Perceptor: Cross-Layer Adaptive Tuning Engine for LLM Inference

Lei Li, Yubo Li, Zhipeng Shen, Anning Cai, Shuaiyi Zhang, Tao Yang, Chaoqun He, Yang Chu, Bing Liu
Lenovo
Beijing, China
{lilei16, liyb17, shenzp1, caian2, zhangsy28, yangtao29, hecq4, chuyang2, liubing25}@lenovo.com

***Abstract*—On-site delivery and tuning of Large Language Model (LLM) inference face several challenges, like diverse environments, complex tuning parameters, and unpredictable workloads. We propose Perceptor, an end-to-end adaptive tuning engine to overcome these issues. Based on a perceive-adaption loop, Perceptor achieves lifetime-span adaptive performance tuning across various environments. It introduces a multi-stage optimization pipeline decoupling cross-layer key tuning parameters, and pioneers a collaborative strategy that combines pre-deployment (static) and post-deployment (online) optimizations to always keep optimal inference performance. Specifically in optimization pipeline, we have proposed novel adaptive optimization algorithms for parallel strategy, RoCE networks, and Prefill-Decode (PD) workload scheduling. Experiments demonstrate significant performance improvements achieved by Perceptor across various scenarios.**

***Index Terms*—LLM inference, performance tuning, on-site delivery, RoCE networks, PD disaggregation**

I. Introduction

Recent years witnessed great LLM innovations. Following the Scaling Law [1], LLM parameters increased from billions to trillions. However, typical GPU memory remains at few hundreds of gigabytes, so that single GPU or node cannot accommodate massive LLM parameters. Distributed inference across multiple GPUs or nodes, especially PD-disaggregated deployment [2], introduces high complexity in deployment and performance tuning. On the other hand, Transformers [3] have dominated LLM architecture, spawning numerous optimizations across different software stacks. The interaction between these optimizations makes end-to-end tuning more challenging, particularly in diverse customer delivery cases.

Some existing works have addressed LLM inference tuning on advanced engines like vLLM [4] and SGLang [5]. KAITO [6] simplifies parameter management across deployment scenarios. SCOOT [7] uses Bayesian optimization for automatic parameter tuning on Service Level Objective (SLO) constraints. Oltunes [8] applies online learning for joint resource and parameter optimization. Dynamo [9] profiles PD-disaggregated deployment for optimal resource allocation and parallelism. However, existing solutions incompletely solve core tuning challenges in diverse delivery cases, including:

- Existing solutions focus mainly on inference engine parameter tuning, and some extend to resource allocation co-optimization. Other tunable stack layers like OS parameters, networking, and workload dispatching remain unexplored. Cross-layer co-optimization could yield performance gains, but tuning with existing works proves nearly impossible due to enormous search space.
- Existing solutions rely on on-site expert tuning (KAITO), pre-profiling (SCOOT, Dynamo), or continuous online learning (Oltunes). On-site manual tuning or pre-profiling increases delivery time and cost. Online learning introduces overhead in customer environments and may fail T0 performance acceptance due to slow warmup.
- Existing solutions focus on pre-deployment tuning based on request statistics. However, actual request patterns often deviate from assumptions, and inference services face interference from co-located jobs. The optimization frequently fails to achieve expected performance.

In response to these challenges, we propose *Perceptor*, an adaptive, cross-layer end-to-end inference tuning engine for in-production delivery. Perceptor achieves breakthroughs in:

- *Adaptive cross-layer joint tuning.* A tuning pipeline decouples complicated cross-layer optimization into independent stages. Each stage identifies and tunes key parameters according to environment and target scenario.
- *Simple, practical algorithms.* Policy-based algorithms with environment-agnostic pre-profiling replace complicated, time-consuming approaches. We propose innovative algorithms for parallelism, RoCE networks, and PD workload scheduling.
- *Entire lifetime tuning.* Perceptor combines static pre-deployment and dynamic post-deployment (online) tuning. To our knowledge, Perceptor is the first production-level tuning engine leveraging entire inference lifetime tuning with promising dynamic algorithms for live network and compute workload adaption.

The paper is organized as follows. Section II introduces Perceptor's optimization pipeline and system implementation. Section III details key optimization aspects: parallelism, RoCE networks, and PD workload scheduling. Section IV presents experimental results. Finally, Section V concludes the paper.

II. System Design

The key designs of Perceptor are environment perception and end-to-end adaptive optimization, namely, *perceive-adaption loop*. It collects versatile static environment and dynamic workload information with built-in probes. Combining

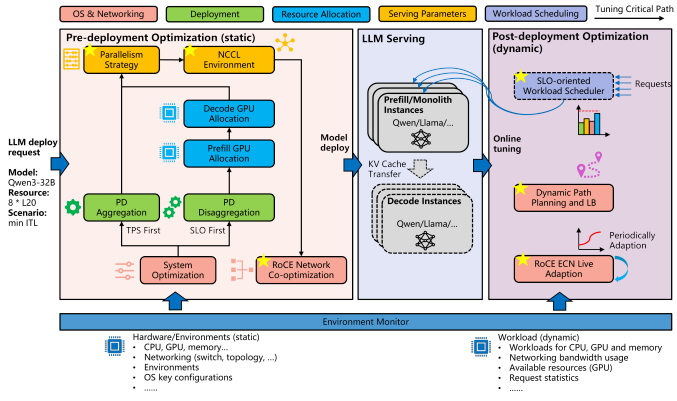

Fig. 1. Design diagram of the end-to-end inference optimization in Perceptor. The dashed box is applied to PD-disaggregated deployment only. Stars indicate highlighted optimization algorithms we proposed.

with deployment requests (target model, GPUs, user scenario), Perceptor gets LLM deployed and tuned with "pre-deployment optimization → model deployment → post-deployment optimization ↔ workload changes" workflow shown in Fig. 1.

