single data center rack, this round trip introduces delays measured in milliseconds—an eternity at the speeds modern AI hardware operates. Worse, as more users send requests simultaneously, these transfers compete for limited network bandwidth, causing response times to spike unpredictably under load. The root cause is a mismatch between where the data lives and how it gets moved. Accessing data directly from memory is orders of magnitude faster than sending it over a network: memory operates at nanosecond timescales and bandwidth measured in tens to hundreds of gigabytes per second, while network transfers add protocol overhead, buffering delays, and contention that push effective latency far higher. The servers in a rack are physically close to each other, yet today's software forces their data to take a long detour through the network stack. A new hardware technology called CXL addresses this mismatch by allowing multiple servers in a rack to share the same pool of physical memory directly. Instead of packaging data and sending it over the network, one

server can write directly into shared memory and another can read it back immediately—much like a program reading from its own RAM, but across machine boundaries. This thesis presents **TraCT**, a system that exploits CXL to eliminate the network transfer in AI serving entirely. With TraCT, the server that processes your prompt writes the intermediate data straight into shared memory, and the server that generates your response reads it back without any network involvement. The core challenge is that current CXL hardware was not designed with the coordination features that software normally relies on, so TraCT rebuilds those guarantees from scratch in software. Experiments on a real CXL-equipped system show that TraCT reduces the time users wait for an AI response by up to 2.6 times and can handle nearly twice as many simultaneous users before response quality degrades—all without requiring faster hardware or more servers. These results demonstrate that rethinking how servers in a rack share data, rather than simply scaling up network infrastructure, offers a practical path to faster and more efficient AI systems.

Dedication

I can do all this through him who gives me strength.

(Philippians 4:13, NIV)

Lord, I am grateful that You have walked with me throughout my life, granting me good opportunities and the wisdom and strength to seize them. You led me to study at Virginia Tech, to publish papers alongside wonderful advisors and colleagues, and to receive an internship opportunity in Silicon Valley. I also felt Your hand through the process of securing a full-time position and transitioning from the doctoral to the master's program. What I could never have accomplished on my own in short time, You brought to pass through the hands of many people around me. I pray that these worldly achievements may be used in a way that truly matters, as a testimony of faithfulness in my walk as a Christian. Above all, I am thankful that You sent Jiyoon, the most lovely and wise woman in the world, as my partner. Together with her, as companions in faith, I will cherish and support one another as we embrace whatever lies ahead.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Sam H. Noh. As an advisor, a mentor, and a person, he is simply the best. He supported me in every possible way throughout my time at Virginia Tech, opening doors I could not have opened on my own — including the opportunity to pursue an internship at SK Hynix. The lessons I have learned from him, both technical and personal, will stay with me for the rest of my career. I also thank my committee members, Professor Ali R. Butt and Professor Huaicheng Li, for their encouragement and constructive feedback. Their supportive approach and thoughtful guidance helped me grow significantly as a researcher. I am grateful to the team at SK Hynix America SOLAB for providing an outstanding research environment during my internship. Access to cutting-edge hardware, including the CXL shared memory device used in this thesis, made it possible to evaluate TraCT on real industry-grade equipment. The dedication of every team member made the experience both productive and genuinely enjoyable. I would also like to thank the faculty of the Virginia Tech Department of Computer Science for their excellent graduate courses. The knowledge and technical skills I acquired in the classroom proved directly useful in this research — from systems design to performance evaluation — and laid the foundation upon which this thesis was built. Finally, I would like to thank the staff of the Virginia Tech Graduate School, the Department of Computer Science, and the Cranwell International Center. When I made the decision to transition from the doctoral program to a master's degree and complete my studies on an accelerated timeline, everyone involved — including Graduate Dean Aimée Surprenant — handled the process with remarkable speed and kindness. Their support made a stressful transition feel manageable, and I am deeply appreciative.

Contents

List of Figures

List of Tables

List of Abbreviations

CICL	Context-Isolated Cacheline
AVX	Advanced Vector Extensions (Intel)
CAS	Compare-And-Swap
CXL	Compute Express Link
DMA	Direct Memory Access
GPU	Graphics Processing Unit
ITL	Inter-Token Latency
KV	Key/Value (as in KV cache)
LLM	Large Language Model
LRU	Least Recently Used
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
PCIe	Peripheral Component Interconnect Express
RDMA	Remote Direct Memory Access
SLO	Service Level Objective
TTFT	Time-To-First-Token

Chapter 1

Introduction

Large language models (LLMs) continue to grow in scale and capability, driving rapid deployment across industry and research. To improve utilization and reduce cost, modern LLM serving systems are increasingly adopting disaggregated architectures that separate the compute-intensive prefill phase from the latency-critical decode phase. Systems such as DistServe [49], Splitwise [32], Preble [36], and NVIDIA's Dynamo [3] show that decoupling these phases enables independent scaling and improved throughput. However, this architectural shift introduces a new bottleneck: the movement of key/value (KV) tensors. As model sizes and context lengths increase, the volume of KV data exchanged between prefill and decoding workers grows to hundreds of megabytes per request, making KV transfer a dominant factor in both request latency and peak system throughput [32, 49].

Today, disaggregated serving stacks overwhelmingly rely on network-based KV transfers, commonly using RDMA-based substrates such as UCX [35] and NIXL [4]. Even when prefix caching eliminates KV recomputation on cache hits, the cached blocks must still be transferred to the decoding worker over the network, incurring NIC serialization overhead and consuming bandwidth that saturates under high concurrency. This network hop significantly inflates prefill latency, increases tail variability, and constrains overall throughput. Prefix-aware caching systems such as LMCache [10] and Mooncake [33] improve reuse but still route all KV traffic across the network, leaving network serialization and congestion as persistent performance limitations.

Meanwhile, Compute Express Link (CXL) [1] has emerged as a promising substrate for rack-scale systems. CXL Type-3 devices provide large, byte-addressable memory pools that multiple hosts can concurrently map using load/store semantics, without involving the network stack. This raises a natural question: Can CXL shared memory replace RDMA as the transport substrate for disaggregated LLM serving, eliminating the network hop entirely?

Using CXL to directly publish and consume KV blocks across nodes could fundamentally change the performance and cost profile of LLM inference. Prefill workers write KV blocks into shared memory via GPU–CXL DMA, and decoding workers read them back with no NIC involvement, no host-to-host copies, and no network-induced variability.

We present **TraCT**, a rack-scale LLM serving system that uses CXL shared memory as both (1) a network-free KV-transfer substrate and (2) a rack-scale prefix-aware KV cache. [1] TraCT tightly couples KV ***Tra***nsfer and ***C***aching ***T***ogether, enabling prefill and decoding workers across the rack to communicate through a unified CXL memory pathway. TraCT removes the RDMA hop entirely: GPUs write newly generated KV blocks directly into CXL memory and fetch reusable KV blocks back through direct GPU–CXL DMA.

Building TraCT, however, is not straightforward. Current CXL devices provide no cross-node atomic operations, do not guarantee coherence across the full device capacity, and expose only raw byte-addressable memory. As a result, even simple operations such as updating metadata, coordinating access, and ensuring visibility require careful software design. To this end, TraCT addresses three fundamental challenges imposed by today's CXL hardware, providing effective solutions for each.

1. **Inter-node mutual exclusion without hardware atomics.** CXL Type-3 devices lack cross-node atomic instructions. To resolve this challenge, we introduce a two-

[1] A preliminary version of this thesis has been published as an arXiv paper [45].

tiered software lock mechanism that bounds contention and supports predictable lock acquisition.

2. **Correct (meta)data visibility on non-coherent shared memory.** Current CXL Type-3 devices expose limited or no hardware cache-coherent memory capacity [17, 19]. Without cross-node coherence, even mutually exclusive writers may expose stale metadata. We show that to guarantee cross-node coherence in a non-coherent shared memory setting, three conditions need to be met. We implement TraCT to satisfy these conditions, and we also show that the solution we propose is rack-scalable.

3. **Shared-memory data structures without shared pointers.** As hosts may map a shared CXL device at different virtual base addresses, absolute virtual pointers are not portable across nodes. We present a solution that encodes inter-object references as offsets from the base of the shared memory region rather than as raw pointers. Each host can then reconstruct a valid local pointer by adding the offset to its local base address. This enables shared data structures to be created once in CXL memory and safely traversed by any host.

We implement TraCT as a KV caching and transfer plugin for the Dynamo–vLLM inference framework [3, 5] and evaluate it against NIXL/UCX and LMCache plugins under both synthetic and real-world multi-turn conversational workloads. Our results show that CXL shared memory, through TraCT, is a viable and effective alternative to RDMA for KV transfer. Under realistic workloads, TraCT improves peak goodput by up to 1.9×, reduces average time-to-first-token (TTFT) by up to 2.6×, and lowers P99 TTFT by up to 3.0×.

The rest of this thesis is organized as follows. Chapter 2 provides background on CXL shared memory, LLM inference, and the role of KV management in disaggregated serving systems. Chapter 3 presents the design of TraCT, including its two-tiered synchronization mecha-

nism, metadata visibility strategy, shared-memory allocator, and prefix-aware KV cache. Chapter 4 evaluates TraCT across microbenchmarks and end-to-end inference workloads. Chapter 5 discusses related work, and Chapter 6 concludes.

Chapter 2

Background and Motivation

2.1 CXL Shared Memory

Compute Express Link (CXL) is an interconnect standard built on top of PCIe that enables high-bandwidth, low-latency communication between CPUs and devices such as accelerators and memory expanders. CXL defines three sub-protocols: CXL.io provides PCIe-compatible semantics for device enumeration and configuration; CXL.cache allows a device to access host memory coherently; and CXL.mem exposes device-attached memory to the host as a byte-addressable region. These protocols are supported by three device types: Type-1 devices (e.g., accelerators without device memory) use CXL.io and CXL.cache; Type-2 devices (e.g., accelerators with device memory) support all three protocols; and Type-3 devices support CXL.io and CXL.mem, providing extended memory that CPUs can map and access directly via load/store operations.

While prior work has used CXL memory either as a transparent extension of local DRAM or as a near-memory capacity tier [22, 23, 26, 39, 41, 50], other systems have begun to explore it as a rack-scale shared-memory substrate accessible from multiple hosts [17, 40, 43, 47, 51]. In current deployments, a CXL Type-3 device typically exposes its memory capacity as a DAX (Direct Access) region. DAX maps device-backed memory directly into the host address space as byte-addressable memory, allowing applications or the kernel to access it with ordinary load/store instructions rather than through the block I/O stack [38]. From the

perspective of a single host, this interface resembles local DRAM: loads, stores, and DMA operations proceed through the normal memory hierarchy. However, when multiple hosts attach to the same device, the memory is not automatically kept coherent across hosts. Any coherence support, when present at all, is typically limited to small device-specific regions and does not extend to the full device capacity [16, 17, 19, 42]. As a result, software must explicitly manage synchronization, visibility, cacheline flushing, and ordering when sharing CXL memory across nodes [17, 47].

CXL shared memory is nevertheless attractive for rack-scale systems. It enables load/store and DMA access without traversing the network stack, avoids NIC buffer copies, and allows multiple compute nodes to access the same physical memory region concurrently. At the same time, it introduces new systems challenges: current CXL Type-3 devices provide no cross-node atomic operations, and the lack of full-device coherence requires software to reason carefully about visibility and update correctness at cacheline granularity. TraCT leverages CXL shared memory both as an inter-node KV transfer substrate and as a rack-scale KV cache, and addresses these synchronization and coherence challenges.

2.2 LLM Inference

Modern LLMs such as GPT, Gemini, and Llama [14, 24, 31] are decoder-only transformers. At their core, each model is a stack of identical decoder blocks, and each block applies two operations in sequence: a masked self-attention layer, which captures contextual relationships across the token sequence, and a feed-forward network (FFN), which applies a position-wise nonlinear transformation.

Masked self-attention. Given an input matrix $X \in \mathbb{R}^{N \times d}$ of N token embeddings of

dimension d, the attention layer computes three linear projections:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V, \tag{2.1}$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_h}$ are learned weight matrices. The output is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_h}}\right) V. \tag{2.2}$$

Intuitively, Q encodes what the current token is looking for, K encodes what each past token contains, and V carries the content to be retrieved. The softmax over $QK^\top/\sqrt{d_h}$ assigns a relevance score to each past token, and the output is the correspondingly weighted sum of value vectors. The causal mask restricts each token to attend only to itself and earlier tokens, which is required for autoregressive generation. In practice, modern LLMs employ *multi-head attention*: H independent heads operate in parallel on $d_h = d/H$ dimensional subspaces, with their outputs concatenated and projected.

