A Generic and Efficient Communication Framework for Message-level In-Network Computing

Xinchen Wan[1] Luyang Li[2,3] Han Tian[4] Xudong Liao[1] Xinyang Huang[1] Chaoliang Zeng[1] Zilong Wang[1]
Xinyu Yang[1] Ke Cheng[5] Qingsong Ning[6] Guyue Liu[7] Layong Luo[5] Kai Chen[1]
[1]*iSINGLab, Hong Kong University of Science and Technology* [2]*ICT,CAS* [3]*UCAS*
[4]*USTC* [5]*Unaffiliated* [6]*ByteDance* [7]*Peking University*

***Abstract*—Message-level in-network computing (MINC) emerges as a promising hardware acceleration method that utilizes accelerators to offload message-level computation and enhance application performance in the datacenter. However, the development of MINC applications is challenging in the communication aspect due to poor portability and under-utilized resource.**

In this paper, we present LEO, a generic and efficient communication framework for MINC. LEO facilitates portability across both application and hardware via introducing a *communication path* abstraction, which is capable of describing generic applications with predictable communication performance across diverse hardware. It further incorporates a built-in multi-path communication over CPU and accelerator to enhance communication efficiency. We have implemented a prototype of LEO and evaluated it with four case studies on testbeds covering FPGA-based, SoC-based smartNICs and GPU. Experiments show that LEO achieves genericity and efficiency across MINC applications, yielding 1.2–4.7× speedup over baselines with negligible overhead.

***Index Terms*—In-network computing, networking hardware**

I. INTRODUCTION

In-Network Computing (*INC*) has emerged as a promising technique to mitigate the gap between high-speed network bandwidth and stagnant CPU compute capability. By offloading partial or entire computation tasks to accelerators or switches located along the datapath, INC alleviates the processing load on the CPU, reduces communication cost, and enhances overall computation throughput.

INC can be categorized into two types: packet-level INC (*PINC*) and message-level INC (*MINC*), based on whether the computation occurs on packets (below the transport layer), or on messages (above the transport layer), as depicted in Figure 1. PINC, which facilitates computation over packets, is primarily executed by in-network devices such as bump-in-the-wire (BITW) smartNICs and programmable switches for datacenter applications such as in-network aggregation [19], [33] and network management [13]. MINC, which operates on messages, is executed by look-aside accelerators such as GPUs, FPGAs, and smartNICs. Owing to the prosperity of applications at end-host, MINC is deployed in multiple scenarios in the datacenter, including key-value store (KVS) [20], distributed file system (DFS) [7], elastic block storage (EBS) [30], and distributed machine learning (DML) [15], [37], etc. Despite the extensive research on PINC application development [14], [43], [47], the exploration in the realm of MINC has garnered less attention, given its wide applicability and potential for performance improvement.

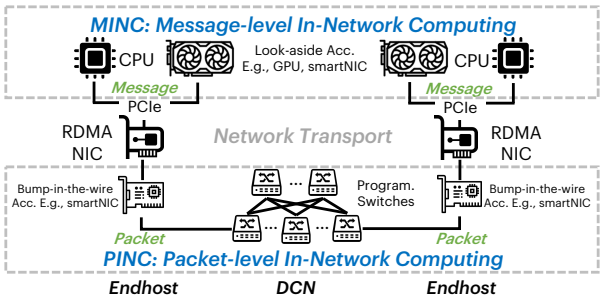


Fig. 1: Categorization of In-Network Computing. The network stack in general can be classified into two domains: the message domain above transport, and the packet domain below transport. In this paper, the in-network computing within the message domain (*e.g.*, key-value store), is called message-level INC, while that within the packet domain (*e.g.*, packet encryption), is called packet-level INC.

The development of MINC applications contains two major aspects: computation and communication. For computation, the task is relatively straightforward, entailing the independent programming of hardware acceleration functions in distinct accelerators. However, communication presents a more intricate challenge, as it involves the effective organization and implementation of dataflows among devices, thereby demanding advanced system expertise from the application developers[1]. We reveal two critical problems encountered in MINC communication development as follows:

#1: Poor portability across applications and hardware. When porting MINC applications to various hardware, the issue of poor portability becomes particularly evident. Notably, there have been several MINC applications such as KVS offloaded to different hardware, *e.g.*, FPGA [20] or SoC-based smartNICs [25], [41]. However, adapting application dataflows between these hardware while maintaining high communication efficiency demands substantial engineering efforts. These efforts include, but are not limited to, the reimplementation of dataflows due to different communication APIs, and the restructure of offloading strategies due to hard-

[1]In the rest of the paper, we refer to the application developers as developers for short.

ware performance variances, etc. Experienced engineers have estimated that such porting could take from weeks to months to complete, highlighting the significance of portability.

#2: Under-utilized resources in existing MINC systems. In MINC context, both CPU and accelerator resources are available for use, providing an inherent opportunity to optimize performance by employing these resources simultaneously. Unfortunately, current MINC systems [20], [30], [34] tend to neglect this opportunity, favoring instead to offload tasks to accelerators as much as possible. Such neglect results in the loss of computational resources in CPUs and communication resources available in CPU-NIC links.