A. Pre-deployment Optimization

In pre-deployment optimization, Perceptor accepts deployment requests and generates optimization setups (configurations, serving parameters, environment variables, etc) based on perceived environment information. Following the pipeline in Fig. 1, main optimization stages include:

System optimization

Default OS configurations are optimized for general purposes, which mismatch heterogeneous computing goals requiring deep orchestration among CPU, GPU, memory and networking. Several critical configurations can be safely tuned to improve inference performance: prohibiting NUMA balance for memory page migration avoidance [10]; disabling swap partition for memory swapping overhead elimination; turning GPU to high-performance mode; enabling CPU binding to NIC interrupts; activating NIC queue compression, etc.

PD aggregation/disaggregation

PD-aggregated deployment [4], [5] runs prefill and decode stages in the same inference instance, maximizing GPU memory usage through shared weights and Key-Value (KV) cache. However, aggregation causes interference between compute-intensive prefill and bandwidth-intensive decode, complicating latency balancing. PD-disaggregated deployment splits PD instances and resources, decoupling latency interference so that Time-To-First-Token (TTFT) in prefill and Inter-Token Latency (ITL) in decode can be tuned independently. Research shows PD aggregation maximizes throughput, while PD disaggregation better suits latency SLO-constrained scenarios [2].

Perceptor determines PD deployment strategy by user scenario: PD disaggregation for SLO-constrained cases, whereas PD aggregation for overall performance optimization. Since PD disaggregation at least doubles GPU memory, total GPU memory must accommodate this requirement.

PD resource allocation

In PD disaggregation, GPU resource allocation between prefill and decode instances critically affects inference performance. Unbalanced allocation causes request queuing at bottleneck stages, degrading overall performance. Dynamo [9] determines optimal PD resources and parallel setups through on-site profiling, prolonging delivery time and requiring re-profiling for different LLMs. Extended works [2], [11], [12] employ performance simulators to shorten on-site profiling time, but these remain model-, framework-, and hardware-specific, hardly adapting to diverse customer requirements.

Inspired by TetriInfer [13], Perceptor adopts a simple "*Decode First*" (DF) strategy: allocate minimum GPUs required by TTFT SLO for prefill first before all remaining to decode. This offers three advantages over alternatives: (1) Prefill TTFT depends on more predictable statistical input length and concurrency than uncertain output length, so that prefill resource requirement is deterministic as prefill GPU compute can be easily saturated without sophisticated parallel setups. (2) Private delivery typically requires resource-constrained deployment with pre-planned GPU counts. Reserving more GPUs for decode potentially supports higher concurrency through increased memory. (3) For bursty prefill requests under DF, locally prefill is proposed to dispatch overloaded prefill workload to decode instance (detailed in Section III-B).

Parallelism optimization

Parallelism critically affects multi-GPU inference performance. In Perceptor, a simple yet effective optimal parallel prediction algorithm is proposed based on model-agnostic profiling (detailed in Section III-A).

RoCE-NCCL co-optimization

NCCL upon RoCE networks builds high-performance data exchange channels for distributed inference. Existing works [14], [15] optimize NCCL and RoCE separately, yet their interdependent parameters require collaborative optimization. Perceptor proposes RoCE-NCCL co-optimization based on inference bandwidth prediction and link congestion modeling (detailed in Section III-B).

Principle of optimization pipelining

The pre-deployment optimization pipeline decouples interference by "*general to specific*" and "*macro to micro*" principles: common system optimization first, then PD-aggregated or -disaggregated deployment determined by user scenario, followed by coarse-to-fine optimizations, as deploy resources (resource allocation) → inference instances (parallelism) → per-instance parameters (NCCL and RoCE networks).

B. Post-deployment Optimization

The inference enters online serving phase after service launched with pre-deployment optimizations. It is difficult to keep inference working consistently on expected optimal state due to dynamic changing of workloads. The tidal characteristic of arriving requests calls for dynamic compute and bandwidth requirements, and newly coming job could also interfere through shared link bandwidth. Perceptor pioneers three algorithms for online inference tuning, addressing dynamic bandwidth and GPU compute adaption, respectively.

Dynamic path planning and load balancing

Request fluctuation and interfered network traffic cause dynamic mismatches between required and available link bandwidth. Perceptor triggers path re-planning based on link congestion monitoring, and continuously migrate traffic to idle paths to balance network workload across all available links (detailed in Section III-B).

RoCE ECN live adaption

RoCE Explicit Congestion Notification (ECN) directly impacts NCCL bandwidth utilization and inference latency. Perceptor monitors live bandwidth utilization, and dynamically changes ECN watermark according to monitored bandwidth and preset NCCL configurations (detailed in Section III-B).

SLO-oriented PD workload scheduling

For PD-disaggregated deployment, TTFT is limited by total sequence length processed on prefill instance, while ITL is determined by activated KV cache size and concurrent requests on decode instance [9]. Perceptor implements a PD dynamic workload scheduler, prioritizing requests to fill up PD instances under SLO constraints. In addition, a locally prefill strategy is exploited to migrate prefill overload to the decode instance. Unlike past works [2], [9] achieving strict SLO constraints with scalable resource, our proposal maximizes SLO-meeting request ratio (*goodput*) with fixed resource (detailed in Section III-C).

C. System Implementation

Perceptor comprises cloud-native microservices on Kubernetes following a master-workers architecture. The master handles monitored metrics accumulation, algorithms implementation and pipelining, RoCE switch configuration, and API serving for deployment management. Workers run on each node, handling metrics collection, optimization execution, and serving lifecycle management. Workers integrate vLLM and SGLang as backends for PD-aggregated, and Dynamo for PD-disaggregated cases. Additionally, Perceptor customizes Dynamo's router to support PD dynamic workload scheduling.

III. Highlighted Inference Optimizations

This section details highlighted adaptive tuning algorithms: parallelism, RoCE network, and PD workload scheduling optimizations.