Two categories of tensors arise from this computation, and they have fundamentally different lifetimes. The weight matrices W^Q, W^K, and W^V are fixed after training and occupy a constant portion of GPU memory throughout serving. The intermediate K and V tensors, by contrast, are produced anew for every input token and must be retained for all subsequent steps of the same request—these are the *KV cache*. It is this asymmetry that gives rise to the two-phase structure of LLM inference.

Prefill. During prefill, the full input prompt of length N is processed in a single forward pass. Every layer computes attention across all N tokens, yielding $O(N^2)$ interactions. This phase is compute-intensive and highly batchable. The model produces the first output token, and a sequence of N key and value (KV) tensors—one per input token—which form the initial KV cache. The latency from request arrival to the generation of the first output

token—commonly referred to as time-to-first-token (TTFT)—is therefore dominated by the prefill stage.

Decode. After prefill, inference enters a token-by-token decode phase. At step t, the model computes query (Q)/K/V vectors for the newly generated output token, and attends over all previously stored K/V tensors. Unlike prefill, per-step computation is small, but the KV cache grows to $N+t$ entries. Thus, the decode phase becomes memory-bound, not compute-bound. The primary performance metric for the decode stage is the inter-token latency (ITL), also referred to as time between tokens (TBT), which measures the time elapsed between successive token generations and directly determines the perceived responsiveness of the system.

KV cache size. The KV cache grows linearly with both sequence length and model depth. For example, Llama3 405B has 126 layers, 8 key/value heads with a head dimension of 128, 2 tensors per layer (key and value), and uses FP16 precision (2 bytes per element). The KV cache size per token is thus $126 \times 8 \times 128 \times 2 \times 2 = 516{,}096$ bytes ≈ 504 KB per token. For a sequence of 8,192 tokens, a single request therefore requires approximately 4 GB of KV cache memory, quickly exhausting GPU memory at scale.

Across recent systems work, the KV cache has emerged as the component that most tightly couples compute throughput, memory capacity, and communication efficiency. This motivates designs that reduce KV recomputation to improve reuse or distribute KV storage across multiple memory tiers and nodes [3, 10, 33, 44].

2.2.1 KV Management and Disaggregated LLM Serving

KV caching mechanisms interact closely with how an LLM serving system is architected. Within a single node, KV tensors are typically kept in GPU memory and managed by

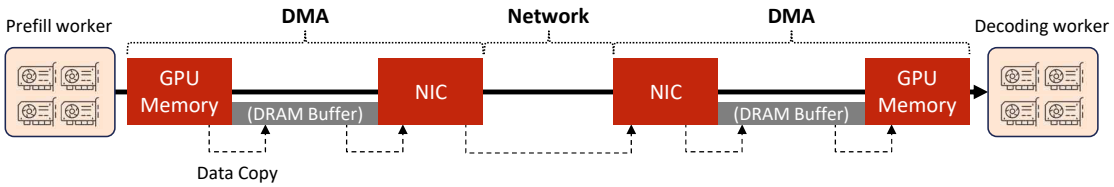


Figure 2.1: KV transfer path in disaggregated LLM serving. A KV block must traverse four data movement steps—GPU memory to host DRAM, DRAM to NIC, across the network, and from remote DRAM to the decode GPU—before becoming available for token generation.

the inference engine. Systems such as vLLM [5] and SGLang [48], for example, optimize this intra-node KV management. Specifically, vLLM addresses GPU memory fragmentation and duplication problems by leveraging the operating system's memory paging mechanism, grouping KV tensors into fixed-size blocks, while SGLang introduces RadixAttention, which allows requests with common prefixes to share KV blocks across them [21, 48].

Beyond a single GPU or node, KV caching has been extended to additional tiers and to multi-node disaggregated settings. Systems such as Splitwise, DistServe, and Preble partition the pipeline into prefill workers, which run the compute-heavy prompt processing, and decoding workers, which perform latency-critical token generation [32, 36, 49]. NVIDIA's Dynamo generalizes this disaggregated architecture into a production-level framework [3]. This separation enables independent scaling of the two phases and improves GPU utilization. However, every request still requires transferring KV blocks from prefill to decoding workers over the network—current systems rely on RDMA-based substrates such as UCX and NIXL [4, 35] for this purpose. Even when prefix reuse is high, each KV cache hit must traverse the NIC and DRAM on both sides, incurring NIC serialization overhead and redundant memory copies, and making throughput sensitive to network congestion.

Figure 2.1 illustrates this transfer path. KV blocks generated in the prefill GPU must traverse multiple data movement steps before reaching the decode GPU: (1) GPU memory

to host DRAM via DMA, (2) DRAM to the NIC transmit buffer, (3) the network switch to the remote NIC, and (4) the remote DRAM to the decode GPU via DMA. Each step adds latency and consumes bandwidth on an independent resource—the PCIe bus, the NIC, and the network fabric—any of which can become a bottleneck under load. Moreover, as concurrency increases and multiple prefill workers simultaneously attempt to transfer KV data, NIC bandwidth saturates, causing transfers to queue and forcing prefill workers to retain KV blocks in GPU memory while awaiting acknowledgment from the decode side.

Beyond raw KV transfer, recent systems have also explored sharing KV state across disaggregated servers to exploit cross-request reuse. LMCache offloads KV blocks to a hierarchy of backends—from host DRAM to distributed object stores—enabling reuse across serving engines [10]. Mooncake constructs a cluster-wide KV pool spanning CPU memory and SSDs, integrated with separate prefill and decode clusters [33]. However, both systems retain network-based movement of KV tensors for cross-node reuse.

Disaggregated LLM serving therefore improves modularity and resource scaling, but also moves KV transfer into the critical path. In contrast to network-centric designs, this work explores using CXL shared memory as a rack-scale KV cache and transfer layer directly accessible to all participating hosts and GPUs through load/store and DMA. This approach removes the network hop for KV reuse but raises new challenges such as synchronization, consistency, and data management on non-coherent CXL memory. TraCT addresses these software support challenges, which we discuss in detail in Chapter 3.

Chapter 3

TraCT Design

This chapter presents the design of TraCT, a rack-scale LLM serving system that uses CXL shared memory as both a KV-transfer substrate and a rack-scale prefix-aware KV cache. We begin by identifying the fundamental challenges imposed by current CXL hardware, then provide a system overview, and finally describe the design of each component in detail.

3.1 Challenges

CXL memory expanders expose only raw byte-addressable memory shared among nodes and provide neither cross-node atomic operations nor full-device cache coherence. Consequently, even basic tasks such as coordinating metadata updates or ensuring visibility of shared state must be implemented entirely in software. Any system that coordinates nodes over CXL shared memory must therefore address the following three challenges.

3.1.1 Mutual exclusion without cross-node atomics (C1)

Mutual exclusion is a fundamental requirement in a shared memory system, as nodes concurrently update shared metadata. While classical mutual exclusion relies on atomic read-modify-write instructions (test-and-set, compare-and-swap), CXL Type-3 does not provide such instructions across hosts. Thus, the challenge is to design a software-only mutual ex-

clusion mechanism for CXL shared memory.

3.1.2 Data visibility on non-coherent memory (C2)

Currently, some CXL Type-3 devices expose limited hardware cache-coherent memory capacity [17, 47], while some do not provide any such capacity [40]. This limitation is not merely an artifact of first-generation hardware: recent analyses show that hardware coherence at the scale of multi-terabyte CXL memory is fundamentally impractical due to snoop-filter size, power constraints, and cross-host coherence traffic [16, 17, 19, 42].

Without (or with limited) hardware cache coherence, writes performed by one node may not become immediately visible to others. Even with mutual exclusion, two visibility hazards remain. First, a *write-back hazard* arises when modified data lingers in the writer's private cache, causing remote nodes to observe a stale value in CXL memory. Second, a *stale-read hazard* arises when a reader retains a cached copy of a value that another node has already overwritten and flushed, causing the reader to observe obsolete data despite the newer value already residing in the device.

Beyond these two hazards lies a more fundamental problem: *write-write conflicts*. If two nodes load the same 64-byte cacheline into their private CPU caches, modify disjoint words within that cacheline, and then flush their copies to the device, the later flush silently overwrites the entire cacheline, including the earlier update. This is not a race in the classical sense, since the two writes target different memory locations, yet one update can still silently destroy the other. Thus, the challenge is to guarantee correct data visibility and update preservation on non-coherent CXL memory, and to do so in an efficient manner.

3.1.3 Shared object management and pointer portability (C3)

While the CXL memory is shared, every process running on each node does so in an independent context. In particular, raw virtual pointers cannot be shared across processes. An address that is valid on one node may be unmapped or may refer to an unrelated location on another node. As a result, conventional pointer-based data structures cannot be placed directly in CXL shared memory, even though the memory itself is physically shared. Thus, the third challenge is to support shared data structures whose internal references remain valid across all nodes despite differences in virtual address mappings. That is, any node must be able to locate and traverse shared objects without assuming a common virtual address space or fixed virtual base for the CXL device.

3.2 TraCT Overview

3.2.1 Architecture

Figure 3.1 illustrates the architecture of TraCT. Within each rack, multiple prefill and decoding workers, residing in multiple nodes, connect to a shared CXL Type-3 device. TraCT exposes the device capacity as a byte-addressable DAX-mapped region uniformly accessible to all participating nodes, forming a common substrate for shared metadata and KV storage.

A shared-memory runtime manages this region through three key primitives. It provides (1) **inter-node locks** for mutual exclusion across processes on different nodes without hardware atomic support, resolving C1 (Section 3.3); (2) a **memory allocator** for managing the shared CXL address space; and (3) an **object directory** for publishing and resolving shared metadata across nodes despite different virtual address mappings, resolving C3 (Section 3.5).

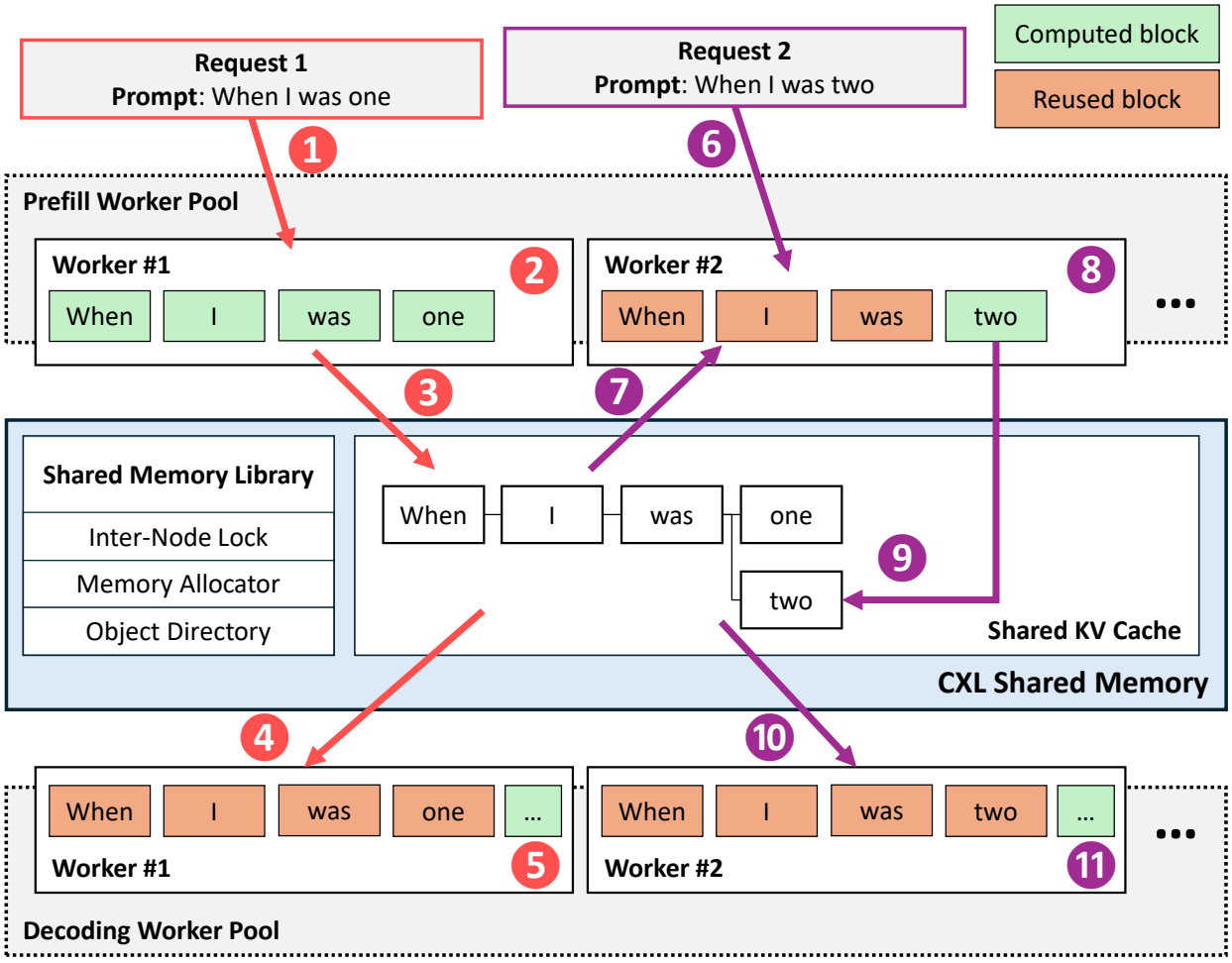


Figure 3.1: TraCT architecture overview.