Given these problems, we argue the need for a new framework that supports portability across applications and hardware, and incorporates hybrid resource utilization. With such a framework, developers would be able to offload routine communication tasks to the framework and focus on the application-specific logic.

In this paper, we present LEO, a generic and efficient MINC communication framework. Specifically, LEO makes the following notable contributions:

- **Generic abstractions for application and hardware.** To facilitate portability, LEO proposes generic abstractions for both application and hardware. Specifically, LEO introduces a *communication path* abstraction to accommodate diverse communication requirements of applications with predictable communication performance across hardware. Utilizing this abstraction, application dataflows can be conceptually represented as a sequence of paths with predictable communication performance, and hardware is abstracted into different models based on its topology and associated pre-profiled performance metrics, hence addressing **#1**.
- **Built-in multi-path optimization.** LEO integrates built-in multi-path communication optimization to efficiently utilize resources in both CPU and accelerator. We formulate the multi-path selection as a linear programming (LP) problem, which is solvable optimally using classical LP solvers [29]. To facilitate real-time path selection, we propose a heuristic algorithm that first generates a path selection priority list offline and then selects the appropriate path online based on the monitored intra-host job completion times (JCTs) of previous requests, reflecting the current path status, thereby addressing **#2**.

We have implemented a prototype of LEO and evaluated it with four representative case studies on three testbeds covering devices including FPGA-based and SoC-based smartNICs and GPU. Our experimental results confirm that LEO is generic and maintains high performance for both applications and accelerators with negligible overhead. Specifically, LEO provides 1.2-4.7× higher speedup than baselines in four case studies. Furthermore, the heuristic multi-path solution employed by LEO yields 1.18–2.17× better throughput than two greedy algorithms, and delivers comparable performance to the optimal solution while satisfying the real-time selection demand. Finally, the overhead of LEO is negligible, accounting for 10.3% of CPU utilization and 1.5µs for path selection.

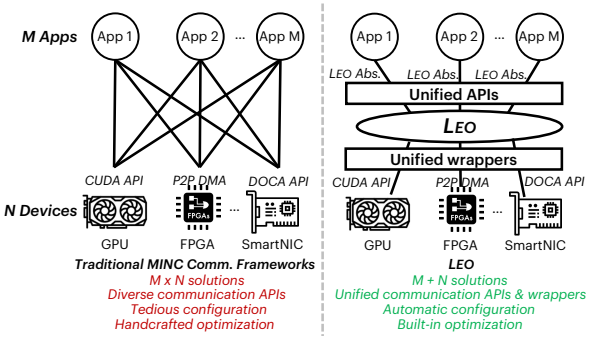


Fig. 2: Comparison between traditional MINC communication frameworks and LEO.

Figure 2 summatively compares LEO against traditional MINC communication frameworks. In traditional frameworks, supporting M dataflows over N hardware types necessitates developers to handcraft M×N solutions, each reliant on vendor-specific APIs. This process is further complicated by the need for tedious and error-prone direct data transmission configuration, as well as handcrafted communication optimization strategy. In contrast, LEO introduces expressive abstractions along with unified APIs and wrappers that simplify the programming and configuration processes for both application and hardware. This effectively reduces the development efforts to M+N solutions. Furthermore, LEO incorporates a built-in multi-path communication to utilize both CPU and accelerator.

II. Background and Motivation

We first provide the background of message-level in-network computing (MINC), and then discuss the problems of MINC application development.

A. Category of In-Network Computing

In-Network Computing (*INC*) is a class of hardware acceleration approaches that offloads part of or whole computation on the network datapath in order to free up CPU cycles, reduce communication overhead, and boost application performance [16], [17], [19], [20], [22], [23], [27], [33], [34].

Figure 1 shows the categorization of INC. Based on whether the computation occurs on packets (below the transport layer) or on messages (above the transport layer), INC can be further categorized into packet-level INC (*PINC*) and message-level INC (*MINC*). PINC processes computation over packets by programmable devices within the network, *e.g.*, bump-in-the-wire (BITW) accelerators [13] and programmable switches [1], with each packet size restricted within Maximum Transmission Unit (MTU), *e.g.*, 1500 B. On the other hand, MINC performs computation over messages by look-aside accelerators at end-host such as GPUs, FPGAs, and smartNICs, with the message size ranging from several bytes to 10s of GB.

The differences in the size of data processing unit and the location of accelerators result in distinct supported applications and developers: PINC favors boosting those applications whose semantics can be expressed within a packet size, *e.g.*, in-network aggregation [19], [26], [33] and in-network

cache [17], [27], and is maintained mostly by infrastructure maintainers; MINC, however, is prosperous in boosting applications that operate on a collection of data, including key-value stores (KVS) [20], [34], file compression and decompression in distributed file system (DFS) [7], metadata operations in elastic block storage (EBS) [30], and collective communication in distributed machine learning (DML) [15], [35], [36], [38], etc., thanks to its wide range size, and is typically operated by application developers. While there has been considerable attention towards PINC frameworks in recent years [14], [21], [43], [47], frameworks for developing MINC applications have not received equal focus, despite their wide applicability and the potential for substantial performance improvements.