A. Parallelism Optimization

Parallel strategy dramatically affects multi-GPU inference performance. Tensor Parallel (TP) reduces inference latency on penalty of high communication overhead, which is widely used for intra-node model sharding. Pipeline Parallel (PP) improves throughput with low communication cost at higher latency expense, which is often used for inter-node parallelism. Data Parallel (DP) replicates inference instances, achieving linear throughput scaling out but suffering inefficient GPU memory usage. Optimal LLM inference parallelism for a target user scenario, like maximizing throughput or minimizing latency, deeply couples with model, GPU, and networking, which is difficult to determine in advance.

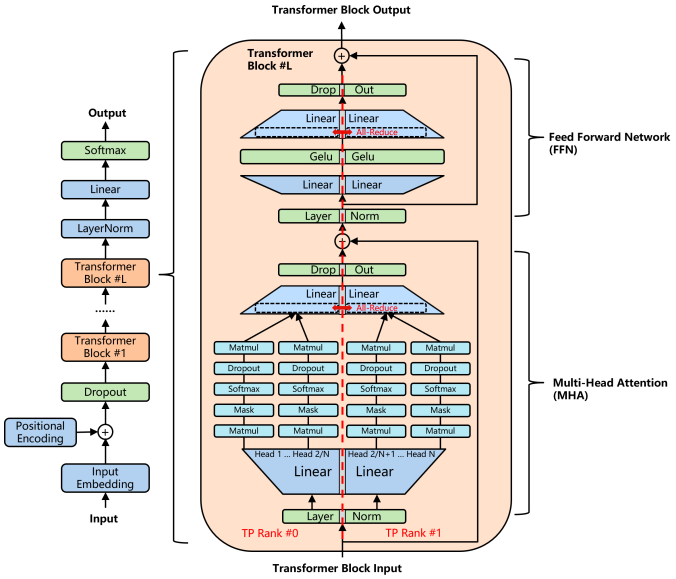

Fig. 2. TP sharding illustration of transformer-based LLMs.

Fortunately, state-of-the-art LLMs predominantly use transformer architecture [3] with similar computing patterns. Consider intra-node case first. Fig. 2 illustrates typical transformer-based LLM forwarding with TP sharding. In both Multi-Head Attention (MHA) and Feed-Forward Network (FFN) stages, TP equally distributes computation across all participants (ranks), invoking an additional All-Reduce operator to aggregate local results on each TP rank together.

Generally, TP introduces performance loss through two aspects: *All-Reduce communication overhead* for local data aggregation, and *General Matrix Multiplication (GEMM) inefficiency* for smaller data chunks computation after tensor sharding. We have the conclusion as below.

Conclusion: The TP performance of transformer-based LLM is dominated by: (1) The width of last projection matrix in MHA and FFN which affects All-Reduce overhead. (2) The maximum matrix width in TP sharding dimension involved in MHA and FFN compute which causes GEMM inefficiency.

Extensive derivation and analysis can be found in [16], key hyper-parameters dominating TP performance include:

- MHA: hidden_size, attention_head_size
- FFN: hidden_size, intermediate_size

where we define

$$\begin{aligned}\text{attention\_head\_size} = (&\text{num\_attention\_heads} \\ &+ 2 \times \text{num\_key\_value\_heads}) \times \text{head\_dim}\end{aligned} \tag{1}$$

and referred hyper-parameter names are from HuggingFace's model definition (config.json). This conclusion holds for versatile transformer decode-only LLMs (MQA, GQA, MLA, MoE FFN, etc.) validated by experiments, and adapts to both PD-aggregated and -disaggregated deployments.

A two-phase intra-node TP/DP parallel optimization algorithm is derived based on the conclusion. In *pre-profiling*

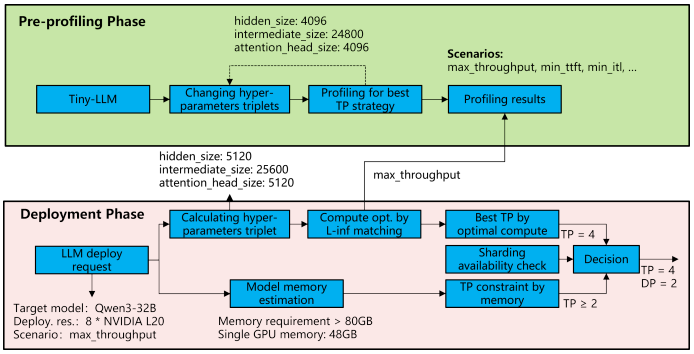

Fig. 3. An example workflow for TP/DP optimization for Qwen3-32B model deployment on 8×NVIDIA L20 GPUs with max_throughput scenario.

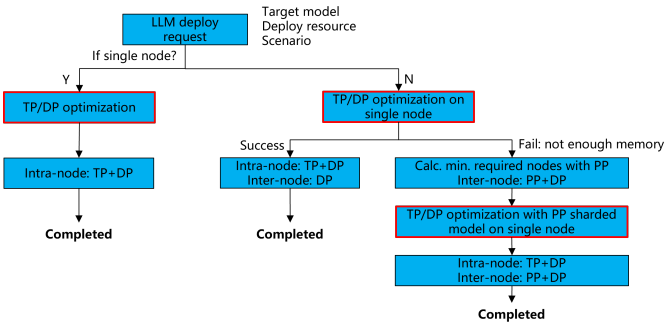

Fig. 4. The overall workflow of parallel optimization among TP, PP and DP. The rectangles with red outline represent optimization algorithm in Fig. 3.

phase, a tiny reference LLM (ref-LLM) profiles with different hyper-parameter triplets (hidden_size, intermediate_size, num_attention_heads) to mimic actual LLMs of different sizes for target optimization scenarios. In *deployment stage*, target LLM's triplet is extracted from its model definition, then compute-optimized parallel setup is chosen from pre-profiling results through minimum distance matching of candidate triplets. GPU memory and sharding availability are additionally checked for final TP size, as shown in Fig. 3.