TraCT also enforces a software coherence protocol to ensure correct visibility and update ordering, resolving C2 (Section 3.4).

Built on top of these primitives, TraCT implements a shared KV cache that stores both KV blocks and prefix-cache metadata in CXL memory, enabling both inter-node KV transfer and rack-scale prefix reuse (Section 3.6). The end-to-end inference flow operates over this shared-memory substrate as follows. When a request arrives (1), a prefill worker processes the prompt and generates KV blocks (2). These blocks are written into the shared CXL region via GPU–CXL DMA (3). A decoding worker then fetches the blocks directly from

CXL memory into GPU memory and performs token generation (4, 5).

When a later request shares a prompt prefix with an earlier one (6), the prefill worker consults the shared prefix cache, copies matching KV blocks from CXL memory into GPU memory (7), and computes only the missing blocks, which are appended to the shared cache (8, 9). The decoding worker then fetches the full KV set (10) and generates output tokens (11). TraCT thus unifies inter-node KV transfer and prefix-aware caching in a single shared-memory data path.

3.2.2 Shared Data Structures

TraCT places a set of shared data structures in CXL shared memory to coordinate KV caching and transfer across nodes. Table 3.1 summarizes these structures, which fall into five functional categories described in the following sections. The global lock (Section 3.3) provides mutual exclusion across nodes without hardware support. The allocator and object directory (Section 3.5) manage the CXL address space and support rack-wide object and memory sharing. The prefix-cache index and KV cache entries (Section 3.6) maintain the rack-wide KV cache metadata for prefix reuse.

3.3 Ensuring Mutually Exclusive Access

To address C1, TraCT must provide mutual exclusion over CXL shared memory using only load, store, and cacheline flush instructions. Existing CXL shared memory systems address this in one of three ways, each with significant drawbacks. First, some systems rely on a small device-specific coherent region and assume that remote atomics within it behave correctly [17, 47]. This simplifies lock implementation, but the coherent region is limited in

Table 3.1: Shared metadata structures in TraCT.

Structure	Fields	Size	Unit	Protection
Global lock	`req[n]`	64 B	1 per node	†CICL
	`granted[n]`	64 B	1 per node	CICL
	Allocated bitmap	64 B	1 per group	CICL+*TTL
	Group metadata	16 B	1 per group	CICL+TTL
Allocator	Bitmap words (1 bit/4 KB page)	64 B	1 per chunk	CICL+TTL
Object directory	Hash table size, lock pointer array	40 B	1	Read-only
	Hash entry	24 B	1 per entry	CICL+TTL
Prefix cache index	Config, lock/hash/eviction list	64 B	1	Read-only
	Runtime cache capacity status	16 B	1	CICL+TTL
KV cache entry	Hash key, KV block DAX offset, lock	24 B	1 per entry	Read-only
	State, refcount, eviction counters	40 B	1 per entry	CICL+TTL
	Eviction list entry	24 B	1 per entry	CICL+TTL

*TTL: Two-Tiered Lock (§3.3), †CICL: Context-Isolated Cacheline (§3.4)

size and not guaranteed across devices or generations. Second, systems such as cMPI [40] avoid locks entirely by using an $N \times N$ matrix of producer–consumer queues for N participants. This provides deterministic communication paths but requires $O(N^2)$ memory and forces each participant to scan N queues, making CPU cost grow with the number of nodes. Third, all metadata operations could be routed through a centralized server, as done in Beluga [43]. While common in network-based systems, this design contradicts the purpose of CXL shared memory: every operation—even a simple key lookup—must round-trip to the metadata server, reintroducing the communication overhead that CXL is meant to eliminate. Centralization also precludes general-purpose critical sections, since arbitrary application code cannot be serialized through a single remote server.

TraCT instead designs a mutual exclusion mechanism with three key properties. First, it uses only load, store, and cacheline flush instructions, avoiding hardware atomics and cross-node coherence. Second, its memory footprint and CPU overhead scale with only the number of participating nodes, rather than with the number of locks or contenders. Third, it offers

a general-purpose acquire/release interface for arbitrary critical sections, enabling higher-level components such as the prefix cache and memory allocator to use it directly. These properties are realized by a two-tiered software synchronization mechanism described below.

3.3.1 Two-tiered lock design

A straightforward approach would let every process across all nodes contend directly for a single global lock. Such a design is impractical, as the lock manager would need to track an unbounded and dynamically changing set of contenders, since processes may fork, exit, crash, or join at any time. Managing per-process identities, detecting failures, and reclaiming abandoned lock state would therefore become complex and expensive.

TraCT avoids this problem with a two-tiered locking design, as shown in Figure 3.2. Each node first serializes its local processes with a DRAM-resident *local_lock*, and only the winning local process is allowed to participate in the CXL-resident *global_lock*. The two locks share the same logical lock identifier, and a process must acquire the *local_lock* before requesting the *global_lock*. This hierarchy collapses potentially many per-process contenders into at most one representative per node, bounding global contenders to at most one per node rather than one per process. This approach is similar in spirit to hierarchical NUMA locks [8, 9, 12] that use a local lock to serialize intra-node threads before competing for a global lock. However, to the best of our knowledge, TraCT is the first to apply this principle to CXL shared memory, where hardware cache coherence and cross-node atomics are unavailable and the mechanism must be built using only loads, stores, and cacheline flushes.

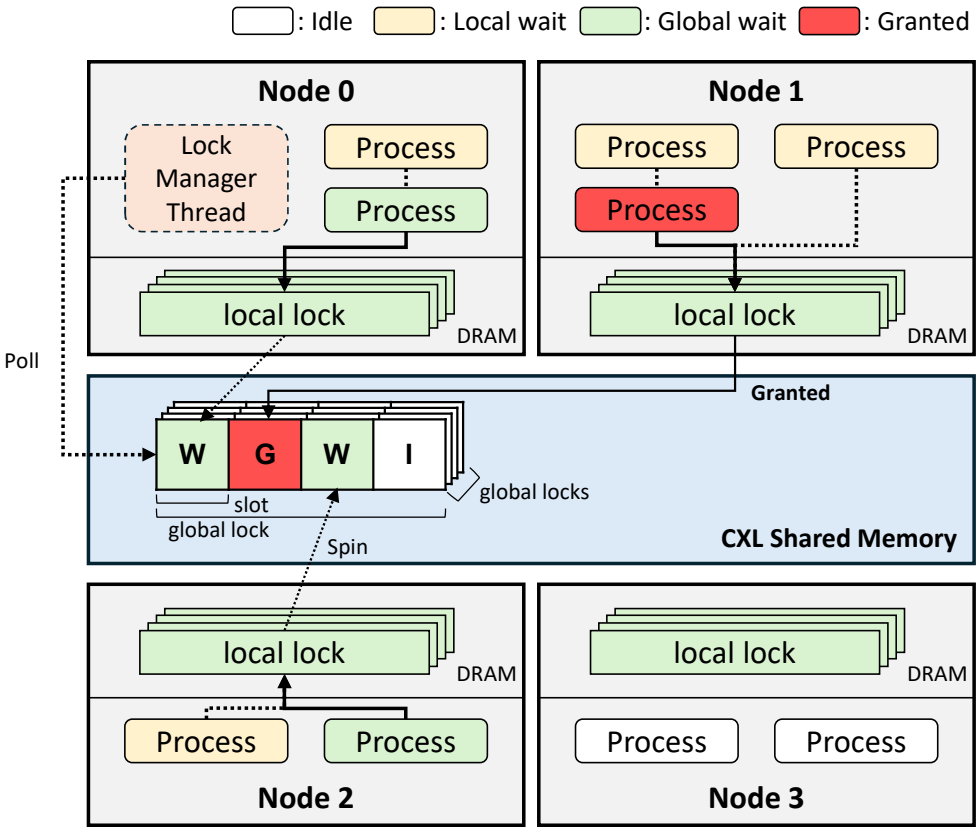


Figure 3.2: TraCT two-tier lock design, with a 4-node example.

3.3.2 Local lock (intra-node serialization)

Before competing for a global lock, a process must first acquire a node-local mutex in DRAM. The local lock is a conventional DRAM-resident mutex (e.g., `pthread_mutex`), which is readily available within a host because local CPU caches are coherent. Its role is not merely to protect local critical sections, but to enforce a structural invariant for the global layer: at any time, each node contributes at most one active requester to the global lock. This drastically simplifies inter-node lock management and isolates process churn within the node.

3.3.3 Global lock (inter-node mutual exclusion)

The global lock provides mutual exclusion across nodes over non-coherent CXL shared memory. As current CXL Type-3 devices provide neither cross-node atomics nor full-device coherence, the lock must be implemented using only ordinary loads, stores, and cacheline flushes. Each node owns a dedicated state slot in CXL shared memory, and a lock manager thread grants ownership by polling these slots in round-robin order. To acquire the lock, a node writes its slot to Waiting (W) and spins until the manager changes the state to Granted (G). To release the lock, it writes the slot back to Idle (I). This design keeps the protocol simple while ensuring that lock ownership is coordinated entirely through shared memory.

Naïve design and its scalability problem. A naïve implementation of this design allocates one lock structure per logical lock, with each structure containing N per-node state slots (Idle, Waiting, or Granted). The manager must repeatedly flush each slot's cacheline on every polling iteration to observe the latest state, resulting in $O(N \cdot L)$ cacheline flushes per polling round for L locks and N nodes. The space overhead is also substantial: each lock consumes $(1 + N)$ cachelines—one for metadata storing the lock identifier, current holder, and fairness counter, plus one per-node state slot—totaling 320 bytes for $N = 4$. As the number of locks grows, both the metadata footprint and the polling cost increase linearly, causing lock acquisition latency to rise sharply under contention.

Compressing and batching lock management. The simple protocol above is further optimized by a lock group abstraction that manages 512 logical locks within a single shared structure. Rather than storing one struct state per lock, each group maintains, for every node, two cacheline-sized bitmaps: `req` and `granted`. A lock handle identifies a lock by its group index and bit position within the bitmap. Acquiring lock b on node n therefore requires only setting bit b in `req[`n`]` and waiting until the manager sets the corresponding

Algorithm 1 Lock Manager: Batched Grant/Release

Require: N: number of nodes
Require: **req**$[N]$, **granted**$[N]$: 512-bit bitmaps per node
Require: **alloc**: used-state bitmap of this lock group

```
 1: procedure HandleLockGroup(epoch)
 2:     for n ← 0 to N − 1 do                                  ▷ release pass
 3:         granted[n] ← granted[n]  &  req[n]
 4:     end for
 5:     held ← ⋃_n granted[n]                                  ▷ grant pass
 6:     free ← alloc  &  ¬ held
 7:     for n ← 0 to N − 1 do
 8:         want ← req[n]  &  free  &  ¬ granted[n]
 9:         granted[n] ← granted[n]  |  want
10:         free ← free  &  ¬ want
11:     end for
12: end procedure

13: procedure Acquire(n, b)                                    ▷ process-side
14:     req[n][b] ← 1
15:     repeat
16:     until granted[n][b] = 1
17: end procedure

18: procedure Release(n, b)
19:     req[n][b] ← 0
20:     repeat
21:     until granted[n][b] = 0
22: end procedure
```

bit in `granted[`n`]`.

