FASTEPD: IN-NETWORK SCHEDULING FOR DISAGGREGATED LMM INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Encode–Prefill–Decode (EPD) disaggregation has been adopted to improve the performance of Large Multimodal Model (LMM) services. It separates the encoding (E), prefill (P), and decoding (D) stages across dedicated GPU servers while generating multi-stage data traffic among different servers. These data flows are commonly transmitted by switches in data centers, where queuing management and scheduling mechanisms significantly affect the overall performance. However, existing EPD inference systems lack in-network optimization which is necessary for efficient flow identification and queue scheduling, given the limited network resources and complex traffic characteristics. In this work, we propose FastEPD, an in-network identification and scheduling framework to accelerate LMM EPD inference by reducing the service completion time (SCT) for network transmissions. It consists of two main components: 1) lightweight multidimensional flow identification, and 2) service-oriented multi-queue scheduling. Experiments on a DPDK prototype driven by real model workloads show that FastEPD outperforms canonical network schemes, improving the overall SCT by 57.67 - 79.09% over FIFO and 41.02 - 68.10% over RSS, respectively. Experiments on a two-tier fat-tree also demonstrate that FastEPD remains beneficial as the topology scales from 32 to 128 endpoints.

1 INTRODUCTION

Large Multimodal Models (LMMs) have emerged as a general interface to diverse data types and applications (Alayrac et al., 2022; Liu et al., 2023). To improve the performance of LMM inference services, Encode–Prefill–Decode (EPD) disaggregation separates encoding (E), prefill (P), and decoding (D) stages onto dedicated GPU servers (Singh et al., 2025). Specifically, encoding servers perform parallel encoding of multimodal inputs, while prefill and decoding servers implement LMM backbones. This separation introduces corresponding inter-server communications, including multimodal token migration from E to P, KV cache migration from P to D, and MoE all-to-all traffic within P and D (Zhong et al., 2024). Therefore, an inference request generates multiple communication stages with strict dependency relationships and heterogeneous data volumes.

For practical implementation of EPD inference services, these communication flows commonly share the same data-center network, and their transfer cost can be substantial in disaggregated serving. In particular, prior work (Zhang et al., 2025) shows that even KV-cache transmission can account for up to 42.2% of total completion time under bandwidth-constrained settings. When these data packets contend for the same output port or link, they are buffered by the network switch that determines their transmission orders. In-network packet scheduling can decide whether a critical flow can be transmitted earlier, thus unlocking the downstream stage of EPD services in advance and impacting total communication time. However, existing EPD systems mainly focus on application-layer optimization, including phase-specific GPU provisioning (Singh et al., 2025; Zhong et al., 2024), request placement (Patel et al., 2024), and efficient KV cache transfer (Qin et al., 2025), while conventional packet schedulers primarily focus on fairness or packet/flow priorities and optimize network-level objectives such as flow completion time (FCT) (Shreedhar & Varghese, 1995; Alizadeh et al., 2013). The specialized network optimization for EPD inference services is still lacking, and introduces three key requirements given the EPD characteristics and network conditions.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Firstly, the network needs compact semantics to identify data packets from different services and EPD stages, since the switch will suffer substantial overhead if it maintains complex information or predictive mechanisms at packet rate. Secondly, the scheduler must efficiently exploit EPD stage dependencies and heterogeneous request sizes, while avoiding starvation across competing requests. Finally, the framework should adapt to dynamic request loads and existing network stacks without tuning related parameters or modifying orthogonal network protocols.

In this work, we propose FastEPD, an in-network identification and scheduling framework to tackle these challenges. Generally, FastEPD combines lightweight multidimensional flow identification with service-oriented multi-queue scheduling. [1] Firstly, the servers reuse an existing packet-header field to encode a compact service identifier, the EPD flow type, and the service size. Secondly, the switches parse the labelled field and map corresponding packets into multiple designated queues, using service-oriented dynamic scheduling schemes to optimize service completion time (SCT). Therefore, FastEPD does not require extra packet tags or incur complex identification overhead, enabling a lightweight design and an independent scheduling building block for existing network protocols.

We implement FastEPD using a DPDK prototype driven by real model workloads and compare its performance with canonical network schemes. Experimental results show that FastEPD reduces overall SCT by 57.67 - 79.09% over FIFO and 41.02 - 68.10% over RSS for 1k - 16k requests in a dumbbell network of 7 nodes. An ablation study and endpoint interference tests verify the effectiveness and robustness of the FastEPD framework. Finally, we test the performance in a two-tier fat-tree with 32 and 128 nodes, demonstrating that FastEPD remains beneficial as the topology scales.

2 Background and Challenges

2.1 LMM Inference with EPD Disaggregation

Figure 1 illustrates the EPD disaggregation deployment, in which E, P and D servers for different computation stages communicate through a data-center network. We mainly focus on LMM services based on the Mixture-of-Experts (MoE) architecture (Fedus et al., 2022; DeepSeek-AI, 2024), while FastEPD can be adapted to dense LMMs with minor modifications to label information and queuing management.

For the EPD disaggregation setting in Fig. 1, encoding servers can use data parallelism (DP) to process multimodal inputs from different prompts (Singh et al., 2025). Prefill and decoding servers implement the LMM backbone with expert parallelism (EP), with each server holding a subset of experts (Lepikhin et al., 2021; DeepSeek-AI, 2024). Then, for a batch of B inference requests, the steps of the EPD disaggregation mode are described as follows.

Encoding phase (E): The multimodal input tensor i_m is processed by the encoding servers, which convert it into a multimodal token tensor $v_t^e = E(i_m) \in \mathbb{R}^{B\times N_m\times M}$ and concatenate it with the text input tensor $i_t \in \mathbb{R}^{B\times N_t\times M}$ to form a joint input token tensor, i.e., $I = Concat(v_t^e, i_t) \in \mathbb{R}^{B\times(N_m+N_t)\times M}$, where N_m denotes the number of multimodal tokens, N_t denotes the number of text input tokens, and M denotes the embedding size of a token.

EP-migration: The generated joint input token tensor I is transferred to prefill servers via the EP-migration process. The communication flow is denoted as F_{EP} with data volume $B \times (N_m + N_t) \times M \times \tau$, where τ denotes the data type size (e.g., 2 bytes for FP16).

Prefill phase (P): Prefill servers process the joint input I to produce the initial KV cache and first output token, i.e., $kv_1^p, o_1^p = P(I)$. Additionally, for each MoE transformer block deployed on prefill servers, expert routing induces dispatch and combine communication across the expert parallel group (Lepikhin et al., 2021; Fedus et al., 2022). In the process, each flow is denoted as F_{PP} with data volume $B \times (N_m + N_t) \times M \times k \times \tau$, and there are a total of $2 \times L$ flows in an all-to-all communication, where k represents the top-k routed experts and L is the number of transformer blocks in the LMM backbone.

[1] FastEPD is designed as an independent building block, which is complementary to existing application-layer optimization and network-wide load balancing or congestion control mechanisms.

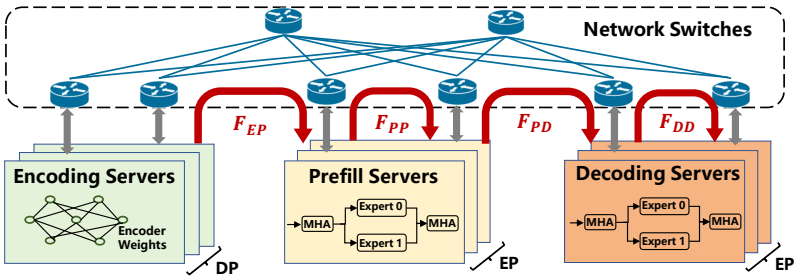


Figure 1: Illustration of inference services with EPD disaggregation mode

PD-migration: The generated KV cache kv_1^p and token o_1^p are transferred to the decoding servers. The communication flow is denoted as F_{PD} with data volume $L \times 2B \times (N_m + N_t) \times M \times \tau + B \times M \times \tau$.

Decoding phase (D): Based on the received KV cache and first token from F_{PD}, the decoding servers generate the next token o_{t+1}^d and update the cache $kv_{t+1}^d, o_{t+1}^d = D(kv_t^d, o_t^d)$ through multiple autoregressive forward computations until the final outputs are generated. Similar to the P phase, there are $2 \times L$ all-to-all communication flows, denoted as F_{DD}. However, the data volume becomes $B \times M \times k \times \tau$, since only the recently updated token is required to be routed during each step of the autoregressive process in the D phase.