Extending to multi-node cases, PP and DP are preferred across nodes for limited inter-node bandwidth assumption (e.g., ethernet). Fig. 4 shows multi-node adaptive parallel optimization with hybrid TP/PP/DP. If model size fits single node, DP is preferred to replicate inference instances across multiple nodes. Otherwise, PP is used to split model layers across nodes. TP optimization still applies to PP-partitioned layers on each node since transformer architecture is preserved.

B. RoCE Networks Optimization

LLM inference faces two typical network challenges. First, inference generates relatively fixed, concentrated single-flow burst transmissions, namely, low-entropy elephant flows, which likely causes congestion with traditional Equal-Cost Multi-Path (ECMP) routing [14]. Second, throughput and latency requirements significantly differ across inference stages. Prefill prioritizes throughput while decode minimizes latency. Traditional fixed RoCE ECN configurations, like watermark

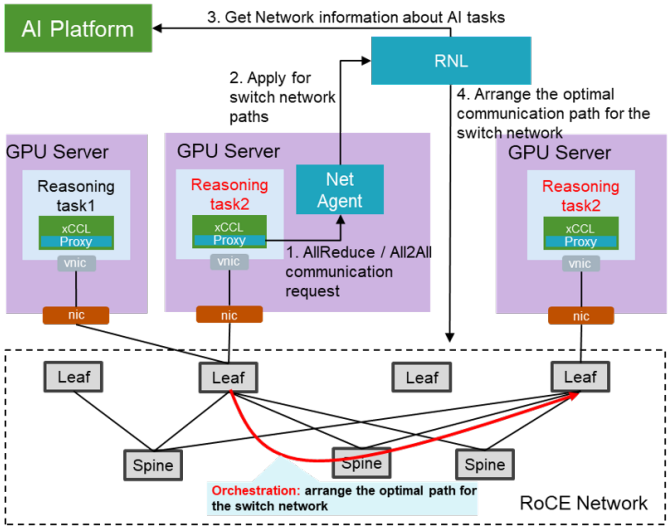

Fig. 5. RoCE dynamic path planning and load balancing in post-deployment optimization.

and queue buffer size, cannot adapt to conflicting link requirements. Existing RoCE network optimization studies [15], [17] mainly focus on generalized cases, failing to fit LLM inference scenarios. Actually, parameters of RoCE network and NCCL are highly correlated. For example, RoCE link bandwidth requirement is directly determined by Cooperative Thread Array (CTA), thread parallelism, and data blocks in NCCL; thus, they should be co-optimized.

Addressing this issue, Perceptor builds a novel optimization loop of initial network tuning (pre-deployment) → model deployment → traffic adaption (post-deployment) ↔ traffic monitoring. It involves three key innovations: RoCE-NCCL co-optimization, dynamic path planning and load balancing (Fig. 5), and RoCE ECN live adaption.

RoCE-NCCL co-optimization

Considering NCCL parameters, inference service bandwidth requirement B is approximately formulated by:

$$B = \min\left(\frac{8N_bN_cN_w\mathrm{W_s}}{10^9 \times T_s}, \quad Q\bar{B}\right) \tag{2}$$

where N_b and N_c denote NCCL data chunk size (NCCL_P2P_NETWORK_CHUNKSIZE) and minimum CTA size (NCCL_MIN_CTAS). In the inference, the number of wraps $N_w = 8$, and per warp size $W_s = 32$. $T_s = 0.1$s measures single-round communication time by experience. $Q = 0.95$ is NIC bandwidth correction factor for physical bandwidth $\bar{B} = 200$Gbps (NVIDIA ConnectX-7).

Based on Eq. (2) and RNL proposal [18], link j's end-to-end congestion score S_j is originally modeled by:

$$S_j = \alpha\frac{B + \sum_{i=1}^{n} B_i}{\bar{B}_j} + \beta\frac{\mathrm{PFC}_j}{\mathrm{PFC}_{th}} + \gamma\mathrm{ECN}_j + \delta\mathrm{BUF}_j \tag{3}$$

where 4 terms denote: link bandwidth utilization (with n interfered workloads B_i), Priority-based Flow Control (PFC)

trigger rate (as link congestion signals), ECN packet ratio, and queue buffer utilization. $\bar{B}_j$ is total link bandwidth. Empirical weights: $\alpha = 0.45$, $\beta = 0.2$, $\gamma = 0.2$, $\delta = 0.45$.

S_j represents link congestion with bandwidth as primary factor, and PFC, ECN, buffer as auxiliary congestion indicators. S_j instructs inference traffic routing as:

- $S_j \geq 0.9$: Severe congestion - Avoid routing.
- $0.8 \leq S_j < 0.9$: Mild congestion - Light traffic only (e.g., decode phase).
- $S_j < 0.8$: No congestion - All traffic allowed.

Key RoCE-NCCL co-optimization criteria and samples validated through numerous experiments:

- NCCL_NTHREADS: Threads per CTA - More threads increase data chunk size, requiring larger RoCE buffer and lower ECN watermark. 256 threads (high bandwidth) $\rightarrow$ 128MB buffer, 72% watermark; 128 threads (low latency) $\rightarrow$ 64MB buffer, 80% watermark.
- NCCL_BUFFSIZE: NCCL buffer size - Larger buffers increase throughput and bandwidth demand, requiring lower watermark and higher adaption coefficient K_{nccl}. 16MB buffer $\rightarrow$ 74% watermark, $K_{nccl} = 1.1$; 8MB buffer $\rightarrow$ 80% watermark, $K_{nccl} = 0.95$.
- NCCL_P2P_NET_CHUNKSIZE: P2P communication block size - Should match RoCE buffer. For TP = 32 case, 64KB block $\rightarrow$ 64MB buffer; 128KB block $\rightarrow$ 128MB buffer.

Dynamic path planning and load balancing

According to bandwidth estimation B and congestion score S_j, we have designed path planning and load balancing (Algorithm 1) to deal with link traffic saturations. We monitor congestion situation of all links, and continuously migrate partial workload to idle links when congestion occurs.