This grouped representation compresses 512 locks into only $(2 + 2N)$ cachelines per group: two cachelines for group metadata and two cachelines per node for `req` and `granted`. For $N = 4$ nodes, a group occupies 640 bytes for 512 locks, or 1.25 bytes per lock, compared with 320 bytes per lock in the naïve per-lock design—a 256× reduction in metadata overhead. More importantly, grouping enables the manager to process hundreds of locks in bulk using bitmap operations. As shown in Algorithm 1, the manager executes two passes over each lock

group. In the *release pass* (lines 2–4), it clears each `granted[`n`]` bit whose corresponding `req[`n`]` bit has been cleared, thereby detecting all releases without polling locks individually. In the *grant pass* (lines 5–12), it computes the set of currently held locks, derives the free locks, and grants waiting requests in round-robin order across nodes. TraCT further accelerates these operations with Intel AVX-512 vector instructions [18], allowing all 512 bits in a group to be processed simultaneously.

3.4 Cache Coherence

As discussed in Section 3.1 (C2), CXL Type-3 devices provide little or no cross-host cache coherence. In a shared-memory setting without global hardware coherence, a write issued by one host may remain in that host's private CPU cache until explicitly or implicitly evicted, and may not become visible to other hosts in a timely manner. Likewise, a reader may continue to observe a stale locally cached copy indefinitely. To provide software-managed coherence for shared metadata in this non-coherent CXL setting, TraCT requires three conditions to hold: the CICL condition, the Flush Protocol (FP), and the Exclusive Writer (XW) condition. We first describe each condition and then explain how they jointly provide coherence.

3.4.1 CICL Condition

Context-Isolated Cacheline (CICL) requires that each cacheline belongs to a single logical context, so that independently updated contexts never share a cacheline. As discussed in Section 3.1, without hardware coherence, sharing a cacheline across contexts can cause write-write conflicts where one host's writeback silently overwrites another's. CICL avoids this by

enforcing that each cacheline is written on behalf of at most one context.

Consider the TraCT metadata contexts listed in the "Fields" column of Table 3.1. These fields are critical to the correct operation of TraCT and must therefore remain coherent. Although some fields are smaller than a cacheline, they are not packed together with other independently updated fields. Instead, TraCT aligns each such field to a cacheline boundary using `__attribute__((aligned(64)))` so that distinct contexts occupy distinct cachelines. This increases space overhead for small fields, but the overhead is modest. In general, an application's contexts can be identified at design time as all shared metadata residing in non-coherent CXL shared memory and shared across hosts.

3.4.2 Flush Protocol (FP) Condition

To address the write-back and stale-read hazards from Section 3.1, every access to CXL shared memory follows an explicit flush-and-fence discipline. Writes use a `flush-fence-update-flush-fence` sequence, while reads use `flush-fence-read`. Here, `update` denotes either a write or a read-modify-write operation. For writes, the first `flush-fence` pair removes any stale local copy before the update, and the second pushes the updated cacheline to the device and orders it before subsequent memory accesses. For reads, `flush-fence` invalidates any local cached copy so that the following load fetches the latest value from shared memory.

3.4.3 Exclusive Writer (XW) Condition

Finally, each CICL context must have a single writer at the time of update. In the common case, this condition is satisfied by requiring the writer to acquire a lock before modifying shared data. Readers on all hosts that synchronize through that lock will then observe the

updated state only after the writer completes its update and releases the lock. In TraCT, this condition is enforced by having the writer acquire the corresponding lock through the two-tiered lock mechanism before modifying shared metadata, as marked 'CICL+TTL' in the "Protection" column of Table 3.1.

There are, however, exceptions. For example, `req[n]` and `granted[n]` are themselves part of the lock implementation and therefore cannot rely on that same lock for protection. Even so, these fields still satisfy the exclusive-writer requirement because each entry has a unique designated writer, while readers merely poll until a target condition becomes true. Thus, whether a reader observes the value before or after the write is immaterial to correctness.

To illustrate how the three conditions interact, consider two shared contexts CXT1 and CXT2 residing in non-coherent CXL memory, accessed concurrently by hosts H1 and H2. Assume that CXT1 and CXT2 are shared contexts residing in non-coherent CXL memory and are accessed by hosts H1 and H2. If H1 and H2 concurrently update different contexts (e.g., CXT1 and CXT2), the CICL condition prevents write-write interference by ensuring that independently updated contexts never share a cacheline. The FP condition complements this by ensuring visibility: writes are pushed from the writer's cache to the device, and reads discard potentially stale local copies before fetching from shared memory. However, FP alone does not impose a global order between a concurrent read and write to the same context. For example, if H1 writes CXT1 while H2 reads it concurrently, H2 may observe either the old or the new value depending on timing. The XW condition eliminates this ambiguity by ensuring that each context has a single synchronized writer and that accesses to that context are properly serialized through the corresponding synchronization mechanism. Consequently, once the writer completes its update and releases the synchronization point, subsequent synchronized readers observe the new value in a well-defined order.

3.4.4 Programmability

In practice, manually placing `flush` and `fence` instructions and explicitly acquiring and releasing locks at every access site is error-prone: a single omission can silently violate visibility or permit a write-write conflict, with neither a compiler warning nor a runtime error. To address this, TraCT draws inspiration from the `std::lock_guard` idiom in C++. A `std::lock_guard` acquires a mutex in its constructor and releases it in its destructor, thereby bounding the critical section to the enclosing lexical scope without requiring an explicit unlock call. TraCT generalizes this pattern with a `CXL_lock_guard` that enforces both the FP and XW conditions within the same lexical scope. Its constructor acquires the two-tiered lock (XW) and issues a pre-access `clflush-fence` sequence (FP), while its destructor issues a post-access `clflush-fence` sequence (FP) and releases the lock (XW). As the guard object is stack-allocated, the critical section is naturally delimited by the enclosing block, and the compiler guarantees that the destructor executes at scope exit regardless of control flow. As a result, any code that accesses CXL-resident metadata through this guard automatically satisfies FP and XW, without requiring the programmer to reason about individual flush placement or lock pairing.

3.4.5 Overhead analysis

The flush-and-fence discipline and CICL alignment impose overhead compared to coherent DRAM, where hardware manages visibility transparently. TraCT's key insight is that this overhead is both bounded and largely hidden in the disaggregated LLM inference context. As shown in Table 3.1, the shared metadata subject to coherence management is small and fixed: the total number of cachelines requiring explicit flushing is determined at compile time and grows only with the number of nodes and logical objects, not with workload intensity.

The dominant occupant of CXL shared memory is KV block tensors, ranging from megabytes to gigabytes depending on the model and block-size configuration. These are written exclusively via GPU-to-CXL DMA, which bypasses CPU caches entirely and thus requires no explicit cacheline flushing. Moreover, GPU computation, KV tensor transfer, and metadata management operate in separate execution contexts and proceed asynchronously, so the metadata flush overhead is hidden behind the far more dominant costs of GPU computation and KV transfer.

3.5 Shared Object Directory and Memory Allocator

As discussed in Section 3.1 (C3), nodes do not share a common virtual address space: each node maps the same CXL device at a different virtual base address, so a raw virtual pointer that is valid on one node is meaningless on another. This creates a fundamental systems challenge for TraCT: every shared object in CXL memory must remain discoverable and accessible from any node, even though neither object identifiers nor internal references can rely on raw pointers.

TraCT supports two classes of shared state under this constraint. The first is a set of shared data structures, such as the prefix-cache index and message-passing structures for inter-node KV transfer, each spanning only a few cachelines. The second is the KV block payloads themselves, which may range from megabytes to gigabytes depending on the model and block granularity, and are continuously allocated and reclaimed as requests arrive and complete. Despite their different sizes and lifetimes, both classes of data must be discoverable by any node at any time without out-of-band coordination. To this end, TraCT introduces two complementary abstractions: a shared object directory for naming and discovery, and a memory allocator for managing the underlying CXL address space.

3.5.1 Shared object directory

The shared object directory solves the naming problem. Rather than attempting to publish every shared object individually, TraCT exposes only a small set of well-known root objects, for example, the prefix-cache index hash table. A node first looks up such an object by key in the directory, obtains its DAX offset, and then reconstructs a valid local pointer by adding that offset to its own CXL mapping base. From that point on, all internal references are expressed as DAX offsets rather than raw virtual pointers, allowing the structure to be traversed correctly from any node regardless of where the device is mapped in that node's address space.

This design confines directory-management overhead to the root level while naturally supporting complex hierarchical structures. For example, in the prefix-index hash table, internal bucket links, chained entries, and eviction-list pointers are all represented as offsets. As a result, once a node resolves the root object, the entire structure becomes portable and self-contained across nodes.

Beyond pointer portability, this design also improves performance over traditional distributed object stores. A conventional store such as Redis [34] operates over the network and passes objects *by value*: to read an object, the client sends a request, and the store serializes and returns a copy. In contrast, TraCT's object directory operates *by reference*: an object lookup returns a DAX offset that directly references the structure, allowing the caller to access the data in place with neither a network round trip nor a data copy. This is particularly beneficial for read-heavy workloads such as prefix-cache lookups, where the entire index already resides in shared CXL memory and can be traversed directly.

3.5.2 Memory allocator

The memory allocator manages the raw CXL shared-memory region from which both metadata objects and KV payloads are carved out. As allocation occurs frequently, especially for KV blocks, directly coordinating every fine-grained allocation through shared metadata would create excessive contention. TraCT therefore adopts a two-tiered allocator that mirrors its two-tiered locking design.

At the global level, a chunk allocator manages a coarse-grained allocation bitmap in CXL shared memory and assigns fixed-size pages (e.g., 4 KB) to nodes. At the local level, each node manages the space within its acquired chunks using a heap allocator whose free lists reside entirely in node-local DRAM. This allows cacheline-granular allocation and reclamation to proceed locally, without touching shared CXL metadata on every `alloc`/`free` operation. Inter-node coordination is thus required only when a node acquires or returns chunks, while routine small-object management remains node-local and contention-free.

Together, these mechanisms make CXL shared memory usable as a cross-node heap. The shared object directory provides portable naming and discovery, while the allocator provides scalable address-space management. By expressing all internal references as DAX offsets, TraCT allows both the metadata structures and large KV payloads to be shared safely across nodes despite virtual-address differences.

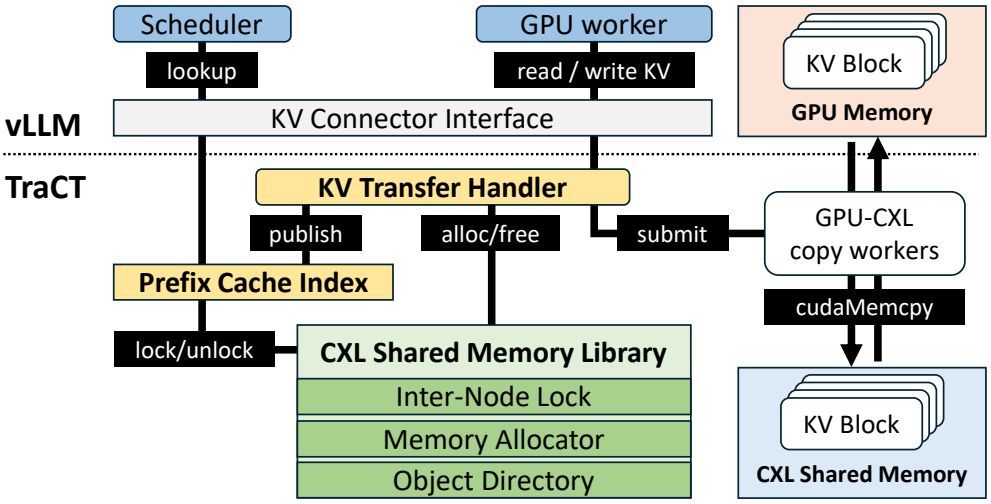


Figure 3.3: TraCT software stack.

3.6 KV Cache Management

3.6.1 Organization

Figure 3.3 shows the TraCT software stack. TraCT is organized as two layers with clearly separated responsibilities. The lower layer is the *CXL Shared Memory Library*, a general-purpose runtime that manages the raw CXL address space and provides safe cross-node access to shared memory. It exports three primitives that directly address the challenges in Section 3.1: an inter-node lock for mutual exclusion, a shared-memory allocator for dynamic space management, and an object directory for publishing and resolving shared objects across nodes. This layer is intentionally application-agnostic and provides a reusable substrate for any application that needs coordinated access to a non-coherent CXL shared-memory region.