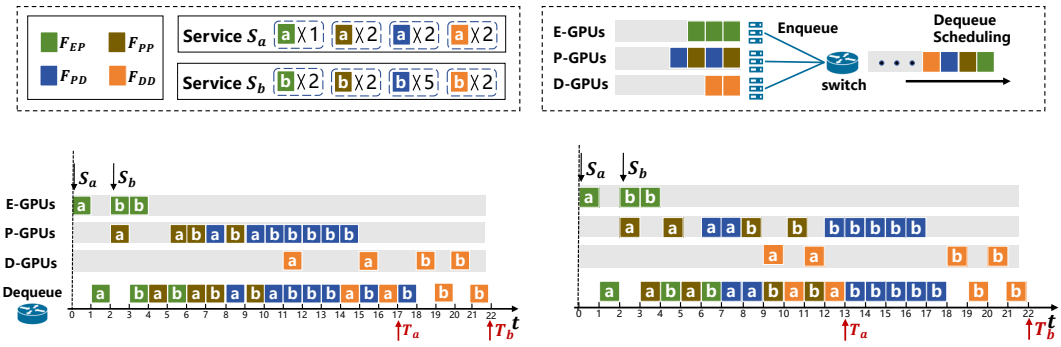
2.2 Performance Bottlenecks in LMM EPD Inference

Existing work on disaggregated LMM inference mainly optimizes the application layer, including resource allocation and placement for distinct stages (Singh et al., 2025; Zhong et al., 2024; Patel et al., 2024), as well as the transfer and reuse of KV-cache state (Qin et al., 2025). However, these systems do not specialize switch-side packet queueing and scheduling for EPD communication semantics. Specifically, we find that different packet scheduling schemes lead to distinct inference performance when communication flows from multiple inference services traverse a single switch simultaneously. Figure 2 demonstrates an example of network scheduling for two LMM EPD inference services S_a and S_b. They are assumed to run on the same encoding, prefill and decoding servers. Packets of different flows are enqueued through corresponding ports by a unique switch, and the switch is assumed to dequeue one data packet per time slot.

Without packet scheduling optimization, as shown in Fig. 2a, the services S_a and S_b have a longer average completion time of $\frac{17+22}{2} = 19.5$. This is because canonical network packet-scheduling protocols mainly focus on objectives such as per-flow fairness (Shreedhar & Varghese, 1995) or one-dimensional packet priorities (Nichols et al., 1998; Alizadeh et al., 2013). They fail to capture the characteristics of concurrent inference services, resulting in random packet dequeue. However, the minimal average completion time of $\frac{13+22}{2} = 17.5$ can be obtained with the optimized scheduling order shown in Fig. 2b. For instance, it prioritizes transferring packets of F_{PP} in service S_a, resulting in earlier arrival of the consequent flow F_{PD}. Besides, F_{DD} of S_a is transferred prior to F_{PD} of service S_b to reduce additional blocking time. This motivates us to efficiently identify and schedule communication flows of different LMM services at the network layer to improve inference speed. Doing so involves three main challenges:

The gap between the identification of various flow features and limited switch resources. As illustrated in Fig. 2, the packet scheduling scheme in switches is optimized based on the identification of various flow features, including the specific type of flow, the service it belongs to and the data volume. Meanwhile, comprehensive identification is difficult to implement in a line-rate switch, as analysis and computation of different traffic states must fit a bounded pipeline and complete at packet rate (Bosshart et al., 2014; Sivaraman et al., 2016). Explicit per-packet metadata also consumes header space and must respect the path MTU, while leveraging predictive algorithms imposes extra overhead on the switch and yields reduced accuracy.

Coordinated scheduling with complex EPD dependencies and heterogeneous flow transmission times. Compute stages and communication flows have complex dependencies during an LMM inference service with EPD disaggregation. If the switch gives excessive priority to one flow type



(a) Illustration of random scheduling

(b) Illustration of optimal scheduling

Figure 2: An example of network-layer flow scheduling for two LMM EPD inference services. The LMM has one transformer layer, each service performs one autoregressive D-phase step, and computation time is set to zero. The dependent flow sequence is $F_{EP} \rightarrow F_{PP} \rightarrow F_{PD} \rightarrow F_{DD}$, with two all-to-all communication rounds in both F_{PP} and F_{DD}.

and overlooks others, dependent compute stages may stall while waiting for data, reducing overall compute utilization. Meanwhile, inference services in data centers have diverse prompt lengths, resulting in highly heterogeneous transmission times for communication flows of different services. In this case, a proper scheduling scheme should prevent short flows from being blocked while also avoiding the starvation of long flows.

Adaptability and compatibility of framework design for dynamic heterogeneous services. Identification and scheduling schemes configured with fixed parameters struggle to adapt to the diverse inference services arriving at different times, while effective hyperparameter tuning is difficult for specific implementations and brings about extra overhead for both servers and switches. Therefore, an adaptive framework is required to constantly accommodate dynamic service loads. Additionally, it should be seamlessly compatible with current network stacks, such as TCP/IP (Eddy, 2022) or RDMA-based (Recio et al., 2007) protocols.

In this paper, our main goal is to address the above challenges by proposing an in-network identification and scheduling framework to accelerate LMM EPD inference. All frequently used notations are summarized in Table A.1 of the Appendix.

3 FastEPD Design

3.1 System Overview

Figure 3 illustrates the overview of FastEPD, which consists of two key components: lightweight multidimensional flow identification and service-oriented multi-queue scheduling. Firstly, for any EPD inference service, servers reuse packet headers of existing protocols and label data packets with essential features before transmission (Sec. 3.2). Secondly, based on the identification results, the network switch places these packets into appropriate queues and performs service-oriented scheduling to accelerate service completion (Sec. 3.3). In this way, FastEPD provides an efficient network communication infrastructure for LMM EPD services.

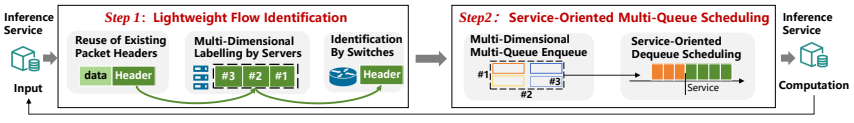


Figure 3: Overview of the FastEPD framework and its major procedures on the server and switch sides.

3.2 LIGHTWEIGHT FLOW IDENTIFICATION

FastEPD fully utilizes server-side assistance by reusing existing packet headers and applying server-side labeling. Specifically, servers can more conveniently obtain relevant information about inference services and have more computational resources to process it, while switches commonly parse compact header or metadata fields for queue management and load balancing (Bosshart et al., 2014; Nichols et al., 1998).

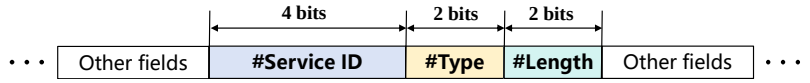


Figure 4: The one-byte FastEPD scheduling tag carried by each packet in the controlled DPDK prototype.

Reuse of Protocol Packet Headers. Figure 4 illustrates the packet headers reused for flow identification. FastEPD requires an eight-bit tag per packet, which abstracts three key dimensions of flow features as follows.

- **#Service ID**: Four bits are used to identify different batches of requests. An ID is released and reusable after its service state retires. [2]
- **#Type**: The two-bit field represents the four communication-flow types: *00* for F_{EP}, *01* for F_{PP}, *10* for F_{PD}, and *11* for F_{DD}.
- **#Service Size (Length)**: Two bits encode short (S), medium (M), and long (L) classes based on $B \times (N_m + N_t)$. [3] *00* identifies non-EPD traffic.

Multi-Dimensional Labeling Assisted by Servers. Servers label packets before transmission. E servers assign the Service ID and classify length using δ_{sm} and δ_{ml}; P and D servers recover this context from preceding flows, while each sender supplies the current flow type. The E-stage serving group shares the thresholds across its workers and periodically selects them to approximately balance aggregate token volume across S/M/L classes (Eq. 1).

$$\sum_{0<l_s<\delta_{sm}} l_s = \sum_{\delta_{sm}\le l_s<\delta_{ml}} l_s = \sum_{\delta_{ml}\le l_s} l_s \ , \ l_s = B \times (N_m^s + N_t^s) \tag{1}$$

3.3 SERVICE-ORIENTED MULTI-QUEUE SCHEDULING

FastEPD combines multidimensional queues with a service-oriented dequeue scheme.

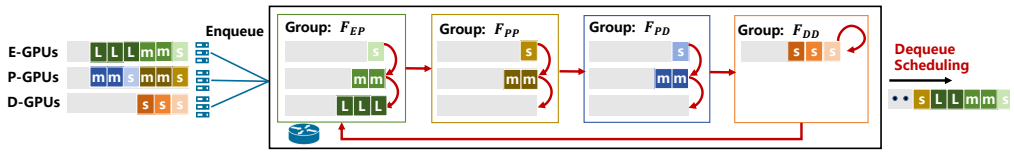


Figure 5: Illustration of multidimensional multi-queue design and dequeue scheduling in FastEPD. Packets of the same color belong to the same communication flow.

Multi-Dimensional Multi-Queue Design. Figure 5 first groups flows by EPD type to expose their dependencies, and then separates each group into three queues based on different service lengths. Each queue records its packet count p_q and active Service IDs to derive the mean packet count per service $\hat{p}_q$ used by the dequeue scheduler.