Algorithm 1 RoCE dynamic path planning and load balancing

Input: link congestion score S_j, link maximum and used bandwidths $\bar{B}_j$ and $\tilde{B}_j$, inference bandwidth requirement B, allocated links $\{\text{src\_links}\}$.

1: Monitor $S_i \geq 0.8,\ i \in \{\text{src\_links}\}$ ▷ trigger migration
2: Calc. $\{S_j\},\ j \in \{\text{other\_links}\}$ ▷ all available links
3: Filter $\{S_k | S_j < 0.8, \bar{B}_j - \tilde{B}_j \geq 1.2B\}$ ▷ migratable links
4: Select $k^* = \arg\min_k \{S_k\}$ ▷ destination link
5: Join $\{\text{src\_links}\} \leftarrow k^*$ ▷ add to source set
6: Migrate $\tilde{B}_{k^*} \leftarrow 0.2\tilde{B}_i$ ▷ 20% traffic migration
7: Update $S'_i \leftarrow S_i,\ S'_{k^*} \leftarrow S_{k^*}$ ▷ update congestion
8: **if** $S'_{k^*} \geq 0.8$ **then**
9: **goto** Step #1 ▷ destination congested
10: **else if** $S'_i \geq 0.8$ **then**
11: **goto** Step #6 ▷ source still congested
12: **else**
13: **goto** Step #15 ▷ no more congestion
14: **end if**
15: Done

RoCE ECN live adaption

ECN watermark (congestion notification threshold) directly affects link bandwidth and latency. Higher watermark with larger queue buffer suppresses false alarms, increasing bandwidth but suffering higher latency and congestion risk, and vice versa. The bandwidth utilization can be improved if ECN watermark adapts to the live inference bandwidth.

We have implemented ECN live adaption for dynamic bandwidth requirements. Based on real-time bandwidth measurement $\tilde{B}$, optimal ECN watermark can be calculated by

$$\text{ECN}_{adapt} = \text{ECN}_{base} \times P \times K_{nccl} \times \frac{\tilde{B}}{Q\bar{B}} \tag{4}$$

where ECN_{base} is baseline watermark benchmarked through stress testing for maximum bandwidth utilization with target latency. P reflects protocol utilization (0.9, 0.95, 0.98 for LL, LL128, Simple protocols). K_{nccl} denotes adaption coefficient determined by NCCL_BUFFSIZE (16MB $\rightarrow K_{nccl} = 1.1$, 8MB $\rightarrow K_{nccl} = 0.95$). ECN_{adapt} is periodically updated by new real-time bandwidth measurement $\tilde{B}$.

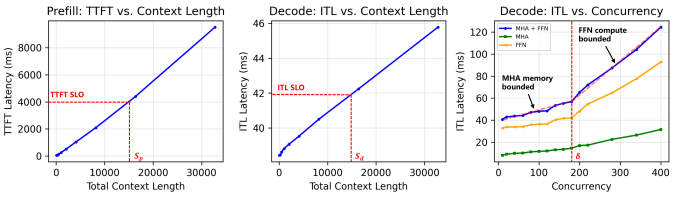

Fig. 6. TTFT (prefill) and ITL (decode) in terms of different input/output lengths and request concurrency. Testing is performed on a single NVIDIA L20, with vLLM inference on Qwen3-14B model. Request concurrency is fixed to 4 and input length varies from 16 to 8192 in first two sub-figures, whereas input length is fixed to 32 and concurrency varies from 10 to 400 in the last sub-figure.

C. SLO-Oriented PD Workload Scheduling

Request scheduling can effectively optimize goodput in PD-disaggregated deployment [19]. Shorter requests have higher queuing latency tolerance for less computing time, and can be delayed scheduling. Besides, carefully designed scheduling can make full use of all GPUs through load balancing. A novel PD dynamic workload scheduling algorithm is proposed, maximizing goodput under latency SLO constraints.

Prefill stage is compute-bounded, TTFT increases approximately linearly with input sequence length (Fig. 6, first sub-figure). Assuming continuous batching [20] enabled, single-forward compute time L_p is proportional to total processed sequence length S_p:

$$L_p = S_p / r \tag{5}$$

where r signifies the prefill compute rate determined by GPU compute capability which can be profiled in advance. Meeting TTFT SLO budget L_{TTFT} requires $L_p \leq L_{TTFT}$, so:

$$S_p \leq L_{TTFT} \times r \tag{6}$$

Multi-GPU prefill tends to use DP for scaling out since GPU compute can be easily saturated with small batch size. Load balancing should be considered across DP instances to fully

utilize GPUs. Assuming prefill request set $U_p = \{\text{Seq}_i\}, i \in \{1, \ldots, n\}$ processed in m prefill instances. For jth instance ($j \in \{1, \ldots, m\}$), assigned requests $U_p^j \subseteq U_p$ satisfy:

$$U_p^j = \left\{ \text{Seq}_{n_j} | \sum_{i \in \{n_j\}} \text{len}(\text{Seq}_i) \leq L_{TTFT} \times r \right\} \quad (7)$$

where $n_j \in \{1, \ldots, n\}$ are scheduled request indices for jth instance, $\text{len}(\cdot)$ signifies request input length. The optimal scheduled request sets for all m instances, denoted $\tilde{U}_p$, is:

$$\tilde{U}_p = \{\tilde{U}_p^0, \ldots, \tilde{U}_p^m\} = \underset{\{U_p^0, \ldots, U_p^m\}}{\arg\max} \left(\sum_{j=1}^{m} |U_p^j| \right) \quad (8)$$

where $|\cdot|$ is set cardinality. Eq. (8) maximizes total scheduled requests under sequence length constraint S_p per instance. This is a *constrained bin-packing problem* and can be approximately solved by First-Fit Decreasing (FFD) algorithm.