B. Problems of MINC Application Development

The development of MINC applications necessitates efforts in both computation and communication aspects. For the computation aspect, the developers typically engage in the programming of hardware acceleration functions for each device independently. Thought time-consuming, this process is relatively easy to handle. The communication aspect, however, poses greater difficulties. It demands not only effective coordination among various devices but also the efficient implementation of dataflows for specific application, making the communication aspect more challenging to handle.

We discuss two problems of MINC application development caused by the communication aspect using examples of state-of-the-art solutions.

Poor portability across applications and hardware. Portability is of paramount importance in application development, particularly given the diversities both in upper-layer dataflows and lower-layer hardware platforms. Notably, multiple MINC applications such as KVS have been successfully offloaded to different kinds of hardware platforms including FPGA-based [20] and SoC-based smartNICs [25], [41]. This process, however, obligates developers to reimplement application dataflows between devices using hardware-specific APIs and communication modules, leading to increased yet redundant development efforts. For instance, developing and testing dataflows with Register Transfer Level (RTL) [11] for FPGA-based devices can consume as much as two months for proof-of-concept (PoC), while the time consumed to program C over SoC-based smartNICs may be reduced to around two weeks[2]. Therefore, the task of porting M dataflows to N hardware platforms may require around $1.25\times M\times N$ months for developing, posing substantial burdens for developers. A more efficient approach is to develop M distinct dataflows and N hardware solutions independently, followed by integration through an intermediate layer. This manner effectively reduces the development time to $1.25\times(M+N)$, resulting in considerable time savings, *e.g.*, a reduction of 8.75 months in the scenario where M is 5 for the dataflow types in KVS [41] and N is 3 as the number of hardware models.

[2]These development time were estimated by experienced FPGA and SoC engineers in a major company.

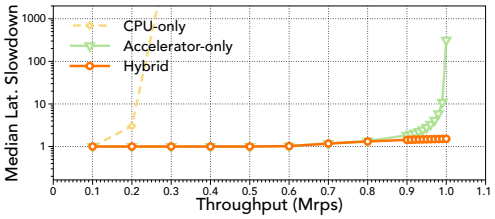


Fig. 3: The performance improvement with hybrid dataflows optimization.

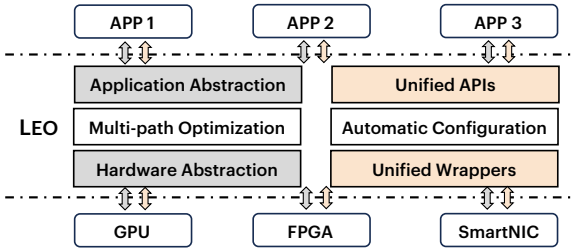


Fig. 4: LEO overview.

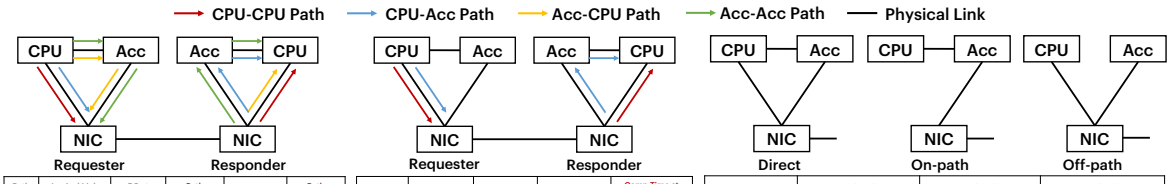
Under-utilized resources in existing MINC systems. Existing MINC systems do not maximally utilize the available resources in the hardware platform. Specifically, traditional MINC systems typically use the accelerator in the datapath for fast processing, but ignore the usage of available host CPU resources[3] [20], [30], [34]. That is, both the CPU resources at endhost and the CPU-NIC links are unused during processing, exhibiting an optimization potential of hybrid solution to embrace both resources. To reveal this potential, we conduct an experiment to emulate the processing of a KVS application at the server which is equipped with one CPU and one smartNIC. We reuse the throughput results reported in [34], *i.e.*, 23.0Mops for 16-core CPU and 71.8Mops for 24-core smartNIC. For the hybrid solution, we randomly select the dataflow for each request during processing. The result shown in Figure 3 reveals that the hybrid solution performs significantly better than other single-path settings owing to its better utilization of resources. Such result indicates that a middle-layer framework with hybrid dataflow optimization can demonstrate better performance for MINC applications.