[2]The field can be widened when more metadata space is available. Figure 6 shows no systematic performance benefit from widening the Service-ID field beyond four bits in the evaluated workload.

[3]The current design assumes that requests belonging to one logical service (or batch) remain grouped for network identification across E, P and D processes.

Service-Oriented Dequeue Scheduling. For multi-queue management, dequeue scheduling determines the polling order and the number of packets per poll. FastEPD polls queues by group, as illustrated in Fig. 5. Within each group, FastEPD sequentially dequeues queues for short, medium or long services. The polling order remains fixed, avoiding periodic updates and reducing switch overhead, while the number of packets dequeued adapts to real-time service loads.

For each group, U service units are dequeued, where U is the minimum number of services among non-empty groups. Eqs. (2) and (3) give the dequeued service units u_S, u_M and u_L for short, middle and long queues, respectively (Appendix C). The scheduler tends to dequeue the corresponding contiguous packet blocks, i.e., $(u_S \times \hat{p}_S)$, $(u_M \times \hat{p}_M)$ and $(u_L \times \hat{p}_L)$, of certain units (u_S, u_M, u_L) of services at once. Additionally, the proportional constraint preserves fairness and avoids over-sacrificing long services. It is updated as p_q and $\hat{p}_q$ vary with real-time service loads.

$$u_S + u_M + u_L = U \tag{2}$$

$$(u_S \times \hat{p}_S) : (u_M \times \hat{p}_M) : (u_L \times \hat{p}_L) = p_S : p_M : p_L \tag{3}$$

3.4 Online FastEPD Procedure

Algorithms 1 and 2 describe FastEPD flow identification and multi-queue scheduling, respectively. Table A.1 of the Appendix summarizes the notation used in Algorithms 1 and 2.

Algorithm 1 Server-assisted flow identification

Input: Service s; role r; type f; packets $\mathcal{P}_f$; thresholds δ_{sm}, δ_{ml}
Output: Each $p \in \mathcal{P}_f$ carries $\langle \bar{s}, f, c_s \rangle$

if the update epoch expires at an E server **then**
  Update δ_{sm}, δ_{ml} using Eq. (1)
if $r = E$ and s is unlabeled **then**
  $\bar{s} \leftarrow$ AllocateID
  $c_s \leftarrow$ Classify$(B(N_m^s + N_t^s), \delta_{sm}, \delta_{ml})$
else
  $(\bar{s}, c_s) \leftarrow$ RecoverContext(s)
for all $p \in \mathcal{P}_f$ **do**
  p.tag $\leftarrow$ Encode$(\bar{s}, f, c_s)$
  Transmit(p)

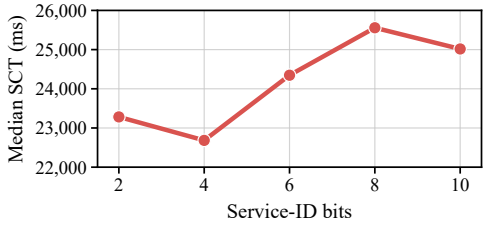


Figure 6: Sensitivity of Service ID width in FastEPD for 16,384 requests with batch size 64 in DPDK tests.

Algorithm 2 Switch queueing and scheduling

Input: Packet p; queues $Q_H, Q_M, Q_{f,c}$; queue states $(p_q, A_q, \hat{p}_q)$; service budget U
Output: Queue assignment and positive active-class budgets

procedure Enqueue(p)
  if p is ACK/PFC/QCN/NACK traffic **then**
    Q_H.Push(p); **return**
  else if p has no FastEPD tag **then**
    Q_M.Push(p); **return**
  $(\bar{s}, f, c) \leftarrow$ Decode$(p$.tag$)$
  $Q_{f,c}$.Push(p); update its state
procedure ScheduleRound
  Serve non-empty Q_H, then Q_M
  for $g \in [F_{EP}, F_{PP}, F_{PD}, F_{DD}]$ **do**
    $\mathcal{C}_g \leftarrow \{c \in \{S, M, L\} : p_{g,c} > 0\}$
    if $\mathcal{C}_g \neq \emptyset$ **then**
      Compute u_c using Eqs. (2)–(3)
      $b_c \leftarrow \min(p_{g,c}, \text{round}(u_c \hat{p}_{g,c}))$
      Enforce $b_c > 0$
      for $c \in [S, M, L] \cap \mathcal{C}_g$ **do**
        Dequeue b_c packets; update state

3.5 Implementation and Overhead

Figure 7 shows the DPDK implementation of FastEPD (Appendix D). The infrequent labeling-threshold update sorts n sampled service sizes in $O(n \log n)$ time and $O(n)$ space on CPU-equipped servers. In the network, FastEPD requires one byte of endpoint-provided scheduling metadata: two bits for flow type, two bits for length class, and four bits for the Service ID. Fixed-size fields and queue mapping give identification and rte_ring enqueue/dequeue operations $O(1)$ complexity. Our controlled DPDK prototype carries this byte in the IPv4 ToS field, so it does not expand the header (Appendix B). The scheduling mechanism is orthogonal to the underlying transport and does not modify congestion-control or routing logic. Table 4 reports processing overheads measured using the DPDK prototype.

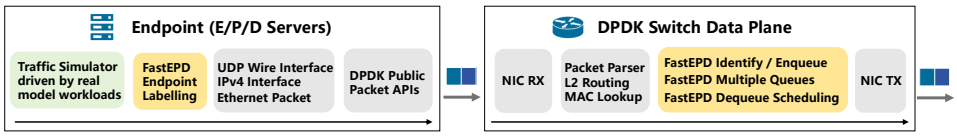


Figure 7: Illustration of the DPDK implementation of the FastEPD design.

4 Experiments

4.1 Experimental Setup

The real-workload traffic simulator in Fig. 7 instantiates DeepSeek-V3 (DeepSeek-AI, 2024) using the parameters in Table 2. [4] Request workloads are drawn from the one-week Azure LMM inference trace released with ModServe (Microsoft Azure, 2025; Qiu et al., 2025). The simulator generates its F_{EP}, F_{PP}, F_{PD}, and F_{DD} flows from Sec. 2.1, with $N = N_m + N_t$ sampled from the workload. All policies replay identical traffic at 1,024, 4,096, and 16,384 requests. SCT measures the network-side completion of a FastEPD service. Since it is not directly equivalent to end-to-end TTFT or TPOT, we further evaluate FastEPD's sensitivity to representative endpoint delay and ordering perturbations. [5]

Table 1 separates the physical DPDK prototype from the ns-3 scale tests (Appendix E). In the ns-3 topology, FastEPD scheduling is enabled at all switch egress ports (leaf and spine), while all policies use the same RDMA/RoCEv2 stack (Guo et al., 2016), ECMP routing (Hopps, 2000), and DCQCN congestion control (Zhu et al., 2015).

Table 1: Execution environments and network configurations.

Environment	Configuration
Platform	Ubuntu 22.04.5 (x86-64); dual Intel Xeon Platinum 8450H; 112 hardware threads; 128 GB RAM; four Broadcom BCM57414 10-GbE NICs
DPDK prototype	DPDK 26.03; 2E+2P+2D dumbbell with one switch; Ethernet/IPv4/UDP; 10 Gb/s links; 5 μs propagation; 10 μs endpoint period
ns-3 simulation	ns-3 3.18; 32 nodes (8E+12P+12D, 8 leaf/4 spine) and 128 nodes (32E+48P+48D, 16 leaf/8 spine); RDMA/RoCEv2, ECMP, and DCQCN; 10 Gb/s links; 5 μs propagation

Table 2: DeepSeek-V3 architecture and traffic-generation parameters.

Model	Scale / architecture	Traffic-generation configuration
DeepSeek-V3 (MoE)	61 transformer layers (58 MoE layers) ; hidden size 7,168; 128 attention heads; vocabulary 129,280; FFN size 2,048; 256 experts/layer (top-8)	Transformer blocks $L = 58$; batch size $B = 64$; embedding size $M = 7{,}168$; experts per layer $E = 256$; active experts $k = 8$; data-type size $\tau = 2$ bytes (FP16); input length $N = N_m + N_t$ sampled from the Azure LMM inference trace.

Baselines. We compare FastEPD with four baselines. FIFO and RSS are system-native baselines that retain arrival-order queueing or receive-side hash distribution, respectively. The other two baselines reuse identification and multi-queue settings in FastEPD, but replace its service-oriented scheduling schemes with common alternatives. Specifically, C-WFQ uses configured weights according to queue length, while C-TP uses type-priority scheduling principles, i.e., $F_{EP} \rightarrow F_{PP} \rightarrow F_{PD} \rightarrow F_{DD}$.

4.2 Experimental Results

Overall Performance on the DPDK prototype. Table 3 reports the DeepSeek-V3 results under increasing workload scale. FastEPD achieves the lowest median SCT for every service-size class and workload scale. Its overall reduction is 57.67–79.09% over FIFO and 41.02–68.10% over RSS.