Decode stage is generally memory-bounded, but could be FFN compute-bounded for relatively large batch sizes (concurrency) and small context lengths (Fig. 6, last two sub-figures). In memory-bounded region, ITL is proportional to total accessed KV cache (equivalently, total context length). In compute-bounded region, ITL is dominated by FFN compute latency, increasing approximately linearly with decode concurrency. Token generation interval L_d can be modeled as:

$$L_d = \begin{cases} k_1 \times S_d + c & p \leq \delta \\ k_2 \times p & p > \delta \end{cases} \quad (9)$$

where S_d, p denote total context length and decode concurrency. Constants c, k_1, k_2 are determined by framework overhead, GPU bandwidth, and GPU compute. δ is compute saturation point depending on GPU arithmetic intensity. These parameters are profiled on specific GPU per LLM hyper-parameters [2]. Meeting ITL SLO budget L_{ITL} requires:

$$\begin{cases} S_d \leq (L_{ITL} - c)/k_1 \\ p \leq \min\{\delta, \quad L_{ITL}/k_2\} \end{cases} \quad (10)$$

Assuming single decode instance (DP = 1) since decode requires large GPU memory for high concurrency, optimal scheduled request set $\tilde{U}_d \subseteq U_d$ from all candidates U_d is:

$$\begin{aligned} \tilde{U}_d = & \underset{\hat{U}_d \subseteq U_d}{\arg\max} \left(|\hat{U}_d| \right) \\ \text{s.t.} \quad & |\hat{U}_d| \leq \min\{\delta, \quad L_{ITL}/k_2\} \\ & \sum_{\text{Seq}_i \in \hat{U}_d} \text{len}(\text{Seq}_i) \leq (L_{ITL} - c)/k_1 \end{aligned} \quad (11)$$

where $\hat{U}_d$ denotes all possible decoding request sets for scheduling. Eq. (11) is a *0/1 knapsack problem* which can be solved by dynamic programming.

Prefill stage may encounter compute bottlenecks with bursty requests since minimum compute resource is allocated by DF strategy. We propose locally prefill strategy to address this issue. Leveraging chunked prefill [21], we schedule overloaded requests to decode instance for prefilling, alleviating compute pressure of prefill instances.

Decode typically operates in memory-bounded region with adequate compute resource allocated. When a new prefill request Seq_l is assigned to decode instance, with existing decoding workload, sustaining ITL SLO requires:

$$\begin{cases} S_d + \text{len}(\text{Seq}_l) \leq (L_{ITL} - c)/k_1 \\ p + \text{len}(\text{Seq}_l) \leq \min\{\delta, \quad L_{ITL}/k_2\} \end{cases} \quad (12)$$

where the new prefill request approximately introduces $\text{len}(\text{Seq}_l)$ extra tokens for KV cache memory access and FFN computation. This also holds for multiple prefill requests. In practice, we repeatedly schedule the shortest queuing request to decoder instance until Eq. (12) no longer holds.

In general, the dynamic PD workload scheduling algorithm is summarized as:

1) Schedule prefill requests. Limit total input length per forward iteration. Optimal scheduling follows Eq. (8).
2) Schedule decode requests. Limit total context length and decode concurrency per forward iteration. Optimal scheduling follows Eq. (11).
3) Schedule queuing prefill requests. Rescheduling to decode instance follows Eq. (12).

TABLE I
HARDWARE CONFIGURATIONS IN EXPERIMENTS

Device	Model	Specifications	Quantity
Server	Lenovo WA5480 G3	GPU: 8×NVIDIA L20 NIC: 4×NVIDIA CX7 200G 2-port	4
RoCE Switch	Ruijie S6980-128DC	128×200G Interfaces	1

IV. EXPERIMENTS AND DISCUSSIONS

Numerous experiments are conducted for Perceptor performance evaluation. Unless specified, testing adopts a 4-node cluster with 32×NVIDIA L20 GPUs and 200G RoCE Spine-Leaf switching network (Table I). vLLM v0.9.2 and Dynamo v0.5.1 are used for PD-aggregated and -disaggregated deployments respectively. SGLang v0.4.10 is used where noted.

A. Parallelism Optimization

Firstly, parallelism optimization is evaluated. Table II shows key hyper-parameters of ref-LLM adopted in pre-profiling. We carefully designed 14 hyper-parameter triplets[1], representing versatile LLMs under 100B parameters. For each triplet, optimal parallel setup is obtained under different user scenarios with moderate request concurrency. Based on profiling results, we evaluated parallelism prediction over 20+ well-known LLMs on single L20 node with PD-aggregated deployment.

Fig. 7 shows experiment results. Relying on pre-profiling results (left sub-figure), optimal parallel prediction accuracies appear in last two sub-figures. Compared to fixed parallelism

[1]14 triplets (hidden_size, intermediate_size, attention_head_size): {(896,4864,576), (1024,3072,2048), (1536,8960,1280), (2048,11008,1280), (2048,6144,2048), (2560,9728,3072), (3584,18944,2304), (4096,14336,3072), (4096,12288,3072), (5120,13824,3584), (5120,27648,3584), (5120,17408,3584), (5120,25600,5120), (8192,28672,5120)}.

TABLE II
KEY HYPER-PARAMETERS OF REF-LLM

Key	Value	Key	Value
architectures	Qwen3ForCausalLM	max_position_embeddings	40960
num_hidden_layers	16	hidden_size	Vary in pre-profiling.
head_dim	128	intermediate_size	Vary in pre-profiling.
num_key_value_heads	8	num_attention_heads	Vary in pre-profiling.

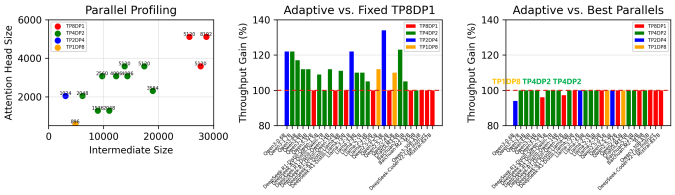

Fig. 7. Experiment results on parallelism optimization. The left sub-figure shows pre-profiled optimal parallelisms among 14 triplets. intermediate_size is marked on each profiling point. The parallel prediction results are shown in last two sub-figures. The bar color denotes predicted parallel, and the actual best parallel is marked on the top of the bar on the right sub-figure if the prediction fails to meet actual parallel setup.