Built on top of this substrate, the upper layer is the *CXL KV Connector*, the TraCT-specific component that integrates with vLLM through its KV connector interface. This layer turns shared CXL memory into a rack-wide KV-transfer and KV-reuse substrate for LLM inference. It consists of two components with distinct roles. The *Prefix Cache Index* forms the KV

cache control plane: it tracks which KV blocks exist in shared CXL memory, where they are located, and whether they are safe to reuse. The *KV Transfer Handler* forms the data plane: it moves KV blocks between GPU memory and CXL memory through asynchronous GPU–CXL DMA. These two components are exposed separately to vLLM. The *Scheduler* interacts with the *Prefix Cache Index* to determine reuse opportunities before dispatch, while the *GPU Worker* interacts with the *KV Transfer Handler* to publish newly generated blocks and fetch reusable ones.

3.6.2 Prefix Cache Index

The *Prefix Cache Index* is a hash table allocated in CXL shared memory and registered in the object directory under a well-known key, allowing any node to resolve it without out-of-band coordination. Each entry is keyed by the token-sequence hash that vLLM uses to identify a KV block, which TraCT takes directly from vLLM, and stores the metadata needed for rack-wide reuse: the DAX-relative offset of the KV payload, eviction metadata, a reference count, and an explicit state. An entry is in state XFER_IN_PROGRESS while its KV payload is being transferred into CXL memory, in VALID once the payload is fully published and globally visible, and in INVALID after eviction. This state machine allows the index to serve not only as a lookup structure, but also as an implicit synchronization point between producers and consumers. A decoding worker or later prefill worker can therefore discover a block through the index and determine, from the entry state alone, whether it is ready to be fetched.

The index is implemented to respect the constraints of non-coherent CXL memory. Structural mutations such as insertion and eviction are serialized by a single index lock, ensuring mutual exclusion across nodes (C1). Fields that are updated independently, such as per-

entry state, are laid out to satisfy the CICL condition (C2). All internal references, including hash-chain links and eviction-list pointers, are stored as DAX-relative offsets rather than raw virtual pointers, so the index remains portable across nodes despite different virtual mappings of the CXL region (C3).

3.6.3 KV Transfer Handler

The *KV Transfer Handler* manages all transfers between GPU memory and CXL shared memory through a pool of GPU–CXL copy workers that issue `cudaMemcpy()` calls asynchronously. A key design requirement is to avoid unintended staging through host DRAM. By default, a CUDA transfer between GPU memory and a non-pinned host region may be routed through an internal DRAM bounce buffer, which adds latency and consumes host memory bandwidth. TraCT eliminates this overhead by registering KV block regions in CXL shared memory with `cudaHostRegister()`, making them page-locked host memory from CUDA's perspective. This allows the GPU DMA engine to access the CXL-resident buffers directly over PCIe, enabling direct GPU–CXL transfers without intermediate copies through host DRAM.

The *KV Transfer Handler* interacts closely with the *Prefix Cache Index*. When publishing a new KV block, it obtains a destination region in CXL memory through the shared allocator and associates that region with a newly created index entry. When reusing an existing KV block, it resolves the payload location through the index and issues a direct GPU fetch from the corresponding CXL-resident buffer. Thus, the index determines visibility and ownership, while the transfer handler performs the actual data movement.

3.6.4 KV Cache Operations and Lifecycle

TraCT manages shared KV blocks through three operations: reuse, publication, and eviction. Together, these operations define the lifecycle of a KV block in CXL shared memory. A newly generated block is first published into the shared cache, later requests may reuse it through lookup and fetch, and the block is eventually evicted when it is no longer needed and space must be reclaimed.

Reuse. Before dispatching prefill, the *Scheduler* queries the *Prefix Cache Index* using the content hashes of the request's prompt blocks. For each matching entry, it determines whether the block can be reused and how much of the prompt can bypass recomputation. To prevent concurrent eviction while the block is in use, TraCT increments the entry's reference count before issuing any fetch. The number of consecutive hits determines the reusable prefix length for the request.

Once reuse candidates are identified, the *KV Transfer Handler* fetches the corresponding KV blocks from CXL memory into GPU memory. If an entry is already in state VALID, the block is fetched immediately. If it is still in XFER_IN_PROGRESS, the consumer waits until the producer advances the state to VALID, and then performs the fetch. No explicit point-to-point notification is required: the state transition itself serves as the synchronization signal between producer and consumer. After the transfer completes, the consumer decrements the reference count. The same mechanism is used both when a later prefill worker reuses a cached prefix and when a decoding worker fetches KV blocks produced by an earlier prefill worker. Thus, a single shared-memory protocol supports both prefix-aware KV reuse and inter-node KV transfer.

Publication. On a cache miss, a prefill worker generates the missing KV blocks on the GPU and invokes the *KV Transfer Handler* to publish them into shared CXL memory. For

each block, TraCT first allocates a payload region from the shared allocator and inserts a corresponding entry into the *Prefix Cache Index* with state XFER_IN_PROGRESS. This insertion makes the block visible to other workers while explicitly indicating that the payload is not yet ready for consumption. The transfer handler then issues a GPU–CXL DMA operation to copy the KV tensors into the allocated CXL region. After the transfer completes, TraCT advances the entry state to VALID, which serves as the publication point: from this moment onward, the block is globally visible and may be safely reused by other workers. This two-phase process cleanly separates reservation from activation, allowing concurrent workers to observe a consistent view of blocks that are still being published.

Eviction. When cache usage approaches a configured capacity threshold, TraCT reclaims space by evicting entries according to a replacement policy, LRU by default. Only entries whose reference count is zero are eligible, ensuring that no block is reclaimed while it is still being accessed. Under the protection of the index lock, the evictor marks the victim entry INVALID, removes it from the hash table and eviction list, returns its payload region to the shared allocator, and updates the cache-usage counter. Marking the entry INVALID before reclaiming its payload ensures that no later lookup can mistake reclaimed space for a reusable KV block. In this way, TraCT maintains a clean separation between logical cache visibility in the index and physical space ownership in the allocator.

Chapter 4

Evaluation

We evaluate TraCT to answer the following questions:

- Can CXL shared memory replace RDMA as a KV transfer substrate without sacrificing throughput or latency? (Section 4.2)
- Does TraCT's prefix-aware CXL cache improve end-to-end throughput and TTFT compared to RDMA-based and DRAM-based caching baselines? (Section 4.3)
- How does TraCT contribute to each component's performance? (Section 4.4)
- Does the two-tiered lock design scale correctly, and what is the overhead of software-managed cache coherence? (Section 4.5)

4.1 Evaluation Setup

Our evaluation environment is configured with 2 servers. Server 1 executes the benchmark client and decoding worker, while Server 2 is provisioned to perform only prefill tasks. This partitioning is specific to our controlled experimental study in contrast to production systems where these roles are generally co-located within a single node.

Hardware. Each server is equipped with an NVIDIA A6000 GPU (48 GB GDDR6) and 512 GB of host DRAM. For the baseline (NIXL/UCX [4, 35]) configuration, the servers

are interconnected using a 100 Gbps Mellanox MT2892 NIC. For the CXL experiments, we employ a Niagara 2.0 device, a second-generation CXL Type-3 memory expander that provides byte-addressable, load/store–accessible memory through the CXL.mem interface. The Niagara 2.0 provides no cache coherence across hosts and no atomic operations, making it representative of the non-coherent CXL Type-3 model that motivates TraCT's software coherence design (Section 3.4). In our setup, the device delivers an access latency of 640 ns and a bandwidth of 10.1 GB/s, measured using Intel's Memory Latency Checker (MLC) [2]. We configured 128 GB of the Niagara device's capacity as the shared memory region between workers.

Software. We deploy the LLM inference runtime using Dynamo v0.5.0 integrated with vLLM v0.10.1.1. Our evaluation includes three configurations: the baseline with RDMA-based NIXL/UCX [4, 35] transfer (denoted NIXL), LMCache [10], and TraCT. LMCache represents a DRAM-resident KV caching baseline, storing a 64 GB prefix cache in the prefill worker's host DRAM. On a prefix cache hit, the prefill worker loads the matching KV blocks directly from its local DRAM into GPU memory, avoiding recomputation for those blocks. Regardless of cache hit or miss, all KV blocks must be transferred from the prefill worker's GPU to the decoding worker via NIXL. TraCT uses an identically sized prefix cache placed in CXL-attached shared memory to enable pooled, load/store–accessible caching. The NIXL/UCX configuration serves as a no-prefix-reuse baseline that transfers KV tensors through NIXL over UCX without caching. To isolate the cost of KV transfer and enable a controlled comparison between CXL-based caching (TraCT) and DRAM-based caching (LMCache), prefix caching on GPU memory is disabled in all experiments. All evaluations use the DeepSeek-R1-Distill-Llama-8B model.

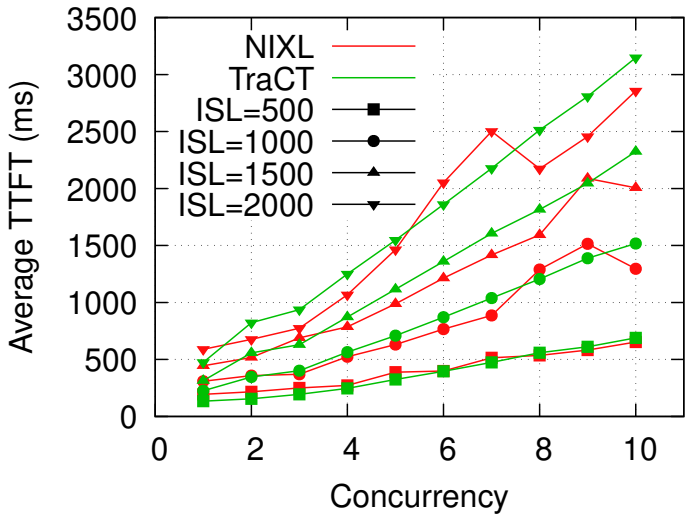


Figure 4.1: NIXL vs. TraCT (no caching).

4.2 CXL as a KV Transfer Substrate

We first evaluate CXL shared memory as a bare KV-transfer substrate by disabling the KV cache in TraCT and measuring time-to-first-token (TTFT) under varying concurrency and input sequence lengths (ISL $\in \{500, 1000, 1500, 2000\}$ tokens). The output length is fixed at 5 tokens to isolate prefill performance, since TTFT primarily reflects how quickly the system completes the prefill phase, which dominates latency for long inputs with short outputs. A lower TTFT therefore indicates faster prefill execution and KV transfer in disaggregated LLM serving. As the prefill computation is identical across configurations in these experiments, the main difference in TTFT is attributable to KV transfer time.

We compare TraCT against NIXL [4], a high-performance KV-transfer library from NVIDIA that operates over a 100 Gbps InfiniBand fabric and serves as a state-of-the-art baseline for GPU-to-GPU KV communication. Prefix caching is disabled in both systems so that prefill execution is not shortened by cache hits, yielding a fair comparison of raw transfer performance.

Figure 4.1 shows that TTFT scales roughly linearly with ISL and increases with concurrency, as expected given the compute-intensive nature of prefill. TraCT incurs moderately higher TTFT than NIXL across all configurations, primarily due to the bandwidth gap between our prototype CXL device (∼10 GB/s) and NIXL's 100 Gbps (12.5 GB/s) interconnect. However, production CXL over PCIe Gen5 ×16 is expected to provide up to ∼128 GB/s [1], indicating substantial headroom beyond our prototype. At the same time, the NIC roadmap is also advancing rapidly: while 400 Gbps has been the recent high end, 800 Gbps adapters are already emerging, and 1.6 Tbps Ethernet is actively being standardized and placed on vendor roadmaps. This means that network-based KV transfer will continue to improve as well. Even so, these trends do not undermine our conclusion. Rather, they suggest that the gap observed here is largely a consequence of current prototype generations rather than a fundamental limitation of CXL. As CXL hardware matures toward production bandwidths, our results show that it remains a promising KV-transfer substrate, with the additional advantage of avoiding the separate NIC and switch infrastructure required by network-based transfer in rack-scale deployments.