[4] We also evaluate three open-weight backbones, i.e., Qwen3-Coder-480B-A35B (Qwen Team, 2025), GLM-4.6 (Z.ai, 2025), and Tencent-Hunyuan-Large (Sun et al., 2024), with detailed results in Appendix F.

[5] Apart from network communication time, TTFT/TPOT are also affected by endpoint batching schemes, KV cache management, and runtime scheduling, which are outside the scope of the DPDK prototype.

Furthermore, FastEPD's schedulers accelerate short services without starving longer services that require more packets to complete.

Ablation Study of the FastEPD Scheduling Scheme. C-TP and C-WFQ isolate the scheduling policy after FastEPD identification and multi-queue classification. FastEPD reduces overall SCT by 48.68–71.74%. Against C-WFQ, the margin falls from 47.01% to 4.61% as workload scale rises. Nevertheless, both FastEPD and C-WFQ outperform FIFO and RSS, showing the effectiveness of FastEPD's identification and classification.

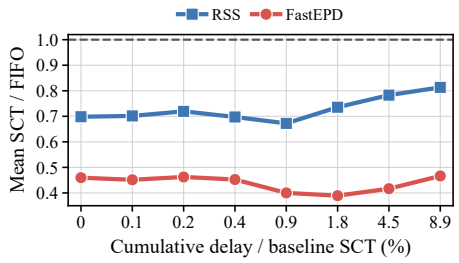
Table 3: DeepSeek-V3 SCT on the DPDK prototype. The left block shows median SCT, and the right block gives FastEPD's reduction from each baseline. Bold SCTs are lowest in their row.

Requests	Class	Median SCT (ms)					FastEPD SCT reduction vs.			
		FIFO	RSS	C-TP	C-WFQ	FastEPD	FIFO	RSS	C-TP	C-WFQ
1,024	Short	1,643	1,067	1,206	666	**247**	85.00%	76.90%	79.56%	62.99%
	Medium	1,643	1,089	1,227	560	**362**	77.96%	66.74%	70.48%	35.38%
	Long	1,644	1,099	1,237	866	**803**	51.16%	23.93%	35.08%	7.27%
	Overall	1,643	1,077	1,215	648	**343**	79.09%	68.10%	71.74%	47.01%
4,096	Short	6,725	4,661	5,346	2,921	**1,778**	73.56%	61.85%	66.74%	39.13%
	Medium	6,726	4,683	5,382	2,675	**1,857**	72.39%	60.34%	65.50%	30.58%
	Long	6,727	4,712	5,399	3,313	**3,240**	51.84%	31.24%	39.99%	2.20%
	Overall	6,726	4,673	5,359	2,879	**1,950**	71.01%	58.28%	63.62%	32.29%
16,384	Short	27,748	19,914	22,879	12,455	**11,503**	58.54%	42.24%	49.72%	7.64%
	Medium	27,798	19,942	22,948	11,966	**11,562**	58.41%	42.02%	49.62%	3.37%
	Long	27,799	20,005	22,945	13,851	**12,618**	54.61%	36.93%	45.01%	8.90%
	Overall	27,768	19,932	22,906	12,324	**11,756**	57.67%	41.02%	48.68%	4.61%

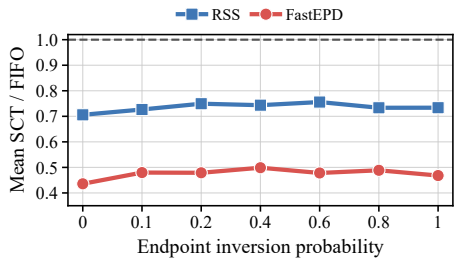
Processing Overhead of FastEPD. Table 4 measures per-packet queue-processing cost and aggregate throughput at 4,096 and 16,384 requests. FastEPD performs additional work to maintain semantic queues and service budgets, and therefore has higher per-packet processing time than FIFO and RSS. The results show that this cost does not cause throughput collapse. Specifically, at the higher load, FastEPD sustains 0.640 Mpps versus 0.642 Mpps for RSS, while completing the most requests per second at both scales. Thus, its lower SCT primarily comes from better completion ordering rather than greater raw forwarding capacity.

Table 4: Queue-processing cost and throughput. Lower processing time and higher throughput are better, and best values are bold.

Requests	Metric	Policy		
		FIFO	RSS	FastEPD
4,096	Queue Process Time (ns/packet)	**1,785**	1,990	2,456
	Total Packet throughput (Mpps)	0.396	**0.555**	0.549
	Request throughput (requests/s)	496	620	**694**
16,384	Queue Process Time (ns/packet)	**1,436**	1,649	1,767
	Total Packet throughput (Mpps)	0.489	**0.642**	0.640
	Request throughput (requests/s)	615	793	**806**



(a) Sensitivity to fixed endpoint processing delay.



(b) Sensitivity to endpoint flow-order perturbations.

Figure 8: Robustness to endpoint scheduling dynamics. The dashed line marks parity with FIFO.

Robustness to Endpoint Delay. We add a fixed 0–1,000 μs processing delay to every endpoint-dispatched flow. With the largest setting, the cumulative delay is 8.9% of baseline SCT. Figure 8a shows that FastEPD remains well below FIFO and RSS throughout the sweep. The mild non-monotonic response reflects changes in which flows overlap at the switch, but the persistent gap shows that FastEPD does not require idealized endpoint dispatch timing.

Robustness to Endpoint Perturbations. We fix the per-flow delay at 20 μs and sweep the probability of inverting nearby endpoint dispatches from 0 to 1. Specifically, the endpoint perturbations mean that the processing order of different services may be changed by the computation scheduling or KV cache management of GPU endpoints. Figure 8b shows that FastEPD remains below both baselines without monotonic degradation as perturbations increase. Packet tags preserve the scheduling semantics even when arrival order changes, so the gain does not depend on exact preservation of the idealized endpoint dispatch order.

Table 5: SCT in the 32- and 128-node ns-3 experiments. R_F and R_R are FastEPD's reductions relative to FIFO and RSS, respectively, computed as $1 - \mathrm{SCT}_{\mathrm{FastEPD}}/\mathrm{SCT}_{\mathrm{baseline}}$.

		Mean SCT (ms)			FastEPD reduction	
Nodes	**Class**	**FIFO**	**RSS**	**FastEPD**	R_F	R_R
32 (4,096 requests)	Short	5,201	4,664	**3,757**	27.75%	19.44%
	Medium	1,348	1,352	**1,192**	11.60%	11.88%
	Long	2,192	2,224	**1,923**	12.29%	13.53%
	Overall	3,415	3,153	**2,596**	23.97%	17.67%
128 (16,384 requests)	Short	21,981	20,099	**19,511**	11.24%	2.93%
	Medium	76,554	74,279	**71,451**	6.67%	3.81%
	Long	110,825	101,636	**94,461**	14.77%	7.06%
	Overall	75,000	67,764	**64,416**	14.11%	4.94%

Topology Scalability by ns-3 Simulation. In 32- and 128-node ns-3 topologies, FastEPD reduces overall mean SCT by 23.97%/17.67% relative to FIFO/RSS at 32 nodes and 14.11%/4.94% at 128 nodes. The gains of R_R diminish, as network-wide contention, link-level load imbalance, and traffic distribution account for more end-to-end SCT beyond switch queue scheduling. FastEPD remains operational and beneficial through fixed metadata, queues, and per-packet work while complementing other in-network mechanisms, including routing, congestion control, and traffic engineering.

5 Related Work

Disaggregated inference systems. EPD isolates encoding, prefill, and decoding (Singh et al., 2025). Specifically, DistServe and Splitwise optimize prefill-decode placement and provisioning (Zhong et al., 2024; Patel et al., 2024), and Mooncake improves KV-cache transfer and reuse (Qin et al., 2025). These systems determine where computation and state reside, whereas FastEPD schedules the resulting dependent flows inside the network.

Queue management & scheduling. Classical schemes target per-flow fairness (Shreedhar & Varghese, 1995), packet priorities (Nichols et al., 1998), or flow completion time (Alizadeh et al., 2013). Programmable data planes support richer line-rate processing (Bosshart et al., 2014; Sivaraman et al., 2016) but favor compact state. FastEPD jointly encodes service identity, EPD flow type, and service-size class to coordinate dependency-aware and load-adaptive queues.

6 Conclusion

In this work, we propose an in-network identification and scheduling framework, FastEPD, to accelerate LMM EPD inference by reducing the service completion time (SCT) for network transmissions. It mainly consists of lightweight multidimensional flow identification and service-oriented multi-queue scheduling. We implement FastEPD using a DPDK prototype and an ns-3 simulator driven by real model workloads, and experimental results show that it outperforms canonical network schemes across traffic and topology scales.