(TP = 8), our proposal achieves up to 34% inference throughput improvement. We correctly predicted 25 of 28 cases (89.3% top-1 accuracy). Even in the remaining 3 failed cases, throughput loss stays under 6%.

Table III provides more extensive results on different inference frameworks and user scenarios. Our proposal has consistently achieved good prediction accuracy above 82%, and controlled performance gap under 10% for failed predictions.

B. RoCE Network Optimization

RoCE-NCCL co-optimization and traffic load balancing are evaluated on the 4-node cluster by throughput over different collective communication operations. As shown in Fig. 8, our optimization has achieved considerable throughput improvement, especially for All-Reduce and All-Gather. Right sub-figure illustrates traffic workload distribution over different links. Dynamic path planning and load balancing greatly improve link balance, enabling efficient bandwidth utilization.

Performance of RoCE network post-deployment (online) optimization is shown in Fig. 9. In different scenarios, dynamic optimization increases inference throughput by 9%-15% while reducing TTFT/ITL latency by 7%-15%.

C. End-to-End Performance for PD Aggregation

End-to-end inference performance with Perceptor adaptive optimization on PD-aggregated deployment is consolidated. Fig. 10 compares performance of different LLMs on single

TABLE III
PARALLELISM PREDICTIONS ON MORE SCENARIOS

User Scenario	Top-1 Acc.	Avg. Loss	User Scenario	Top-1 Acc.	Avg. Loss
Max Thr. (vLLM)	89.3% (25/28)	4.27%	Max Thr. (SGLang)	92.6% (25/27)	9.95%
Min TTFT (vLLM)	89.3% (25/25)	3.55%	Min TTFT (SGLang)	88.9% (24/27)	10.00%
Min ITL (vLLM)	82.1% (23/28)	3.02%	Min ITL (SGLang)	88.9% (24/27)	3.97%

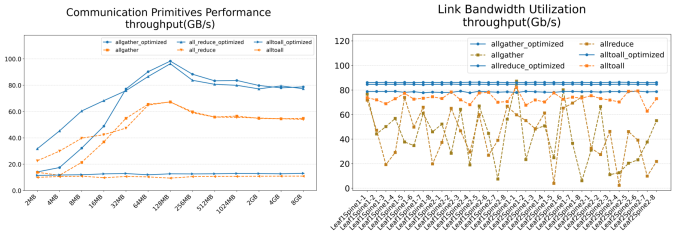

Fig. 8. The throughput performance of NCCL operators. "*_optimized" indicate the cases with RoCE network co-optimization as well as dynamic path planning and load balancing enabled.

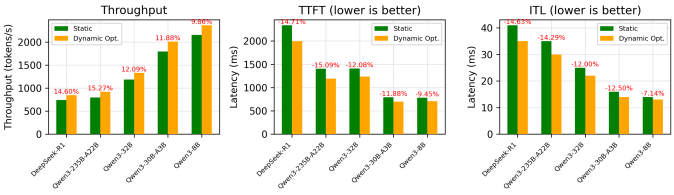

Fig. 9. Inference performance on different LLMs of RoCE network dynamic optimization, i.e., dynamic path planning and load balancing, and RoCE ECN live adaption. The ratio on bars denotes the gain over baseline.

L20 node. Throughput, TTFT, and ITL are improved for two testing LLMs, especially at high concurrency.

Multi-node results are shown in Fig. 11, performed on a 2-node cluster with 8×NVIDIA H20 GPUs and 4×NVIDIA CX7 200G RoCE on each node. RoCE network optimization is independently evaluated. At 500 concurrency, Perceptor optimization increases throughput by 77% while reducing TTFT and ITL by 33% and 43% respectively.

D. End-to-End Performance for PD Disaggregation

We have verified goodput impact through workload scheduling optimization in PD-disaggregated deployment. Assuming SLO constraints TTFT $\leq$ 3s and ITL $\leq$ 50ms, PD GPU allocation ratio is 1 : 3 according to SLO and DF strategy. In Fig. 12 left sub-figure, PD dynamic workload scheduling

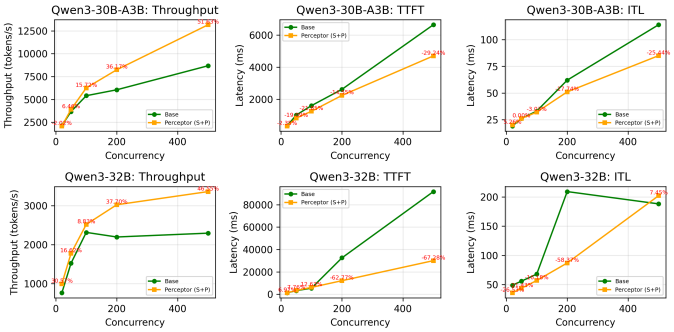

Fig. 10. Performance comparison with Perceptor adaptive optimization on single node. Input/output lengths are fixed to 256. In baseline, TP size is determined by minimum sharding for GPU memory adaption, and scales out to 8 GPUs with DP. "S" indicates system optimization. "P" indicates parallelism optimization. The ratio on the curve denotes the gain over baseline.

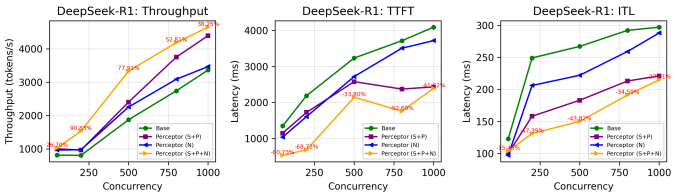

Fig. 11. Performance comparison with Perceptor adaptive optimization on a 2-node cluster with SGLang. Input/output lengths are fixed to 512, and TP $= 16$ is adopted in baseline. "N" indicates RoCE network optimization. The ratio on the curve denotes the gain over baseline.