III. LEO DESIGN

We aim to design a middle-layer communication framework for MINC applications that addresses the above problems simultaneously. Specifically, this framework should achieve the following design goals:

- *Generic.* The framework should be generic to support multiple kinds of dataflows and diverse hardware platforms. This genericity enables the portability across applications and hardware, and hence reduces the development workload for developers.
- *Efficient.* The communication performance this framework provides should be as efficient as possible. Specifically, the

[3]iPipe [25] supports hybrid usage of host CPU and accelerator, but its approach is inefficient when network fluctuates, as revealed in §V-B.



Path Type	Logical Link Seq	DS at each link	Path Bandwidth	Path MPS	Path Latency
C-C	[CN, NC]	[f_w, f_R]	min(link_bw)	min(link_mps)	sum(link_lat)
C-A	[CN, NA]	[f_w, f_d]	min(link_bw)	min(link_mps)	sum(link_lat)
A-C	[CA, AN, NC]	[f_w, f_t, f_R]	min(link_bw)	min(link_mps)	sum(link_lat)
A-A	[CA, AN, NA]	[f_w, f_t, f_t]	min(link_bw)	min(link_mps)	sum(link_lat)

Req Type	Req Dep	Path List	DS at each link	Comp Time at device
GET	(PUT, ADDR)	[C-A,C-C]	[[8, 8], [64, 64]]	[[0, 3], [3.3, 0]]
GET	(PUT, ADDR)	[C-C,C-C]	[[8, 8], [64, 64]]	[[0, 3.3], [3.3, 0]]
PUT	NONE	[C-C,C-C]	[[8, 8], [64, 64]]	[[0, 3.3], [3.3, 0]]

	Logical Link CA	Logical Link CN	Logical Link AN
Physical Link	[C-A]	[C-N]	[A-N]
Bandwidth	90.8 Gbps	90.8 Gbps	106.8 Gbps
MPS	149.34 Mmps	149.34 Mmps	175 Mmps
Latency	1000 ns	890 ns	702 ns

(a) Communication path. (b) Application abstractions. (c) Hardware abstractions.

Fig. 5: MINC communication abstractions. The blue features should be provided by developers, and the red features can be automatically profiled by LEO.

framework should utilize the resources both in CPU and accelerator with high efficiency.

Figure 4 shows an overview of LEO. LEO proposes the abstractions of application and hardware for describing the diversities of both sides (§III-A). For efficiency, LEO designs a built-in multi-path communication optimization for hybrid resource utilization (§III-B). Besides, LEO also provides unified APIs and wrappers to developers (§III-C) and automatic communication configuration (§IV) for ease of programming and freeing developers from complex configuration.

A. MINC Communication Abstractions

In this section, we start by introducing the abstraction of the *communication path*, which can describe diverse dataflows and estimate their performance based on the attached hardware transmission features provided by the lower layer. We then elaborate on the application and hardware abstractions.

Communication path. The communication path describes the logical links traversed by the request from the requester to the responder, above the transport layer, with each path associated with specific hardware transmission features. As shown in Figure 5a, there are four types of communication paths in MINC communication, *i.e.*, CPU-CPU path, CPU-accelerator path, accelerator-CPU path, and accelerator-accelerator path. The features of each path include:

- *Logical link sequence*: This describes the logical link sequence traversed by the request from the requester to the responder from the perspective of developer. For example, the CPU-CPU path sequentially traverses CN (CPU-NIC) at requester and NC (NIC-CPU) at responder.
- *Data size at each link*: This feature describes the size of data traversed at each link during request processing. This value is defined by developers to be elaborated in the application abstractions.
- *Estimated path performance*: These features are determined by metrics measured on the hardware platform, as detailed in the hardware abstractions.

The benefits of the communication path abstraction are two-fold. First, it provides a uniform abstraction to describe a dataflow from the requester to the responder, and can be extended to describe multiple round trips by composing a sequence of multiple communication paths, thereby successfully describing the diverse dataflows from the application layer. Second, it conveys estimated quantitative performance metrics derived from the hardware layer to showcase its communication performance. These estimated metrics are essential for multi-path optimization to analyze the communication interference across paths (§III-B).

Application abstractions. Leveraging the communication path abstraction, the dataflow of MINC communication can be conceptually expressed as a sequence of multiple communication paths, with each path conveying a request for one round trip. Formally, the application abstractions are formulated as follows, where we use KVS for illustration:

- *Request type*: This represents the type of the request for processing, *e.g.*, `GET` and `PUT` in KVS.
- *Request dependency*: This describes the dependency between requests to ensure correct execution, as out-of-order cases may occur along with multi-path optimization (§III-B). In KVS, a `PUT` request must be executed before a `GET` request if both access data at the same address to avoid the `GET` obtaining a stale value. Dependency types include: `NONE` (no dependency), `ADDR` (same address), and `MSG` (same message).
- *Path list*: This describes the sequence of communication paths traversed for the corresponding dataflow. In KVS, a `GET` request has two dataflow types, whereas `PUT` has only one, each involving two round trips due to the network amplification effect [41].
- *Data size at each link*: This describes the predetermined size of the data structure traversed at each link during request processing. In KVS, a `GET` request firstly traverses the 8-byte key at each link and then traverses the 64-byte value on the same path. The `PUT` request operates similarly.
- *Compute time*: This lists the compute time or memory access time at each endhost device, which can be profiled before processing the application.