AI USE STATEMENT

In this work, we used generative AI tools to polish the writing, identify related work and design proper experiments (such as selecting models and inference trace). We did not use generative AI tools for research ideation, and the remaining required disclosure tasks are not applicable to this work. We have reviewed all AI-assisted work. For example, we checked the literature identified by generative AI and the accuracy of expressions polished by AI. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Section 3 specifies the packet-tag encoding, service-size classification, queueing rules, and online procedures, while Section 4 reports the Azure LMM trace source, model and traffic parameters, hardware and software environments, network configurations, baselines, workload scales, and SCT metric used in our DPDK and ns-3 evaluations. All compared policies replay the same traffic within each workload. Table A.1 provides the complete notation; Appendix B discusses candidate tag carriers; Appendix C gives the scheduling derivation; Appendix D documents the DPDK interfaces and scheduler state; Appendix E depicts the evaluation topologies; Appendix F reports the configurations and repeated-run aggregation for the additional LMMs; and Appendix G describes the stage-wise ns-3 slowdown methodology and results.

REFERENCES

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob L. Menick, Sebastian Borgeaud, Andy Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikołaj Bińkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karén Simonyan. Flamingo: A visual language model for few-shot learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/960a172bc7fbf0177ccccbb411a7d800-Abstract-Conference.html.

Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pp. 435–446. ACM, 2013. doi: 10.1145/2486001.2486031.

Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014. doi: 10.1145/2656877.2656890. URL https://www.cs.princeton.edu/~jrex/papers/P4-ccr14.pdf.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. URL https://arxiv.org/abs/2412.19437.

DPDK Project. *Data Plane Development Kit Programmer's Guide*. Linux Foundation, 2026. URL https://doc.dpdk.org/guides/prog_guide/index.html. Accessed 2026-08-29.

Wesley M. Eddy. Transmission control protocol (TCP). RFC 9293, Internet Engineering Task Force, 2022. URL https://www.rfc-editor.org/info/rfc9293.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL https://www.jmlr.org/papers/v23/21-0998.html.

Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitendra Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 202–215. ACM, 2016. doi: 10.1145/2934872.2934908. URL https://doi.org/10.1145/2934872.2934908.

Christian E. Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992, Internet Engineering Task Force, 2000. URL https://www.rfc-editor.org/info/rfc2992.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 34892–34916, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/6dcf277ea32ce3288914faf369fe6de0-Abstract-Conference.html.

Microsoft Azure. Azure LMM inference trace 2025. Azure Public Dataset, 2025. URL https://github.com/Azure/AzurePublicDataset/blob/master/AzureLMMInferenceDataset2025.md. One-week trace collected October 15–22, 2024; accessed 2026-09-25.

Kathleen Nichols, Steven Blake, Fred Baker, and David L. Black. Definition of the differentiated services field (DS Field) in the IPv4 and IPv6 headers. RFC 2474, Internet Engineering Task Force, 1998. URL https://www.rfc-editor.org/info/rfc2474.

NVIDIA Corporation. *Holoscan Sensor Bridge Documentation: Dataplane*. NVIDIA Corporation, 2026. URL https://docs.nvidia.com/holoscan/sensor-bridge/fpga-ip/dataplane. Accessed 2026-09-18.

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)*, pp. 118–132. IEEE, 2024. doi: 10.1109/ISCA59077.2024.00019.

Jon Postel. Internet protocol. RFC 791, Internet Engineering Task Force, 1981. URL https://www.rfc-editor.org/info/rfc791.

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)*, pp. 155–170. USENIX Association, 2025. URL https://www.usenix.org/conference/fast25/presentation/qin.

Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ramachandran Ramjee, and Rodrigo Fonseca. ModServe: Modality- and stage-aware resource disaggregation for scalable multimodal model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, pp. 817–830. ACM, 2025. doi: 10.1145/3772052.3772254. URL https://doi.org/10.1145/3772052.3772254.

Qwen Team. Qwen3-Coder: Agentic coding in the world. Qwen Blog, 2025. URL https://qwenlm.github.io/blog/qwen3-coder/. Accessed 2026-09-25.

Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. A remote direct memory access protocol specification. RFC 5040, Internet Engineering Task Force, 2007. URL https://www.rfc-editor.org/info/rfc5040.

M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 231–242. ACM, 1995. doi: 10.1145/217382.217453.

Gursimran Singh, Xinglu Wang, Yifan Hu, Timothy Yu, Linzi Xing, Wei Jiang, Zhefeng Wang, Xiaolong Bai, Yi Li, Ying Xiong, Yong Zhang, and Zhenan Fan. Efficiently serving large multimodal models using EPD disaggregation. *arXiv preprint arXiv:2501.05460*, 2025. URL https://arxiv.org/abs/2501.05460.

Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 15–28. ACM, 2016. doi: 10.1145/2934872.2934900. URL https://arxiv.org/abs/1512.05023.

Xingwu Sun, Yanfeng Chen, Yiqing Huang, Ruobing Xie, Jiaqi Zhu, Kai Zhang, Shuaipeng Li, et al. Hunyuan-Large: An open-source MoE model with 52 billion activated parameters by tencent. *arXiv preprint arXiv:2411.02265*, 2024. URL https://arxiv.org/abs/2411.02265.

Z.ai. GLM-4.6. Technical blog and official model release, 2025. URL https://z.ai/blog/glm-4.6. Accessed 2026-09-25.

Zeyu Zhang, Haiying Shen, Shay Vargaftik, Ran Ben Basat, Michael Mitzenmacher, and Minlan Yu. HACK: Homomorphic Acceleration via Compression of the Key-Value Cache for Disaggregated LLM Inference. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, pp. 1245–1247, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715242. doi: 10.1145/3718958.3750481. URL https://doi.org/10.1145/3718958.3750481.

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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 523–536. ACM, 2015. doi: 10.1145/2785956.2787484. URL https://doi.org/10.1145/2785956.2787484.

A Frequently Used Notations

Table A.1 summarizes the notation used throughout the paper. When a letter is overloaded, its meaning is determined by the context stated in the description.

Table A.1: Frequently Used Notations

Symbol	Description
LMM inference and EPD traffic	
E, P, D	Encoding, prefill, and decoding stages; in functional expressions, the corresponding stage operators. In Table 2, E also denotes the number of experts per MoE layer.
B	Number of requests (samples) grouped into one inference service, i.e., the mini-batch size.
i_m	Multimodal input tensor.
i_t	Text input tensor, with shape $B \times N_t \times M$.
v_t^e	Multimodal token tensor produced by the encoding stage, $v_t^e = E(i_m)$.
N_m	Number of multimodal input tokens per request (sample).
N_t	Number of text input tokens per request (sample).
M	Embedding size of one token.
I	Joint input token tensor formed by concatenating v_t^e and i_t.
$\mathrm{Concat}(\cdot,\cdot)$	Concatenation operator used to construct the joint input token tensor.
N	Per-request input length, $N = N_m + N_t$; the corresponding batched service size is $B(N_m + N_t)$.

Continued on the next page

Table A.1 (continued)

Symbol	Description
L	Number of transformer blocks in the LMM backbone. In the implementation-complexity statement, $L = 3$ denotes the number of length classes scanned by the scheduler.
k	Number of top-ranked experts selected for each token in MoE routing.
τ	Data-type size in bytes, e.g., 2 for FP16.
kv_1^p, o_1^p	Initial KV cache and first output token generated by the prefill stage.
kv_t^d, o_t^d	KV cache and output token at autoregressive decoding step t.
t	Autoregressive decoding-step index.
F_{EP}	EP-migration flow carrying the joint input tokens from encoding servers to prefill servers.
F_{PP}	Dispatch or combine flow generated by expert routing within the prefill stage.
F_{PD}	PD-migration flow carrying the KV cache and first output token from prefill servers to decoding servers.
F_{DD}	Dispatch or combine flow generated by expert routing within the decoding stage.
S_a, S_b	The two example batched LMM inference services in Fig. 2.
Flow identification and labeling (Algorithm 1)	
s	An inference service comprising a batch of requests; it is also used as a service index.
$\bar{s}$	Compact Service ID allocated to the request batch represented by service s and encoded in the packet tag.
f	Communication-flow type, one of F_{EP}, F_{PP}, F_{PD}, and F_{DD}.
r	Server role, i.e., encoding, prefill, or decoding.
$\mathcal{P}_f$	Set of packets belonging to flow f.
p	An individual packet; $p.\mathrm{tag}$ is its one-byte FastEPD scheduling tag.
c_s	Service-size class assigned to the batched service s.
S, M, L	Short, medium, and long service-size classes, respectively; here L is a class label rather than the number of transformer blocks.
N_m^s, N_t^s	Per-request multimodal-token and text-token dimensions used by the batch of service s.
l_s	Batched service size of service s, $l_s = B(N_m^s + N_t^s)$.
δ_{sm}	Threshold separating the short and medium length classes.
δ_{ml}	Threshold separating the medium and long length classes.
Queueing and service-oriented scheduling (Algorithm 2)	
q	Queue index.
Q_H	High-priority queue for ACK, PFC, QCN, and NACK control traffic.
Q_M	Queue for packets without a FastEPD tag; the subscript M here is a queue name, not the medium length-class label.
$Q_{f,c}$	FastEPD data queue indexed by flow type f and length class c.
p_q	Current packet backlog of queue q.
A_q	Set of active Service IDs observed in queue q during the current scheduling epoch.
$\hat{p}_q$	Estimated average number of packets per active service in queue q.
U	Total number of service units allocated to the queues of one flow group in a scheduling round; it is selected from the minimum service count across the four flow groups.
u_S, u_M, u_L	Service units allocated to the short-, medium-, and long-class queues, satisfying $u_S + u_M + u_L = U$.
p_S, p_M, p_L	Packet backlogs of the short-, medium-, and long-class queues in the flow group currently being scheduled.
g	Flow group visited by the scheduler, where $g \in \{F_{EP}, F_{PP}, F_{PD}, F_{DD}\}$.
c	Generic length-class index, where $c \in \{S, M, L\}$.
$\mathcal{C}_g$	Set of non-empty length classes in flow group g.
$p_{g,c}$	Packet backlog of the queue indexed by flow group g and length class c.
$\hat{p}_{g,c}$	Estimated mean packet count per active service in the queue indexed by g and c.
u_c	Service units allocated to active length class c in the current flow group.