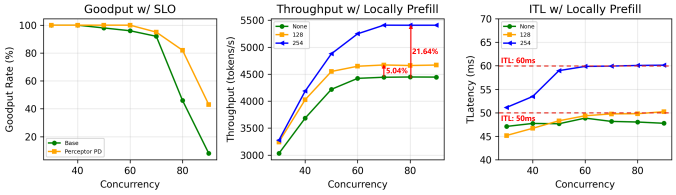

Fig. 12. Inference performance on PD-disaggregated deployment with Perceptor workload scheduling optimization. A single node with 4×NVIDIA L20 GPUs is adopted in the testing, and target model is Qwen3-14B. Input lengths are uniformly distributed between 32 and 512, while output length is fixed to 64. In the last two sub-figures, "None" disables locally prefill. 128 and 254 represent the maximum locally prefill length Seq_l.

achieves notable goodput improvement. Intuitively, requests with longer inputs are prioritized for goodput achievements on harder cases. The latter two sub-figures provide performance with locally prefill strategy. Maximum prefill lengths Seq_l are calculated as 128 (original ITL SLO) and 254 (relaxed ITL $\leq$ 60ms). The throughput is increased by 5.04% and 21.64% with locally prefill enabled, maintaining ITL SLOs.

V. Conclusions and Prospects

We have introduced Perceptor, an end-to-end adaptive tuning engine for LLM inference service delivery. Overcoming diverse environments and complex tuning parameters, Perceptor achieves lifetime-span adaptive tuning through its perceive-adaption loop. Composed of pre- and post-deployment optimizations, a novel multi-stage tuning pipeline is built, and complicated cross-layer optimization is decoupled by stages. We have proposed innovative algorithms in tuning stages for parallel strategy, RoCE networks, and PD workload scheduling optimizations. Numerous test results validate the superiority of Perceptor across various environments and user scenarios.

There are several ongoing works. Firstly, we are trying to involve Expert Parallel (EP) to parallelism optimization for rapid growth of MoE LLMs. Secondly, we are attempting to introduce reinforcement learning in RoCE network adaption. Thirdly, we are working on optimizing PD workload scheduling with live inference metrics feedback.

References

[1] J. Kaplan, S. McCandlish, T. Henighan *et al.*, "Scaling laws for neural language models," 2020. [Online]. Available: https://arxiv.org/abs/2001.08361

[2] Y. Zhong, S. Liu, J. Chen *et al.*, "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*, ser. OSDI'24. USA: USENIX Association, 2024.

[3] A. Vaswani, N. Shazeer, N. Parmar *et al.*, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[4] W. Kwon, Z. Li, S. Zhuang *et al.*, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626.

[5] L. Zheng, L. Yin, Z. Xie *et al.*, "Sglang: efficient execution of structured language model programs," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24. Red Hook, NY, USA: Curran Associates Inc., 2025.

[6] Microsoft, "Kaito," 2023. [Online]. Available: https://github.com/kaito-project/kaito

[7] K. Cheng, Z. Wang, W. Hu *et al.*, "Scoot: Slo-oriented performance tuning for llm inference engines," in *Proceedings of the ACM on Web Conference 2025*, ser. WWW '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 829–839.

[8] S. Kim, J. Ha, and Y. Kim, "Oltunes: Online learning-based auto-tuning system for dl inference in heterogeneous gpu cluster," *Cluster Computing*, vol. 28, no. 9, Aug. 2025.

[9] NVIDIA, "Dynamo," 2025. [Online]. Available: https://github.com/ai-dynamo/dynamo

[10] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying numa and contention effects in multi-gpu systems," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: Association for Computing Machinery, 2011.

[11] X. Hu, T. Zeng, X. Yuan *et al.*, "Bestserve: Serving strategies with optimal goodput in collocation and disaggregation architectures," 2025. [Online]. Available: https://arxiv.org/abs/2506.05871

[12] Microsoft, "Vidur," 2024. [Online]. Available: https://github.com/microsoft/vidur

[13] C. Hu, H. Huang, L. Xu *et al.*, "Inference without interference: Disaggregate llm inference for mixed downstream workloads," 2024. [Online]. Available: https://arxiv.org/abs/2401.11181

[14] C. Hopps, "Rfc2992: Analysis of an equal-cost multi-path algorithm," USA, 2000.

[15] M. Alizadeh, T. Edsall, S. Dharmapurikar *et al.*, "Conga: distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 503–514.

[16] Y. Li, L. Li, C. He *et al.*, "Adaptive parallelism for llm inference with model irrelevant profiler," in Proceedings of IEEE 29th International Conference on Computer Supported Cooperative Work in Design (CSCWD'26), In press.

[17] H. Luo, J. Zhang, M. Yu, Y. Pan, T. Pan, and T. Huang, "Seqbalance: Congestion-aware load balancing with no reordering for roce," 2024. [Online]. Available: https://arxiv.org/abs/2407.09808

[18] Z. Shen, L. Li, L. Zhang *et al.*, " RNL: RoCE Network Loadbalance with AI Traffic Characteristics and Link Congestion Awareness ," in *2025 IEEE Cyber Science and Technology Congress (CyberSciTech)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2025, pp. 45–54.

[19] C. Wang, P. Zuo, Z. Chen *et al.*, "Prefill-decode aggregation or disaggregation? unifying both for goodput-optimized llm serving," 2025. [Online]. Available: https://arxiv.org/abs/2508.01989

[20] G.-I. Yu, J. S. Jeong, G.-W. Kim *et al.*, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538.

[21] A. Agrawal, N. Kedia, A. Panwar *et al.*, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," *Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, Santa Clara*, 2024.