Hardware abstractions. The hardware abstractions can be explicitly described by the hardware model of the accelerator and the associated performance metrics, *i.e.*, bandwidth, message per second, and latency [24]. We illustrate the hardware abstractions in Figure 5c, showcasing the direct model of an FPGA-based hardware platform (§IV) transmitting 64B mes-

Symbols	Description
$P = \{p_1, p_2, ..., p_i\}$	Set of all paths
$L = \{l_1, l_2, ..., l_n\}$	Set of all intra-host links
$C = \{c_1, c_2, ..., c_o\}$	Set of all compute nodes
$R = \{r_1, r_2, ..., r_t\}$	Set of all request types
$\mathcal{L}(p_i) = [c_{i1}, l_{i1}, ..., c_{ik}]$	Sequence of nodes and links in p_i
$\mathbb{P}(r_t) = [p_{t1}, ..., p_{tn}]$	Path list of r_t
Δt_t	Time interval between r_t and r_{t+1}
$S(r_t)$	Size of r_t
$BW(l_j)$	Bandwidth capacity of l_j
$T_{tr}(l_j)$	Base traverse time of l_j
$T_c(c_j, r_t)$	Compute time of c_j for r_t
$X(p_i)_t = \{0, 1\}$	Whether select request r_t to p_i at t
$Q(l_j)_t$	Queue latency of l_j at step t
$Q(c_i)_t$	Queue latency of c_i at step t

TABLE I: Key notations in problem formulation.

sages over PCIe Gen3x16. The hardware model is expressed in nodes and edges, where each node represents a hardware device (CPU, accelerator, NIC), and each edge represents a physical link between devices. Note that the mapping between logical links and physical links is determined by the hardware model. We provide three representative hardware model abstractions [25], [39]: direct, on-path, and off-path, and support customized model abstractions for flexibility. The performance metrics are measured during initialization with profiling tools such as VTune [2] and Hostping [24] for intra-host communication.

B. Multi-path Optimization

LEO should support efficient multi-path optimization for hybrid resource utilization. Unlike previous multi-path works which select paths between hosts [28], [32], LEO selects paths within each host, exhibiting several unique properties: i) Only intra-host communication needs to be considered since inter-host communication is handled and transparentized by the hardware transport, *e.g.*, RDMA; ii) Sufficient information is available in both the application and hardware, including communication paths for each request type, profiled compute times, and hardware models.

These properties simplify the path selection problem, making it solvable theoretically. We first formulate the path selection problem as a linear programming (LP) problem, which is solvable optimally with off-the-shelf LP solvers [29] (§III-B1). Next, we propose a heuristic algorithm to address the time constraints of the LP solver (§III-B2).

1) Problem Formulation: Table I lists the key notations used for problem formulation. We address the path selection problem at a fine-grained, per-request level. Specifically, we discretize the entire processing time into separate steps, with each step spanning Δt_t, representing the arrival time interval between request r_t and r_{t+1}. Our primary objective is to minimize the total intra-host JCT, which are the sum of their compute and communication time within hosts. As the compute time $T_c(c_j, r_t)$ is fixed and known in advance, our main focus is to minimize the communication cost, defined as the sum of transmission time and queue latency at each link and compute node.

While the transmission time is fixed, the queue latency varies over time and depends on the previous queuing status:

$$Q(c_i)_t = \begin{cases} \max\{Q(c_i)_{t-1} + T_c(c_i, r_t) - \Delta t_t, 0\} & \text{if } c_i \in \mathcal{L}(p_i) \\ & \&\&X(p_i)_t = 1 \\ \max\{Q(c_i)_{t-1} - \Delta t_t, 0\} & \text{otherwise} \end{cases} \tag{1}$$

$$Q(l_j)_t = \begin{cases} \max\{Q(l_j)_{t-1} + \frac{S(r_t)}{BW(l_j)} - \Delta t_t, 0\} & \text{if } l_j \in \mathcal{L}(p_i) \\ & \&\&X(p_i)_t = 1 \\ \max\{Q(l_j)_{t-1} - \Delta t_t, 0\} & \text{otherwise} \end{cases} \tag{2}$$

where $Q(c_i)_t$ and $Q(l_j)_t$ increases only if the request at time t selects p_i and traverses c_i or l_j, otherwise they decrease by Δt_t. As these equations contain non-linear $\max\{\cdot\}$ functions, we apply the big-M method to linearize them for resolution.