Continued on the next page

Table A.1 (continued)

Symbol	Description
b_c	Positive packet budget dequeued from active length class c in the current scheduling round.

B Candidate Tag Carriers in Existing Protocol Headers

FastEPD requires one byte of endpoint-provided scheduling metadata. Besides the IPv4 ToS field used by our controlled DPDK prototype, a closed deployment can consider unused or alignment-related positions in existing protocol headers. Figures B.1 and B.2 redraw two candidate layouts. These positions are deployment-specific carriers rather than standards-compliant extension points: endpoints and switches must agree on the encoding, preserve the selected bits, and update all affected integrity checks.

InfiniBand Base Transport Header. The 12-byte BTH contains a six-bit reserved field next to FECN/BECN and a seven-bit reserved field next to the Acknowledge Request bit, as shown in Fig. B.1 (NVIDIA Corporation, 2026). The 13 reserved bits are sufficient for the eight-bit FastEPD tag while leaving FECN, BECN, and the Acknowledge Request bit unchanged. For example, a controlled implementation can split the tag across six bits in the second word and two bits in the third word. Because reserved BTH bits are not a negotiated metadata channel, this mapping requires a closed fabric with coordinated endpoints and switch parsers; unmodified devices should continue to transmit these bits as zero.

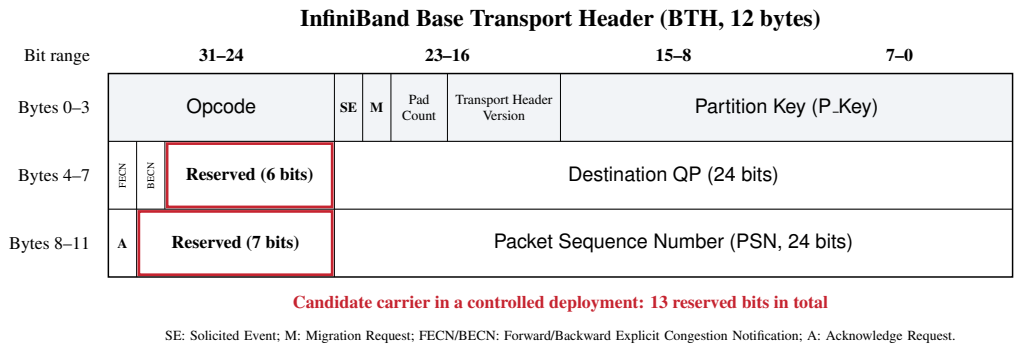


Figure B.1: Redrawn InfiniBand Base Transport Header (BTH). Red boxes mark the two reserved bit ranges that can carry a FastEPD tag only in a controlled, mutually configured deployment.

IPv4 header padding. The IPv4 Options area is variable length, and zero-valued Padding is added only when needed to align the end of the header to a 32-bit boundary (Postel, 1981). Consequently, the highlighted position in Fig. B.2 is not present in every packet and may range from zero to three octets. A controlled sender can construct an option layout that leaves at least one padding octet and place the FastEPD tag in that byte, provided every participating parser recognizes the convention. This is an experimental alternative to the ToS-based prototype mapping: standard IPv4 requires padding to be zero, and ordinary middleboxes may normalize or reject nonzero padding.

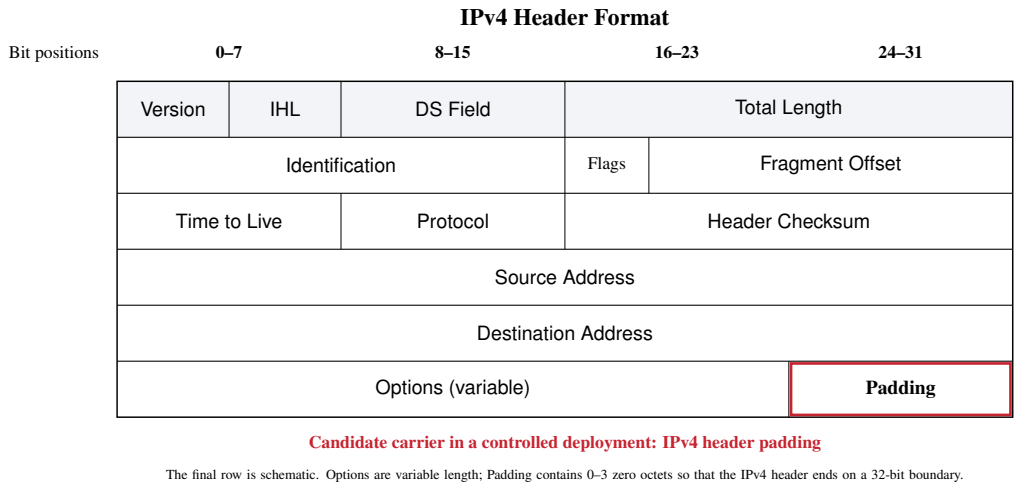


Figure B.2: Redrawn IPv4 header with standard English field names. The red box highlights the variable Padding area considered as a controlled-deployment carrier for the FastEPD tag.

C Closed-Form Solution and Scheduling Properties

This section gives the closed-form solution of Eqs. (2)– (3) and characterizes the regime in which FastEPD exhibits backlog-proportional, WFQ-like behavior. Consider one flow group g, and let $\mathcal{C}_g$ contain its non-empty length classes. For every $c \in \mathcal{C}_g$, the packet backlog satisfies $p_c > 0$ and the estimated packets-per-service value satisfies $\hat{p}_c > 0$.

C.1 Closed-Form Service-Unit Allocation

The ratio constraint in Eq. (3) implies that a common factor $\lambda > 0$ exists such that

$$u_c \hat{p}_c = \lambda p_c, \qquad c \in \mathcal{C}_g. \tag{C.1}$$

Substituting $u_c = \lambda p_c / \hat{p}_c$ into Eq. (2) gives

$$\lambda = \frac{U}{\sum_{j \in \mathcal{C}_g} p_j / \hat{p}_j}. \tag{C.2}$$

Therefore, the unique continuous solution for every non-empty class is

$$u_c^\star = \frac{U(p_c/\hat{p}_c)}{\sum_{j \in \mathcal{C}_g} p_j / \hat{p}_j}. \tag{C.3}$$

The corresponding unrounded packet budget is

$$\widetilde{b}_c = u_c^\star \hat{p}_c = \frac{U p_c}{\sum_{j \in \mathcal{C}_g} p_j / \hat{p}_j}. \tag{C.4}$$

Hence, within the selected flow group,

$$\frac{\widetilde{b}_c}{\widetilde{b}_j} = \frac{p_c}{p_j}, \qquad c, j \in \mathcal{C}_g. \tag{C.5}$$

Equation (C.5) follows directly from the load-balance constraint: before integer rounding and backlog capping, FastEPD allocates packet service in proportion to the current queue backlogs, independently of the absolute scale of the density estimates $\hat{p}_c$.

C.2 Active-Service Interpretation

Let n_c be the number of active batched services (equivalently, distinct active Service IDs) represented in class c. If the average-packets estimator is exact for the current backlog, i.e.,

$$\hat{p}_c = \frac{p_c}{n_c}, \tag{C.6}$$

then Eqs. (C.3)–(C.4) reduce to

$$u_c^\star = \frac{U n_c}{\sum_{j \in \mathcal{C}_g} n_j}, \tag{C.7}$$

$$\widetilde{b}_c = \frac{U p_c}{\sum_{j \in \mathcal{C}_g} n_j}. \tag{C.8}$$

Thus, the service-unit allocation is proportional to the number of active services, while the resulting packet budgets remain proportional to queue occupancy. This verifies the proposed property under the stated assumptions. If the estimator has relative error $\hat{p}_c = (p_c/n_c)(1+\epsilon_c)$, then

$$u_c^\star = \frac{U\, n_c/(1+\epsilon_c)}{\sum_{j \in \mathcal{C}_g} n_j/(1+\epsilon_j)}. \tag{C.9}$$

The estimator error perturbs the service-unit split, whereas the ideal packet budget ratio in Eq. (C.5) is still imposed by Eq. (3).