4.3 LLM Inference Performance

We evaluate end-to-end performance under a ShareGPT multi-turn conversation workload [6] with a constant 500 conversations (1,650 requests in total), while varying the number of concurrent sessions. We measure goodput as the rate of requests satisfying a service-level agreement (SLO) of TTFT $\leq$ 400 ms and inter-token latency (ITL) $\leq$ 100 ms. These thresholds reflect widely adopted targets for interactive LLM serving. For TTFT, prior work has targeted sub-500 ms response initiation to avoid perceptible lag in chat-based applications [49]. For ITL, a 100 ms threshold corresponds to roughly 10 tokens/s, comfortably above the

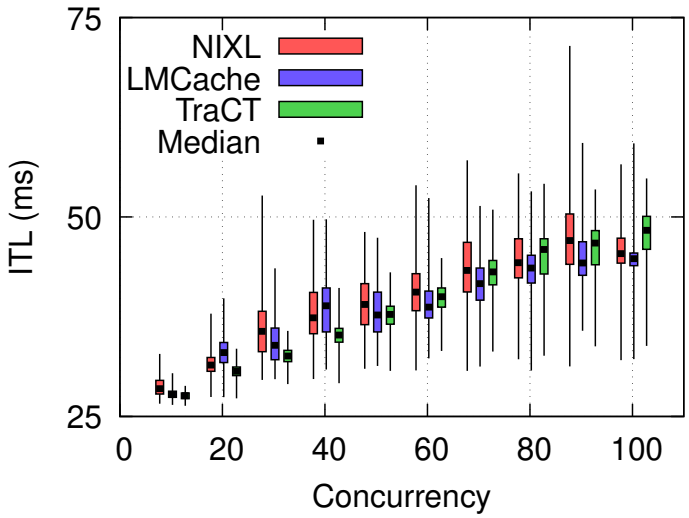


Figure 4.2: ShareGPT ITL distribution.

average human reading speed of about 4–5 tokens/s (approximately 250 words/min) [49], thereby enabling smooth token streaming without perceptible stalls. TTFT captures prefill performance including prefill computation and prefill-to-decode KV block transfer, while ITL reflects token generation latency of the decoding phase.

4.3.1 ITL

Figure 4.2 shows that ITL distributions are similar across all three systems at every concurrency level. This is because the workload is decoding-bound: the GPU is fully saturated by autoregressive token generation, and neither prefix caching nor KV transfer mechanism alters the decoding process itself. Consequently, all three systems also achieve nearly identical raw request throughput, also depicted with the dashed lines in Figure 4.4.

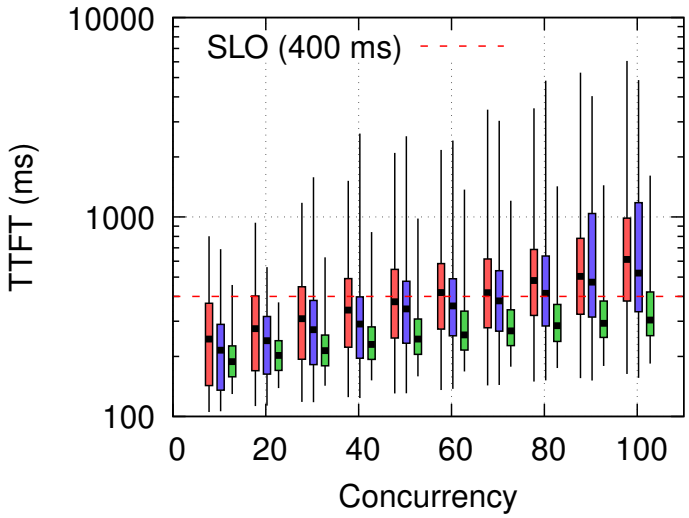


Figure 4.3: ShareGPT TTFT (log scale).

4.3.2 TTFT

Since ITL is comparable across systems, goodput is determined entirely by TTFT (Figure 4.3). NIXL's TTFT distribution sits near the 400 ms SLO threshold at low concurrency and shifts above it as sessions increase. Every KV block must be transferred to the decoding worker over the 100 Gbps NIC regardless of cache state, introducing queuing delays that grow with load. LMCache reduces TTFT at low concurrency by serving prefix cache hits from local DRAM, avoiding recomputation for matched blocks. However, all KV blocks—regardless of cache hit or miss—must still be transferred to the decoding worker via the network. As concurrency grows, this transfer overhead dominates and LMCache's TTFT converges toward NIXL's. (Details of network transfer overhead are discussed in Section 4.4.) TraCT's TTFT median remains consistently below the SLO threshold with a narrow first-to-third quartile range (box in Figure 4.3) even at peak concurrency as KV blocks are transferred directly through CXL shared memory without network overhead.

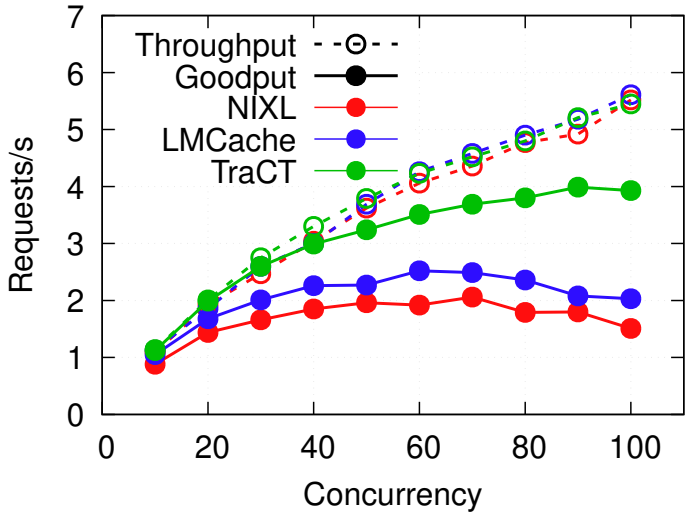


Figure 4.4: Request throughput and goodput under ShareGPT multi-turn conversation workload with 500 total conversations, varying concurrent session count. Dashed lines denote request throughput; solid lines denote goodput (requests satisfying SLO: TTFT $\leq$ 400 ms, ITL $\leq$ 100 ms.)

4.3.3 Goodput

Figure 4.4 shows that TraCT achieves the highest goodput across all concurrency levels, reaching approximately 4 req/s at 100 concurrent sessions—a 1.9× improvement over LMCache. This advantage is most pronounced under multi-turn conversation workloads, where KV blocks from the previous turn accumulate in the shared CXL cache and are reused in place by any worker without retransmission, reducing the redundant network copies that LMCache cannot avoid even on cache hits.

4.4 Performance Breakdown and GPU Utilization

To understand why TraCT achieves the best TTFT in Section 4.3, we break down the average prefill latency into three phases: (1) *Schedule*, from request submission to GPU memory

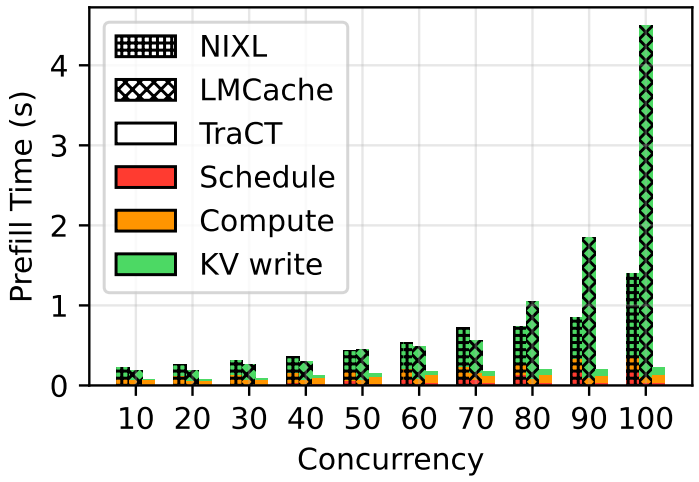


Figure 4.5: Average prefill time breakdown (Schedule, Compute, and KV write) for NIXL, LMCache, and TraCT under varying concurrency.

allocation; (2) *Compute*, spanning prefix-cache KV reads through prefill computation; and (3) *KV write*, from prefill completion until the written KV blocks become visible to the decoding worker.

Figure 4.5 shows that scheduling and compute times are small and comparable across all three systems, confirming that the bottleneck lies primarily in the KV write path. Both NIXL and LMCache exhibit sharply increasing KV write times beyond concurrency 70. For example, LMCache reaches ∼1.8 s and ∼4.6 s at concurrency 90 and 100, respectively. This behavior stems from two compounding factors. First, as concurrency increases, network bandwidth becomes saturated, causing KV transfers to queue and forcing both systems to retain KV blocks in GPU memory until the transfers complete. The resulting GPU memory pressure reduces effective resource utilization and further amplifies TTFT growth. Second, LMCache writes KV blocks to local DRAM for prefix-cache reuse simultaneously with the NIXL transfer, consuming additional PCIe bandwidth on top of the network transfer. Under high concurrency, these two data paths saturate their respective bandwidths concurrently,

causing LMCache's KV write time to exceed that of NIXL.

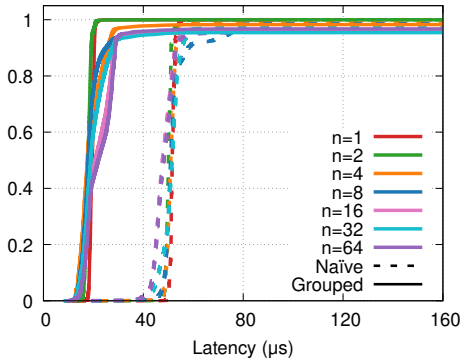
TraCT avoids both overheads by writing KV blocks directly to CXL shared memory via GPU–CXL direct DMA. This eliminates the need for an acknowledgment (ACK) from the decoding worker: the transfer completes as soon as the DMA copy finishes, immediately freeing GPU memory for subsequent requests. Moreover, TraCT writes only missed KV blocks, skipping those already resident in shared memory from prior turns. In the ShareGPT trace used in our evaluation, the prefix-cache hit rate reaches 63%, meaning that TraCT transfers less than half the KV data that NIXL or LMCache must send for the same requests, directly bounding KV write time regardless of concurrency. As shown in Figure 4.5, whereas NIXL and LMCache scale poorly with increasing concurrency, TraCT's KV write time remains consistently low across all concurrency levels, confirming that its design is well suited to high-concurrency, multi-turn workloads.

4.5 Soundness of TraCT Key Components' Design

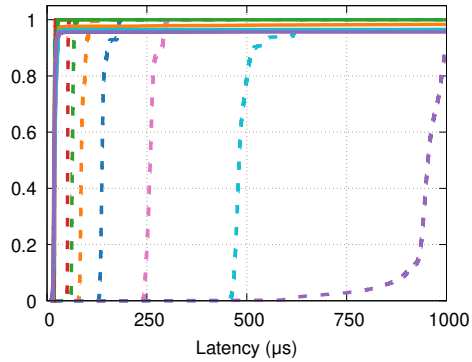
The successful deployment of TraCT and its superior performance depend on the soundness of the three key components it introduces. Although the results presented are promising, they were obtained on a limited testbed. In this section, we discuss the implications of scaling TraCT to a full-rack system.

4.5.1 Two-Tiered Lock

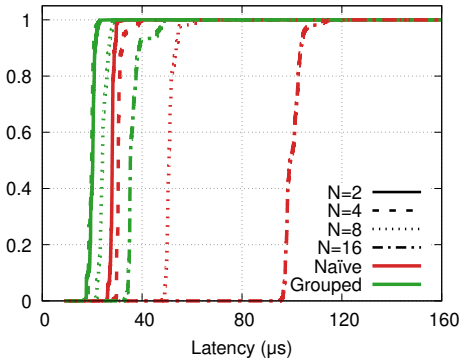
Here, we evaluate lock acquisition latency as load increases. Specifically, we evaluate the two-tiered lock design under two contention scenarios in a 2-node configuration, with up to 64 threads per node. The benchmark measures raw lock acquisition latency to isolate the



(a) 1-lock, n threads per node.



(b) n-lock, n threads per node.



(c) 1-lock, N nodes.

Figure 4.6: Lock acquisition latency CDF. Under varying thread counts (n) per node in a 2-node configuration. (a) n threads contend on a single lock. (b) n threads each contend against another node's thread on a dedicated lock (n locks total). (c) A single lock acquisition by one thread without any contention to isolate and measure the lock manager's intrinsic cacheline flush overhead induced by the lock-free CICL rule as the number of nodes N increases. (Grouped: our bitmap-based grouped design, Naïve: per-lock-struct baseline.)

cost of the lock mechanism itself, directly stressing the polling and flush overheads of the lock manager described in Section 3.3.