We denote the queue latency and traverse time of path p_i at time t as $Q(p_i)_t = \sum_{c_i \in \mathcal{L}(p_i)} Q(c_i)_t + \sum_{l_j \in \mathcal{L}(p_i)} Q(l_j)_t$ and $T(p_i, r_t)_t = \sum_{l_j \in \mathcal{L}(p_i)} T_{tr}(l_j) + \sum_{c_j \in \mathcal{L}(p_i)} T_c(c_j, r_t)$, respectively. Formally, the objective function and constraint of k-step path selection problem are:

$$\min \sum_{t=n}^{t=n+k} \sum_{p_i \in P} X(p_i)_t \cdot [Q(p_i)_t + T(p_i, r_t)_t] \tag{3}$$

$$\sum_{p_i \in \mathbb{P}(r_t)} X(p_i)_t = 1 \tag{4}$$

where (3) aims to minimize the total intra-host JCTs for all paths over a short horizon, *i.e.*, the next k request sequence predicted using historical request trace [44], while (4) imposes the path selection constraint, ensuring that each request is assigned to exactly one path.

Notably, when k=1, the LP algorithm simplifies to a greedy version that selects the path with the least JCT for each request type. This approach optimistically assumes the network can handle all queued requests. However, in practice, high request throughput can cause queuing across paths, leading to high queuing latency and degraded performance, as our results reveal (§V-C).

2) Heuristic Multi-path Selection: Though promising, the time cost of LP solver [29] is too high, *e.g.*, $\geq$2.4 ms in our evaluation (§V-C), to fulfil the real-time requirement of µs-level applications. To address this, we propose a per-request-type heuristic algorithm for real-time path selection. We observe a trade-off between selecting a queuing path with low latency and an idle path with high latency. Adding a request to the queuing path increases the queue latency for all subsequent requests until the queue is drained, leading to an extra cost $\hat{k} \cdot [\sum_{c_i \in \mathcal{L}(p_i)} T_c(c_i, r_t) + \sum_{l_j \in \mathcal{L}(p_i)} \frac{S(r_t)}{BW(l_j)}]$ according to (3), where $\hat{k}$ denotes the number of requests that will experience the queue latency caused by the current request. We call it the *communication toll* of the path decision. Finally, the trade-off is transformed into balancing between the immediate JCT gain and the incurred communication toll.

Algorithm overview. Our algorithm operates in two phases: we firstly offline calculate the traverse time and communication tolls for all request-path pairs, and then online select path via estimating the path status according to the intra-host JCTs of completed requests.

In the offline phase, we calculate the basic path traverse time $T(p_i, r_t)_t = \sum_{l_j \in \mathcal{L}(p_i)} T_{tr}(l_j) + \sum_{c_j \in \mathcal{L}(p_i)} T_c(c_j, r_t)$ and

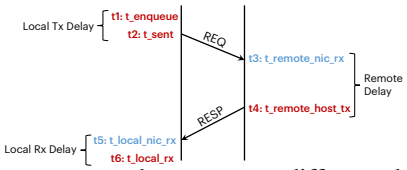


Fig. 6: Timestamps used to measure different delays at end-hosts. The hardware and software timestamps are shown in blue and red, respectively. The intra-host JCT is the sum of all delays shown in the figure.

Fig. 7: Message format of LEO.

the communication toll $Q_{toll}(p_i)_t = \sum_{c_i \in \mathcal{L}(p_i)} T_c(c_i, r_t) + \sum_{l_j \in \mathcal{L}(p_i)} \frac{S(r_t)}{BW(l_j)}$ for each path p_i and request type r_t. We use the path traverse time $T(p_i, r_t)_t$ to generate the initial path selection priority list for each request type, as the queue is initially empty in each path.

Next, in the online phase, we select the appropriate path according to both the current JCT and the communication toll. We denote $\hat{k}_{p_i}$ as the estimated number of requests that will experience queue latency caused by the current request if selecting p_i. Then, the JCTs of the subsequent $\hat{k}_{p_i}$ requests will also increase by $Q_{toll}(p_i)_t$. Consequently, the total intra-host JCT is $T_{total}(p_i)_t = \hat{k}_{p_i} \cdot Q_{toll}(p_i)_t + T(p_i, r_t)_t$. By simply comparing the $T_{total}(p_i)_t$ of all paths, we select the path with the minimal value as the final choice, thus approximating (3).

Estimation of $\hat{k}_{p_i}$. The key challenge of the algorithm is to estimate the value of $\hat{k}_{p_i}$ to accurately reflect each path's status. We achieve this by monitoring the intra-host JCTs of previous requests, following the similar method as [18], and using these JCTs as indicators of the current path status. Notably, modern NICs widely support accessible hardware timestamps at hosts [10], [18]. Therefore, LEO uses multiple hardware and software timestamps to collaboratively monitor the intra-host JCT of each path.

Figure 6 illustrates the event sequence of a request traversing the path. t_1 and t_2 are the times when the message is enqueued to the corresponding path and to the NIC queue, respectively, recorded by LEO using the CPU. t_3 is the time when the descriptor of the request, *i.e.*, CQE, is obtained at the server side by RNIC. t_4 is the time when the message is enqueued to the NIC queue, recorded by the CPU. Note that t_4-t_3 is calculated by the server CPU and appended in the message header. t_5 and t_6 are the corresponding receive timestamps for the response at the client side and are appended locally at the client. Details of the message format are shown in Figure 7. Finally, the overall intra-host JCT of the path is calculated as $(t_2 - t_1) + (t_4 - t_3) + (t_6 - t_5)$.