The implementation converts the continuous budget into an integer budget and prevents a non-empty class from receiving zero service:

$$b_c = \begin{cases} 0, & p_c = 0, \\ \min\Big(p_c, \max\Big(1, \mathrm{round}(\widetilde{b}_c)\Big)\Big), & p_c > 0. \end{cases} \tag{C.10}$$

When neither the one-packet floor nor the backlog cap is active, $|b_c - \widetilde{b}_c| \le 1/2$. Consequently, the relative rounding error vanishes as the budget grows, while the floor and cap deliberately cause deviations for very small or nearly drained queues.

C.3 Relation to C-WFQ at High Concurrency

The proportionality in Eq. (C.5) is best described as a backlog-proportional fluid allocation, or WFQ-like weighted service, rather than an exact reduction to classical WFQ. Classical WFQ uses configured weights and virtual finish times; FastEPD instead recomputes endogenous weights from current queue state, retains the fixed flow-group polling order, and applies integer budgets with positive-service protection.

At high concurrency, however, the length-class queues remain persistently backlogged, the class mixture varies more slowly, and the budgets are large enough that the one-packet floor, backlog cap, and rounding error are less influential. The ratios p_c/p_j therefore change slowly across successive rounds, so FastEPD approaches a stable weighted-fair service regime similar to the C-WFQ baseline. This provides a mechanism-level explanation for Table 3: the overall FastEPD reduction relative to C-WFQ decreases from 47.01% at 1,024 requests to 32.29% at 4,096 requests and 4.61% at 16,384 requests. The property explains the convergence trend, but does not imply identical SCT because the two policies still differ in polling order, state updates, and behavior when queues become empty or budgets are capped.

D DPDK Interfaces and Scheduler State

Table D.1 maps the endpoint–switch–endpoint pipeline to the concrete DPDK interfaces in Fig. 7 and summarizes the queue state used by FastEPD. Public DPDK interfaces and types follow the DPDK

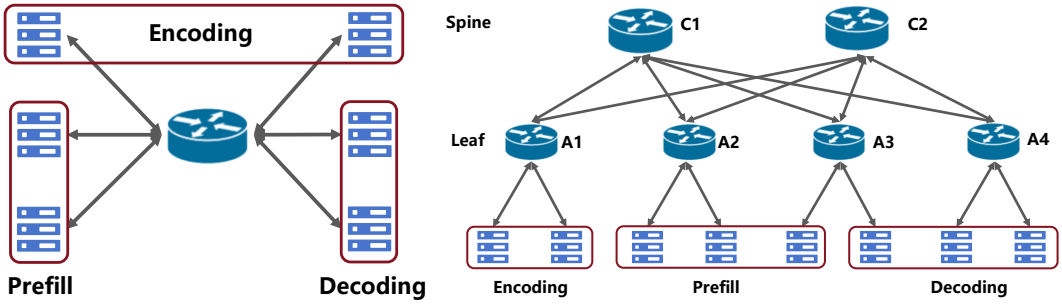
Programmer's Guide (DPDK Project, 2026); the two `Fastepd_queue_*` routines are prototype-specific wrappers. In particular, `rte_udp_hdr`, `rte_ipv4_hdr`, and `rte_mbuf` are C structure types rather than callable functions. The legacy member names shown in parentheses come from the original queue-state description; the DPDK prototype maintains equivalent per-queue state even when its concrete structure names differ.

Table D.1: DPDK interfaces and FastEPD queue-state definitions.

Pipeline stage / state	**Interface or representation**	**FastEPD role and update rule**
Packet-path interfaces		
Sending endpoint	Traffic simulator and custom FastEPD endpoint process	Generates the real-workload flow sequence, allocates one Service ID per request batch, derives the flow type and service-size class from $B(N_m + N_t)$, and encodes the one-byte tag. These are application routines, not public DPDK APIs.
Header construction	`struct rte_mbuf`; `struct rte_ipv4_hdr`; `struct rte_udp_hdr`; `rte_ipv4_cksum()`	Stores the Ethernet/IPv4/UDP packet in an mbuf. The controlled prototype writes the FastEPD tag to `rte_ipv4_hdr.type_of_service` and updates the IPv4 checksum in software or through NIC offload.
NIC receive	`rte_eth_rx_burst()`	Retrieves a burst of received packets from an Ethernet RX queue as an array of `rte_mbuf` pointers. The returned burst size bounds the work in one run-to-completion iteration.
Packet parser	`rte_pktmbuf_read()` and fixed header offsets	Reads Ethernet, IPv4, and UDP headers even when an mbuf is segmented; validates header lengths and extracts the tag. It returns an untagged or malformed outcome when the expected FastEPD metadata is unavailable.
L2 forwarding	Prototype MAC-learning / output-port lookup	Selects the output port independently of FastEPD scheduling, which does not alter routing, ECMP, or congestion-control state.
Classify and enqueue	`Fastepd_queue_enqueue()`; `rte_ring_enqueue_burst()`	Maps ACK/PFC/QCN/NACK traffic to Q_H, untagged traffic to Q_M, and a tagged packet to $Q_{f,c}$. Enqueues the mbuf pointer into the selected ring and updates its backlog and observed-service state.
Multi-queue storage	`struct rte_ring` handles for Q_H, Q_M, and $Q_{f,c}$	Implements two special queues and four flow groups, each with three length classes. The ring stores mbuf pointers; the FastEPD state is maintained in compact counters and service-ID records alongside the ring.
Dequeue scheduler	`Fastepd_queue_dequeue()`; `rte_ring_dequeue_burst()`	Serves non-empty Q_H and then Q_M, followed by $F_{EP} \rightarrow F_{PP} \rightarrow F_{PD} \rightarrow F_{DD}$. Within a group it visits S/M/L queues, computes the packet budgets from Eqs. (2)–(3), and dequeues at most the active budget.
NIC transmit	`rte_eth_tx_burst()`	Submits the scheduled mbuf array to the selected Ethernet TX queue. The return value is checked so unsent packets can be retried or released under the prototype's configured policy.
Receiving endpoint	DPDK public packet/header interfaces and custom endpoint process	Parses the forwarded packet, updates per-flow completion state, and triggers the next dependent EPD flow after its predecessor completes.
FastEPD queue and scheduler state		
Queue handle	$Q_H, Q_M, Q_{f,c}$	Identifies the control queue, default untagged queue, or tagged data queue indexed by flow type f and length class c.
Packet backlog	p_q (`m_queue_pkts[q]`)	Packets currently resident in queue q; incremented after a successful enqueue and decremented after a successful dequeue. It supplies the occupancy term in Eq. (3).
Byte backlog	`m_queue_bytes[q]`	Bytes currently resident in queue q; updated with packet length on enqueue and dequeue. It supports monitoring and capacity checks; scheduling uses packet counts.
Observed batched services	A_q (`m_arrival_tasks[q]`; legacy member name)	A set or compact record of batch-level Service IDs observed for queue q in the current measurement epoch. It is updated on tagged enqueue and reset or retired at the configured epoch/service boundary.
Arrival count	`m_arrival_pkts[q]`	Number of packets observed for queue q during the measurement epoch; incremented on enqueue and used with the observed-service count to update the density estimate.
Density estimate	$\hat{p}_q$	Mean packet count per observed batched service. The legacy implementation uses the epoch packet count divided by the number of observed Service IDs, with a positive lower bound and a backlog-based fallback when no new arrival is observed.
Service-unit budget	U and u_c	U is the group-level service-unit budget; u_c is the transient share assigned to an active S/M/L class by Eq. (C.3). Both are recomputed from queue state rather than stored per packet.
Packet budget	b_q (`m_dequeue_NPkts[q]`)	Remaining packets that may be removed from queue q in the current visit. It is initialized using Eq. (C.10), decremented after each successful dequeue, and refreshed at the next scheduling round.
Polling cursor	Current flow-group and length-class indices	Records the scheduler's position in the fixed group order and S/M/L order. Empty queues are skipped; exhausting b_q advances the class cursor, and completing the active classes advances the group cursor.
Service-ID lifetime	Active-ID reference/retirement state	Prevents reuse of a four-bit Service ID while packets carrying that ID remain represented in any queue state. The ID becomes reusable after the corresponding request-batch service state retires.

E EXPERIMENTAL TOPOLOGIES

Figure E.1 illustrates the two network topologies used in our evaluation. The dumbbell topology places the encoding, prefill, and decoding endpoints around a single bottleneck switch, matching the DPDK prototype. The spine–leaf topology distributes these endpoints across leaf switches connected through the spine layer, matching the topology-scale ns-3 evaluation. FastEPD scheduling is applied at the evaluated switch egress queues; endpoint multiplicities are abstracted in the diagrams for clarity.



(a) Dumbbell topology.