As a baseline, we implement a simple but naïve two-tiered lock design that allocates one lock structure per logical lock, with each structure containing N per-node state slots, each of which can be IDLE, WAITING, or GRANTED. Under this design, the manager must repeatedly poll and flush the cacheline of every slot to observe up-to-date lock state. For L locks and N nodes, this requires $O(N \cdot L)$ cacheline flushes per polling round. The space overhead is also

substantial: each lock consumes $(1+N)$ cachelines, consisting of one cacheline for metadata (storing the lock identifier, allocation status, current holder, and the round-robin counter used for fairness) and one cacheline per node for its per-node state slot. As the number of locks grows, both the metadata footprint and the polling cost increase linearly. We compare our grouped lock design against this naïve design to validate the $O(N)$ vs. $O(N \cdot L)$ flush-cost analysis and to show that the grouped design eliminates latency growth as the number of locks and threads increases.

Figure 4.6 shows the lock-acquisition latency CDFs for both designs. In the single-lock scenario (Figure 4.6a), n threads per node contend for a single shared lock (L=1). Even at L=1, the grouped design exhibits lower latency than the naïve design across all thread counts. Both designs incur $O(N)$ flushes when L=1, but the naïve manager must examine each per-node slot individually through state transitions (Idle→Waiting→Granted), whereas the grouped manager processes all slots using bitwise operations. This difference in per-round handling cost leads to higher baseline latency in the naïve design even when only a single lock is present.

The scalability gap becomes much more pronounced in the n-lock scenario (Figure 4.6b), which directly exercises the $O(N \cdot L)$ cost predicted by the design analysis. Each node runs n threads, where thread i on one node forms a pair with thread i on the other node, and each pair contends for its own dedicated lock (n locks total, L=n). The naïve design exhibits latency that grows linearly with n: at n=64, the tail exceeds 750 μs, confirming the $O(N \cdot L) = O(N \cdot n)$ cost model. In contrast, the grouped design maintains stable latency well below 100 μs across all thread counts, validating that its bitmap-based manager reduces the flush cost to $O(N)$ regardless of L.

4.5.2 Cache Coherence

Although TraCT's two-tiered lock successfully reduces the flush overhead from $O(N \cdot L)$ to $O(N)$, the number of nodes N itself remains a scalability concern. Modern AI inference racks are power-constrained: a single rack-scale system such as the NVIDIA GB200 NVL72 houses 18 compute nodes at up to 120 kW [28, 29, 30]. The two-tiered lock must therefore provide proper scalability as N grows within this rack-scale range.

Since our physical testbed consists of only two server nodes, we simulate larger rack-scale configurations by creating N logical nodes. This faithfully models the lock manager's $O(N)$ search space, as the manager must flush one cacheline pair per node per polling iteration regardless of whether the nodes are physical or logical. Note that this differs from Figures 4.6a and 4.6b: there, additional threads are filtered by the *local_lock* before reaching the global manager, so thread count does not affect its polling overhead. However, increasing N directly expands the search space the manager must flush per iteration.

Figure 4.6c shows lock acquisition latency as N increases from 2 to 16 nodes, with only one thread in total acquiring a lock in isolation to measure the lock manager's overhead. The remaining $O(N)$ cost is attributable to the cacheline flush overhead imposed by the lock-free CICL rule: upon each lock management iteration, the manager flushes one cacheline pair per logical node to ensure cross-node visibility of lock state updates. The naïve design exposes this growth directly, with average latency increasing from $\sim$20 μs at N=2 to approximately 100 μs at N=16. The grouped design, by contrast, maintains stable latency well below 40 μs across all tested node counts by accelerating the procedure using bitwise operations and minimizing the number of cachelines that must be flushed per polling iteration. This shows that TraCT's two-tiered lock scales to rack-size deployments even as N approaches the upper bound of a production AI inference rack.

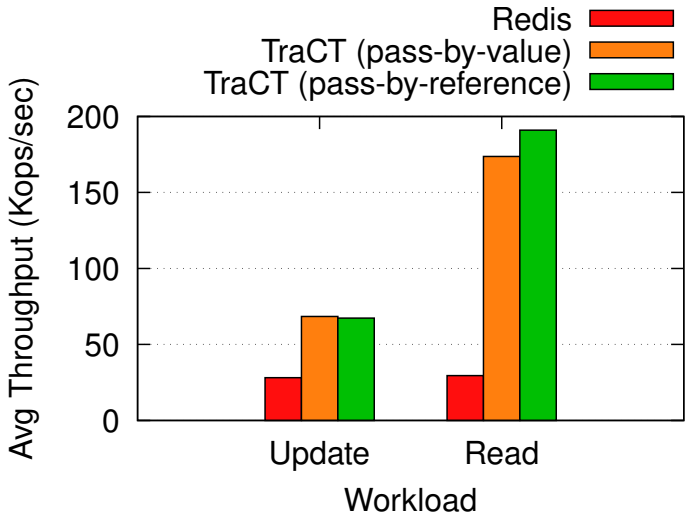


Figure 4.7: YCSB benchmark comparing TraCT's shared object directory against Redis in-memory database.

4.6 Shared Object Directory

We now evaluate our approach to shared object-directory management. As a baseline, we use Redis, a widely deployed in-memory key-value store commonly used for distributed metadata management and cache coordination in serving systems [34]. In disaggregated LLM serving, Redis can naturally serve as a shared metadata store for tracking the locations of reusable KV blocks across workers; notably, LMCache also supports Redis as a storage backend [25]. We compare three configurations: Redis; TraCT with pass-by-value, which copies the value from CXL memory on each read; and the default TraCT, which uses pass-by-reference, i.e., returns a pointer to the value directly without copying. This comparison isolates two orthogonal benefits: (1) the gain from eliminating the network round trip, captured by comparing Redis with TraCT using pass-by-value, and (2) the additional gain from eliminating the data copy, captured by comparing TraCT using pass-by-value and pass-by-reference.

Figure 4.7 reports throughput using the YCSB benchmark [11] with 5K records, 10M operations, and an 8 MB record size matching the KV-block size in our environment. Read

performance is evaluated using YCSB-C (100% reads), while update performance is evaluated using a 100% update workload. The Redis baseline runs a single locally deployed `redis-server` instance with its default configuration.

For updates, TraCT achieves ∼2.4× higher throughput than Redis, showing that bypassing the network stack alone provides a substantial benefit even when pass-by-value is used. Pass-by-reference offers no additional advantage for updates, since updates necessarily require copying the full object into CXL shared memory. For reads, which are more important in practice as prefix-cache lookups occur on every incoming request, pass-by-reference provides an additional gain, achieving ∼191 Kops/s, about 10% higher than pass-by-value and 6.5× higher than Redis. The gap between the two TraCT configurations reflects the cost of pass-by-value, which copies an 8 MB KV block out of CXL memory on every read. Pass-by-reference eliminates this cost entirely by returning a direct pointer into the shared CXL region.

Chapter 5

Related Work

5.1 Disaggregated LLM Inference

As LLM deployments scale out across larger GPU clusters and serve higher request rates, the resource inefficiency of colocating prefill and decode on the same GPU becomes a primary challenge. Orca [46] replaced this with iteration-level scheduling—also known as continuous batching—which allows finished requests to vacate their slots immediately and new requests to join without waiting, substantially improving GPU utilization as request arrival rates grow.

Even with continuous batching, however, the prefill and decode phases continue to interfere when colocated on the same GPU. A long prefill from one request stalls the ongoing decode iterations of all other requests in the batch, inflating inter-token latency. Sarathi-Serve [7] addresses this through chunked prefill: a long prompt is split into fixed-size chunks, each small enough to be interleaved with decode iterations without causing a generation stall. While chunked prefill reduces the severity of prefill–decode interference within a single GPU, it does not eliminate it, because the two phases ultimately compete for the same compute and memory resources. Moreover, as clusters grow to dozens or hundreds of GPUs, the optimal resource allocation for prefill and decode diverges: prefill benefits from tensor parallelism and large compute batches, while decode benefits from replication and low-latency scheduling. Coupling the two phases on the same set of GPUs prevents implementing these tailored

strategies independently.

Disaggregated serving resolves this by assigning prefill and decode to separate worker pools, each provisioned and scaled independently. Splitwise [32] generalizes this design to both homogeneous and heterogeneous deployments and introduces layer-wise KV-cache transfer to overlap communication with computation. DistServe [49] develops the prefill/decode split as a full system architecture, co-designing request scheduling and resource provisioning across the two worker types. It shows that the optimal prefill-to-decode GPU ratio depends on request length distribution and arrival rate, and that disaggregation significantly improves per-GPU goodput under SLO constraints. Preble [36] focuses on the scheduling layer, dynamically routing requests to prefill and decode instances based on KV-cache reuse opportunities and real-time workload variation. By steering requests that share a common prefix to the same prefill worker, Preble reduces redundant KV recomputation without modifying the underlying transfer mechanism. NVIDIA's Dynamo [3] brings these ideas into a production-oriented framework, offering a general substrate for prefill/decode disaggregation together with a KV block manager (KVBM) that coordinates KV movement and caching across workers at scale.

A common assumption across these systems is that KV transfer remains a network operation. In practice, they typically move KV blocks over RDMA using communication substrates such as UCX and NIXL [4, 35]. Although pipelining and compute–communication overlap can partially hide this overhead, the network path remains on the critical path of KV movement. TraCT differs by replacing this network transfer path with CXL shared memory, allowing decoding workers to fetch KV blocks through a shared-memory substrate rather than through host-to-host communication.

5.2 Multi-tier KV Caching

As LLMs are increasingly deployed for long-context and multi-turn workloads, the KV cache for a single request can reach several gigabytes, quickly exhausting GPU memory under concurrent load. Early inference systems managed KV tensors as contiguous, request-length buffers, which led to fragmentation and prevented sharing across requests.

PagedAttention [21] addressed this by managing KV tensors as fixed-size pages, borrowing the virtual memory abstraction from operating systems. This eliminated fragmentation and enabled copy-on-write sharing of KV blocks between requests that diverge from a common prefix. SGLang's RadixAttention [48] extended prefix reuse further by organizing cached KV blocks in a radix tree, allowing requests with shared prefixes—such as repeated system prompts or few-shot examples—to reuse the corresponding blocks without recomputation. Both techniques operate entirely within a single GPU and do not address the problem of sharing KV state across nodes.

As models and context lengths grew, researchers extended KV storage beyond the GPU to form multi-tier hierarchies. LMCache [10] offloads KV blocks to a hierarchy of backends ranging from host DRAM to distributed object stores, enabling reuse across serving engines and across requests that share only partial prefixes. CacheBlend [44] extends this further by relaxing the requirement that reused blocks form a strict prefix, allowing non-contiguous reuse and blending cached blocks with newly computed ones. Mooncake [33] takes a cluster-wide view, constructing a global KV pool that spans CPU memory and SSDs across separate prefill and decode clusters, and integrating it with the serving framework to exploit reuse across users and workloads.

These systems demonstrate that KV state is worth sharing broadly and that a tiered storage hierarchy can meaningfully reduce redundant computation. However, they treat the KV

cache as residing behind a network or storage stack: regardless of whether a requested block is a cache hit, it must still be transmitted to the decoding worker over the network. This keeps the network on the critical path even under high reuse, and creates a structural separation between the KV transfer mechanism and the KV caching mechanism that neither system alone can resolve. TraCT eliminates this separation by placing the KV cache directly in CXL shared memory, which is accessible to all participating nodes via DMA without network involvement. The same shared-memory substrate thus serves as both the KV transfer channel and the prefix-aware KV cache, unifying what prior work treats as two distinct mechanisms into a single datapath.

5.3 CXL-based Memory Systems and Shared Memory

The idea of treating remote memory as part of a shared address space predates CXL. Remote direct memory access (RDMA) allows one host to read and write another host's DRAM without involving the remote CPU, and has been used to build distributed key-value stores [20, 27], distributed shared memory [13], and remote memory paging systems [15]. These systems achieve low-latency remote access but still traverse the NIC and network fabric on every operation, incurring serialization overhead, NIC buffer copies, and sensitivity to network congestion. CXL Type-3 shared memory occupies a different point in the design space: hosts access the same physical device memory through the normal load/store interface and DMA, without any network involvement, while the device presents a single coherent physical address space to all attached nodes.