Based on the above monitored intra-host JCT, we can now estimate the value of $\hat{k}_{p_i}$ for each path. Specifically, we initialize $\hat{k}_{p_i}$ to 0 for each path and follow these procedures during application runtime: i) Upon receiving a completed request, we compare the monitored JCT with the base traverse time $T(p_i, r_t)_t$; ii) If the monitored value is larger, it indicates the queue exists in the path, and we increase $\hat{k}_{p_i}$ by one, otherwise we reset $\hat{k}_{p_i}$ to zero.

APIs	Description
`init()`	Init LEO comm. context.
`send(m)`	Two-sided send a message.
`recv()` → `m`	Two-sided receive a message.
`write(m)`	One-sided write a message.
`read()` → `m`	One-sided read a message.
`handle(m,r)`	Handle a message for role.
Wrappers	**Description**
`reg_comm_func(f,n,r)`	Register a comm. function.
`reg_mem_func(f,n,r)`	Register a memory function.
`reg_ts_func(f,n)`	Register a timestamp function.
`reg_app_func(f,n,r)`	Register an application function.

TABLE II: LEO APIs and wrappers.

C. LEO *Programming*

APIs and wrappers. LEO wraps its communication and memory configuration together with communication optimization in unified APIs and wrappers. As listed in Table II, LEO APIs and wrappers include:

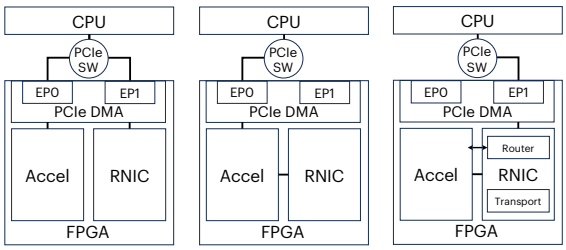
- `init` initializes the MINC communication. It automatically sets up direct communication across devices based on the registered handlers and functions.
- `send`, `recv`, `write`, and `read` are simple RDMA ib_verbs-style communication APIs for data transmission.
- `handle` is a unified API for handling incoming requests. Upon receiving a request, it automatically invokes the registered application functions at the specified device role. Note that the role can be either "cpu" or "acc".
- `reg_xxx_func` is a series of function pointer wrappers for registering hardware-specific functions, including communication, memory, timestamp service, and application.

Programming procedures. Using LEO APIs and wrappers, developers can easily program MINC applications by following these steps:

1) Implement application functions for processing incoming requests on both CPU and accelerator.
2) Configure the communication path in `YAML` format based on the communication path abstraction in §III-A.
3) Configure the dataflows for each request type in `YAML` format based on the application abstractions in §III-A.
4) Write initialization code with LEO APIs to register functions, pin memory, configure memory mapping, and initialize dataflows. We omit the example configuration code here due to space limitation.
5) Implement the application processing code with LEO APIs.

IV. IMPLEMENTATION

We have implemented a fully functional LEO prototype for developing MINC communication. It currently supports representative FPGA-based smartNICs [6], SoC-based BlueField-3 [3] and GPU. Below we illustrate LEO's implementation, our efforts undertaken for an FPGA-based emulation platform and the automatic configuration scheme.



(a) Direct (b) On-path (c) Off-path

Fig. 8: Hardware platform architecture.

Modules	LUT	Registers	BRAM	URAM
PCIe P2P	3.3%	1.66%	3.01%	5%
Router	0.12%	0.09%	3.95%	0%
EBS lookup functions	4.8%	3.8%	12.9%	8.5%

TABLE III: Resource usage of hardware platform in Xilinx VU35P FPGA board.

LEO implementation. We implement LEO in C++ and integrate it with MLNX_OFED-5.4-3.0.3.0, CUDA 12.4, NCCL 2.20.5, and DOCA 2.2.2. We create 5 queue pairs (QPs) in userspace for ease of deployment and to avoid kernel syscall overhead. For the timestamp service, we set flags of CQs for hardware timestamping [9], and record the software timestamps when enqueuing and dequeuing descriptors. To calculate the time interval, we enable PTP in NIC and convert HCA clock to ns using `mlx5dv_ts_to_ns` before calculation. Additionally, we implement the LP algorithm using `CPLEX` [29] and the heuristic algorithm purely in C++, both running on separate threads for online path selection.

FPGA-based emulation platform. To evaluate the genericity of LEO across hardware models, we implement an FPGA-based platform with a Xilinx VU35P FPGA [6], featuring a PCIe Gen3x16 interface and a 100Gbps Ethernet port. This platform integrates accelerator and RDMA NIC [40] logic. We emulate different hardware models by modifying FPGA transmission logic (Figure 8): in the direct model, the accelerator and NIC are isolated and connected via a shared PCIe switch; in the on-path model, they are connected via AXI, with only end-point 0 (EP0) linked to the CPU; and in the off-path model, only EP1 is active, with a router in the RNIC enabling packet routing.