(b) Spine–leaf topology.

Figure E.1: Network topologies used for the DPDK prototype and ns-3 simulations.

F RESULTS ON OTHER MOE BACKBONES

We further evaluate FastEPD with three recent open-weight LMM backbones. All models use the same request arrivals, modality mix, sequence-length multiset, and length ordering for a given request scale. We retain the DPDK setup in the main experiment (2E+2P+2D endpoints, 10 Gb/s links, and batch size $B = 64$) and change only the model-derived traffic parameters. Each model–scale–policy configuration is repeated four times with a balanced execution order. The sampled requests are grouped into FastEPD services of B requests; therefore, $N = N_m + N_t$ denotes per-request input length, whereas the S/M/L classification uses the batched service size $B(N_m + N_t)$. Table F.1 summarizes the model configurations and the resulting traffic profiles.

Table F.1: Architectures and traffic-generation parameters for the additional LMMs. Here N is the per-request input length; FastEPD classifies the batched service size $B(N_m + N_t)$.

Model	**Scale / architecture**	**Traffic-generation configuration**
Qwen3-Coder-480B-A35B (MoE)	62 mapped MoE layers; hidden size 6,144; 96 attention heads; 160 experts/layer (top-8)	Transformer blocks $L = 62$; batch size $B = 64$; embedding size $M = 6{,}144$; experts per layer $E = 160$; active experts $k = 8$; data-type size $\tau = 2$ bytes (FP16); input length $N = N_m + N_t$ sampled from the workload trace
GLM-4.6 (MoE)	89 mapped MoE layers; hidden size 5,120; 96 attention heads; 160 experts/layer (top-8)	Transformer blocks $L = 89$; batch size $B = 64$; embedding size $M = 5{,}120$; experts per layer $E = 160$; active experts $k = 8$; data-type size $\tau = 2$ bytes (FP16); input length $N = N_m + N_t$ sampled from the workload trace
Tencent-Hunyuan-Large (MoE)	64 mapped MoE layers; hidden size 6,400; 80 attention heads; 16 experts/layer (top-1)	Transformer blocks $L = 64$; batch size $B = 64$; embedding size $M = 6{,}400$; experts per layer $E = 16$; active experts $k = 1$; data-type size $\tau = 2$ bytes (FP16); input length $N = N_m + N_t$ sampled from the workload trace

Table F.2 follows the presentation of Table 3, but reports only FIFO, RSS, and FastEPD because the two scheduler ablations are outside the scope of this cross-model test. Each SCT entry is the median

of the four run-level mean SCTs. The reduction columns report FastEPD's SCT reduction relative to FIFO and RSS, respectively.

Table F.2: SCT of additional LMMs on the DPDK prototype. The Requests column gives the raw input-request count, and Class denotes the batched service-size class. The left block reports median SCT, and the right block reports FastEPD's reduction over FIFO and RSS. Bold SCTs are lowest in their row.

Requests	Class	Median SCT (ms)			FastEPD SCT reduction	
		FIFO	RSS	FastEPD	over FIFO	over RSS
Qwen3-Coder-480B-A35B						
1,024	Short	1,100	835	**666**	39.48%	20.28%
	Medium	1,104	843	**676**	38.80%	19.88%
	Long	1,104	833	**693**	37.19%	16.78%
	Overall	1,102	838	**672**	39.05%	19.81%
4,096	Short	4,564	3,876	**2,759**	39.54%	28.81%
	Medium	4,564	3,899	**2,779**	39.11%	28.73%
	Long	4,566	3,909	**2,810**	38.45%	28.10%
	Overall	4,564	3,886	**2,770**	39.30%	28.71%
16,384	Short	19,187	16,656	**12,124**	36.81%	27.21%
	Medium	19,196	16,683	**12,142**	36.75%	27.22%
	Long	19,240	16,713	**12,205**	36.56%	26.97%
	Overall	19,194	16,670	**12,138**	36.76%	27.19%
GLM-4.6						
1,024	Short	1,034	824	**655**	36.68%	20.47%
	Medium	1,037	832	**665**	35.91%	20.11%
	Long	1,037	834	**674**	34.99%	19.19%
	Overall	1,035	827	**660**	36.28%	20.23%
4,096	Short	4,275	3,718	**2,719**	36.40%	26.87%
	Medium	4,275	3,743	**2,733**	36.06%	26.98%
	Long	4,277	3,744	**2,749**	35.72%	26.56%
	Overall	4,275	3,728	**2,727**	36.22%	26.86%
16,384	Short	17,965	16,290	**12,011**	33.14%	26.27%
	Medium	17,970	16,307	**12,021**	33.10%	26.28%
	Long	18,014	16,317	**12,068**	33.01%	26.04%
	Overall	17,971	16,298	**12,020**	33.12%	26.25%
Tencent-Hunyuan-Large						
1,024	Short	946	738	**562**	40.60%	23.87%
	Medium	951	746	**571**	39.94%	23.45%
	Long	951	749	**586**	38.37%	21.70%
	Overall	948	741	**567**	40.18%	23.52%
4,096	Short	3,965	3,545	**2,472**	37.66%	30.28%
	Medium	3,965	3,571	**2,485**	37.32%	30.41%
	Long	3,967	3,584	**2,518**	36.53%	29.74%
	Overall	3,965	3,557	**2,480**	37.45%	30.26%
16,384	Short	17,068	14,946	**10,605**	37.87%	29.05%
	Medium	17,075	14,964	**10,610**	37.86%	29.09%
	Long	17,118	14,985	**10,665**	37.69%	28.83%
	Overall	17,075	14,955	**10,612**	37.85%	29.04%

FastEPD achieves the lowest median SCT in every model–scale–class row. Across the nine overall rows, it reduces SCT by 33.12–40.18% relative to FIFO and by 19.81–30.26% relative to RSS. The result is also consistent across service-size classes: FastEPD reduces SCT in all 27 class rows by 33.01–40.60% over FIFO and by 16.78–30.41% over RSS. The margin over RSS is larger at higher concurrency: it is 19.81–23.52% for 1,024 requests and 26.25–30.26% for 4,096 and 16,384 requests. These results show that the benefit of FastEPD's semantic classification and service-aware scheduling persists across the three tested MoE architectures, service-size classes, and load levels, although its magnitude remains model- and scale-dependent.

G Stage-Wise Flow Slowdown in ns-3

We decompose the 32- and 128-node ns-3 traces by the four communication stages F_{EP}, F_{PP}, F_{PD}, and F_{DD}. For each completed flow f, slowdown is

$$\text{Slowdown}_f = \frac{\text{FCT}_f^{\text{real}}}{\text{FCT}_f^{\text{ideal}}}. \tag{G.1}$$

We match flows across FIFO, RSS, and FastEPD by simulator task identifier (used only for cross-policy trace matching and distinct from FastEPD's batch-level Service ID), source, destination,

payload size, ideal FCT, and occurrence order among otherwise identical flows. This yields 924 and 26,792 common logical flows in the 32- and 128-node traces, respectively.

Table G.1 reports the mean and P95 slowdown in each stage.

Table G.1: Stage-wise flow slowdown on logical flows shared by FIFO, RSS, and FastEPD. Bold values are lowest in their row for the corresponding statistic.

Nodes	Stage	Flows	Mean slowdown			P95 slowdown		
			FIFO	RSS	FastEPD	FIFO	RSS	FastEPD
32	F_{EP}	32	4.648	4.534	**4.083**	10.178	10.138	**10.118**
	F_{PP}	496	3.915	3.924	**3.444**	7.958	8.108	**6.793**
	F_{PD}	28	3.181	3.123	**2.752**	6.258	6.683	**5.771**
	F_{DD}	368	1.291	1.156	**1.131**	2.228	1.725	**1.317**
128	F_{EP}	692	3.245	3.033	**3.010**	8.798	8.796	**8.771**
	F_{PP}	21,120	2.087	1.849	**1.842**	4.088	3.433	**3.422**
	F_{PD}	756	4.220	2.386	**2.263**	5.346	5.286	**4.609**
	F_{DD}	4,224	1.036	1.027	**1.009**	1.114	1.037	**1.034**

FastEPD has the lowest mean and P95 slowdown in every stage at both topology sizes. At 32 nodes, its mean slowdown is 12.03–13.49% below FIFO and 2.16–12.23% below RSS across the four stages; the corresponding P95 reductions are 0.59–40.89% and 0.20–23.65%. At 128 nodes, the improvement over FIFO remains visible in every stage, with mean/P95 reductions of 2.61–46.37%/0.31–16.29%. The margins over RSS are smaller in F_{EP}, F_{PP}, and F_{DD}, but are still positive; the clearest separation is in F_{PD}, where FastEPD reduces mean slowdown by 5.16% and P95 slowdown by 12.81%. Meanwhile, F_{PP} accounts for 21,120 of the 26,792 matched flows, and FastEPD remains slightly below RSS while reducing mean/P95 slowdown by 11.74%/16.29% relative to FIFO.