A growing body of work explores how to build systems on top of this substrate. CXL-SHM presents a general shared-memory substrate over CXL and demonstrates its use for distributed data structures and RDMA offload, but assumes device-side compare-and-swap

(CAS) support [47]. Tigon studies an in-memory database over CXL shared memory and characterizes the limits of coherence support, showing that any hardware coherence, when present, is confined to small device-specific regions rather than the full device capacity [17]. cMPI replaces the traditional network path in MPI communication with CXL shared memory [40]. Beluga explores KV cache management over CXL shared memory and routes control-plane metadata operations through a dedicated metadata server [43]. OASIS uses CXL shared memory as a software datapath for pooling PCIe devices such as NICs and SSDs across hosts, thereby avoiding expensive PCIe switch hardware [51]. Suetterlein et al. analyze the overhead of software synchronization and the limits of software-based coherence on CXL Type-3 hardware, demonstrating these constraints using an enhanced Peterson lock [37].

Together, these systems establish CXL shared memory as a practical substrate for cross-host communication and data sharing. However, their synchronization assumptions differ from those of TraCT: CXL-SHM and Tigon rely on hardware support—CAS or device-specific coherent regions—that current Type-3 devices do not provide across nodes. cMPI and Beluga avoid fine-grained mutual exclusion altogether, either through queue-based designs or by delegating coordination to a centralized server. TraCT takes a different approach: it targets non-coherent CXL memory without assuming cross-host atomics, full-device coherence, or centralized coordination, and builds all required synchronization and object-management primitives entirely in software. This allows TraCT to operate correctly on commodity CXL Type-3 hardware while using the shared memory directly as a rack-scale KV substrate.

Chapter 6

Conclusion

Disaggregated LLM serving improves GPU utilization and throughput, but transfers the KV cache onto the critical path: every request requires moving hundreds of megabytes of KV tensors between prefill and decode workers, and today's systems have no choice but to route this traffic through the NIC and network fabric. This thesis asked whether CXL shared memory could replace this network hop entirely.

TraCT answers this question affirmatively. By using CXL shared memory as both a network-free KV-transfer substrate and a rack-scale prefix-aware KV cache, TraCT allows prefill and decoding workers to exchange KV blocks through direct GPU–CXL DMA, without involving the NIC, network switch, or host DRAM copy path. Realizing this design on today's non-coherent CXL Type-3 devices required solving three fundamental software challenges. First, we introduced a two-tiered inter-node lock that provides scalable mutual exclusion without hardware atomic operations, reducing lock metadata overhead by 256× relative to a naïve design while maintaining stable acquisition latency up to rack scale. Second, we formalized a software coherence discipline—the CICL, FP, and XW conditions—that guarantees correct data visibility on non-coherent shared memory. Third, we introduced a shared object directory with offset-based references, enabling pointer-based data structures to be created once in CXL memory and safely traversed by any node despite differences in virtual address mappings. Together, these mechanisms form a general-purpose CXL shared-memory library that is application-agnostic and reusable beyond LLM serving.

Evaluated on real CXL hardware against RDMA-based baselines under a realistic multi-turn conversational workload, TraCT reduces average time-to-first-token by up to 2.6×, lowers P99 TTFT by up to 3.0×, and improves peak goodput by up to 1.9×.

6.1 Limitations

Despite these results, TraCT has several limitations that bound the scope of the current work.

Two-node testbed. Our physical evaluation is limited to a two-server configuration, which falls short of the scale needed to fully characterize rack-level behavior. Although Section 4.5 uses logical node simulation to evaluate the two-tiered lock up to N=16 nodes, the end-to-end serving experiments in Sections 4.2–4.4 reflect only a two-node disaggregated setting. Modern AI inference racks operate at a substantially larger scale: for example, the NVIDIA DGX GB200 NVL72 houses 18 compute nodes and 72 GPUs within a single rack [28, 29, 30]. In a fully disaggregated deployment at this scale, tens of prefill and decode workers would concurrently access the shared CXL device, generating far greater aggregate KV volumes and lock contention than our two-node setup exercises. Whether TraCT's performance advantages hold under such conditions, where CXL bandwidth is shared among many concurrent DMA streams and the lock manager must track a larger number of active nodes, remains to be validated on larger hardware.

No fault tolerance. TraCT currently provides no mechanism for recovering from node failures. If a node crashes while holding the global lock, the lock manager will never observe a release, and all subsequent lock acquisitions will stall indefinitely. Allocated CXL memory chunks belonging to the crashed node also become unreachable, causing a gradual capacity leak. The problem is more complex when the crashed node is the lock manager itself: because

no further grants can be issued, any recovery procedure that depends on acquiring a lock is fundamentally unsafe, and the entire locking service becomes unavailable until the manager is manually restarted. Addressing these failure modes requires a fault-tolerance mechanism that can detect crashes and reclaim their state without relying on the lock infrastructure that may itself be compromised.

6.2 Future Work

The limitations above, together with the broader design space that TraCT opens, point to concrete directions for future work.

Fault tolerance and crash recovery. A production-ready deployment of TraCT must tolerate node failures without stalling other workers. The core challenge is that standard recovery strategies, such as acquiring a lock to serialize reclamation, are unsafe when the failed node may be the lock manager itself. TraCT therefore calls for a lock-free recovery design. One viable approach is a ring-based failure detection model in which responsibility for detecting and reclaiming a failed node is statically assigned to exactly one peer: its designated successor in a logical ring ordered by rack-local node indices. This single-watcher discipline ensures that no two live nodes can race to reclaim the same resources, with uniqueness guaranteed structurally rather than through an additional synchronization protocol. Each node periodically increments a heartbeat counter stored in its own CICL-compliant slot in CXL shared memory; its designated successor monitors this counter and, if it remains unchanged beyond a configurable timeout, declares the node failed. Reclamation then proceeds lock-free in three steps: resetting the failed node's lock slots, returning its allocated memory chunks to the shared allocator bitmap, and clearing its heartbeat slot to prevent duplicate detection. If the failed node was also the lock manager, the watcher additionally elects itself by

writing its identifier into the well-known manager slot in CXL shared memory, resuming lock service without state reconstruction since all manager state already resides in shared CXL memory. Implementing and evaluating this design under realistic failure scenarios, including concurrent failures and lock-manager crashes, is a concrete direction for future work.

Multi-tiered KV cache. TraCT currently stores all KV blocks in a single CXL shared memory tier. As the active KV working set grows beyond the capacity of the CXL device, a natural extension is to introduce additional storage tiers. Host-local DRAM, NVMe SSDs attached to CXL via device pooling [51], or even HDDs could serve as overflow tiers for less recently accessed KV blocks. In such a design, CXL shared memory would serve two complementary roles: a *consistent shared metadata index*, where the prefix cache index and all inter-node coordination state remain in CXL memory regardless of where the KV payload resides, so any node can locate, reference-count, migrate, and evict any block through a single coherent control plane without network round trips; and a *high-bandwidth staging layer* for hot KV blocks likely to be reused in the near term, providing direct GPU DMA access at CXL latency while colder blocks are demoted to slower tiers. Prior work such as LMCache [10] and Mooncake [33] has demonstrated the value of multi-tier KV hierarchies in network-based settings; the key distinction here is that CXL shared memory enables the metadata index to remain globally consistent across all nodes without any network involvement, eliminating the coordination overhead that network-based tiered caches incur on every cache hit and miss. Designing the tiering policy, eviction across tier boundaries, and the promotion and demotion strategy to hide the latency gap between CXL and SSD are interesting directions for future work.

Looking ahead, the broader implication of this work is that the boundary between inter-node communication and shared memory is shifting. As CXL hardware matures toward production bandwidths and rack-scale deployments, systems that today rely on network-

based coordination protocols will have the opportunity to replace those protocols with direct shared-memory access—provided the software can meet the correctness requirements that hardware no longer guarantees. TraCT demonstrates that this is achievable today, on real hardware, and that the performance gains are substantial. The synchronization primitives, coherence conditions, and object-management abstractions introduced in this thesis offer a starting point for building that next generation of rack-scale systems.

Bibliography

[1] CXL specification. https://computeexpresslink.org/cxl-specification/.

[2] Intel® Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[3] NVIDIA Dynamo. https://github.com/ai-dynamo/dynamo.

[4] NVIDIA Inference Xfer Library (NIXL).

[5] vLLM. https://github.com/vllm-project/vllm.

[6] ShareGPT. https://sharegpt.com, 2023.

[7] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[8] Milind Chabbi, Abdelhalim Amer, Shasha Wen, and Xu Liu. An efficient abortable-locking protocol for multi-level numa systems. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 61–74, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level numa systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, page 215–226, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference, 2025.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[12] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, pages 247–256, 2012.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Google Gemini Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.

[15] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[16] Jaewan Hong, Marcos K. Aguilera, Emmanuel Amaro, Vincent Liu, Aurojit Panda, and Ion Stoica. The dawn of disaggregation and the coherence conundrum: A call for federated coherence, 2025.

[17] Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. Tigon: a distributed database for a cxl pod. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[18] Intel Corporation. *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*. Intel Corporation, 2023. Document Number 319433-053.

[19] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. Memory sharing with cxl: Hardware and software design approaches, 2024.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 295–306, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[22] 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. 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 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered memory management beyond hotness. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[24] AI @ Meta Llama Team. The llama 3 herd of models, 2024.

[25] LMCache. Redis backend — LMCache documentation. https://docs.lmcache.ai/kv_cache/redis.html, 2025.

[26] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 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 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[28] NVIDIA. DGX GB200 NVL72: Rack-scale AI system. https://developer.nvidia.com/blog/nvidia-gb200-nvl72-delivers-trillion-parameter-llm-training-and-real-time- 2024.

[29] NVIDIA. NVIDIA GB200 NVL72. https://www.nvidia.com/en-us/data-center/gb200-nvl72/, 2024.

[30] NVIDIA Corporation. Key components of the DGX SuperPOD. https://docs.nvidia.com/dgx-superpod/

reference-architecture-scalable-infrastructure-gb200/latest/dgx-superpod-components.html, 2025. Accessed: 2025.

[31] OpenAI. Chatgpt (gpt-5). https://openai.com/chatgpt.

[32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132, 2024.

[33] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 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)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[34] Redis Ltd. Redis. https://redis.io, 2024.

[35] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. Ucx: an open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[36] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. In *The Thirteenth International Conference on Learning Representations*, 2025.

[37] Joshua Suetterlein, Joseph Manzano, and Andres Marquez. Synchronization for cxl based memory. In *Proceedings of the International Symposium on Memory Systems*,

MEMSYS '24, page 178–185, New York, NY, USA, 2024. Association for Computing Machinery.

[38] The Linux Kernel Documentation. DAX Devices — The Linux Kernel Documentation. https://docs.kernel.org/driver-api/cxl/allocation/dax.html, 2024.

[39] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 79–94, New York, NY, USA, 2024. Association for Computing Machinery.

[40] Xi Wang, Bin Ma, Jongryool Kim, Byungil Koh, Hoshik Kim, and Dong Li. cmpi: Using cxl memory sharing for mpi one-sided and two-sided inter-node communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '25, page 2216–2232, New York, NY, USA, 2025. Association for Computing Machinery.

[41] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-Exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, Santa Clara, CA, July 2024. USENIX Association.

[42] Tong Xing and Antonio Barbalace. Rethinking applications' address space with cxl shared memory pools. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems*, HCDS '25, page 52–59, New York, NY, USA, 2025. Association for Computing Machinery.

[43] Xinjun Yang, Qingda Hu, Junru Li, Feifei Li, Yicong Zhu, Yuqi Zhou, Qiuru Lin, Jian Dai, Yang Kong, Jiayu Zhang, Guoqiang Xu, and Qiang Liu. Beluga: A cxl-based memory architecture for scalable and efficient llm kvcache management, 2025.

[44] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, page 94–109, 2025.

[45] Dongha Yoon, Younghoon Min, Hoshik Kim, Sam H. Noh, and Jongryool Kim. Tract: Disaggregated llm serving with cxl shared memory kv cache at rack-scale, 2025.

[46] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[47] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 658–674, New York, NY, USA, 2023. Association for Computing Machinery.

[48] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc.

[49] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association.

[50] 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. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024. USENIX Association.

[51] Yuhong Zhong, Daniel S. Berger, Pantea Zardoshti, Enrique Saurez, Jacob Nelson, Antonis Psistakis, Joshua Fried, and Asaf Cidon. My cxl pool obviates your pcie switch. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 58–66, New York, NY, USA, 2025. Association for Computing Machinery.