We list the detailed hardware resource consumption in Table III. Note that the EBS lookup functions are developed for emulating three lookup functions in EBS which consumes on-chip resource, while for KVS, its index lookup consumes no on-chip resource as we store the large KV table (20M KV-pairs) in FPGA HBM (§V-A).

Automatic communication configuration. LEO simplifies device communication by automating configuration using developer-provided wrappers, reducing complexity. For example, in an FPGA Direct RNIC setup, the FPGA memory is mapped to the PCIe BAR space, which LEO registers as memory regions (MRs) in RNIC along with address translation data. LEO then initializes inter-host communication using standard RDMA procedures and exchanges MR information to enable direct data transfer. By adhering strictly to PCIe P2P and RDMA protocols [5], [8], this approach ensures generality across devices..

Application	Testbed	HW Model	Baseline
KVS	F-NIC	Off-path	DrTM-KV [42]
DFS	BF-3	Off-path	DOCA [7]
EBS	F-NIC	On-path	Solar [30], iPipe [25]
DML	G+F-NIC	Direct	NCCL [4]

TABLE IV: Representative MINC applications with their employed testbed configurations, hardware models, and baselines.

V. EVALUATION

In this section, we evaluate LEO's performance to answer the following questions:

- **How generic and efficient is LEO against the state-of-the-art frameworks when processing each application?** We show that LEO supports all types of MINC applications across different testbeds listed in Table IV, and can achieve 1.2–4.7× better performance than state-of-the-art frameworks (§V-B).
- **How effective is LEO in terms of its component and overhead?** We show that LEO's components effectively improves the performance of MINC communication (§V-C). Specifically, the multi-path optimization reduces the median latency of MINC communication by 1.3×. Besides, we also demonstrate that LEO introduces negligible CPU overhead and low processing time (§V-C).

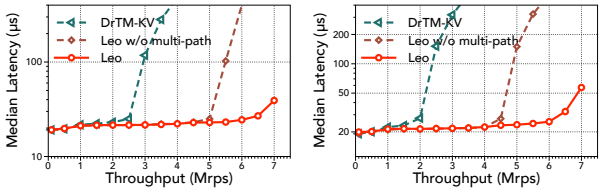
A. Experimental Setup

Testbed configurations. We run our experiments using three testbed configurations: F-NIC, G+F-NIC, and BF-3. All testbeds use the same topology, *i.e.*, two servers directly connected with each other, but with different devices:

- F-NIC adopts Intel servers with 8-core Xeon Silver 4110 CPU, 192GB memory, PCIe 3.0, 100GE cables, and Xilinx VU35P FPGAs.
- G+F-NIC employs the same servers as F-NIC with the addition of NVIDIA A10 GPUs connected.
- BF-3 uses Intel servers with 32-core Xeon Silver 4309Y CPU, 512GB memory, PCIe 4.0, 200GE cables, and NVIDIA BlueField-3.

Case studies & baselines. To extensively evaluate LEO, we implement four representative MINC applications running on three types of hardware models and compare them with state-of-the-art baselines, as listed in Table IV.

- Key-value store: We implement KVS by executing the index lookup in FPGA and facilitating one READ to retrieve the index from FPGA and another WRITE/READ for the value in host memory. We compare LEO with DrTM-KV [42] over the same off-path F-NIC testbed.
- Distributed file system: We use DOCA [7] as the baseline which compresses/decompresses 1MB file with 256KB IO chunks and a 51.8% compression ratio, and compare with LEO using the same off-path BF-3 testbed.



(a) YCSB-C workload. (b) YCSB-B workload.

Fig. 9: Performance of key-value store.

- Elastic block storage: As Solar [30] is not open-sourced, we implement its offloaded QoS, block, and address lookup functions in FPGA to emulate read requests[4]. iPipe [25] is a customized on-path smartNIC framework that supports dynamic function placement between the host CPU and smartNIC. We emulate it by first profiling the *base JCT* of offloading all functions in FPGA and then online invoking functions in CPU to emulate actor migration[5] when *current JCT* is larger than *base JCT* and vice versa. We conduct these experiments using the same on-path F-NIC testbed.
- Distributed machine learning: We use only the RNIC module in FPGA and configure GPUDirect-FNIC via PCIe P2P [5]. We compare LEO with NCCL [4] over the same direct G+F-NIC testbed.

In all experiments, we add delays at the requester NIC to simulate network variation between hosts [46], with the value ranging from 3 to 9µs according to [45].

Metrics. We report the median latency under different request rates for latency-intensive key-value stores and throughput for other throughput-intensive applications.

B. End-to-end Application Performance

Key-value store. We use YCSB-C and B [12] as our workloads, as used in prior work [42]. The YCSB-C workload only contains the `GET` request. As shown in Figure 9a, we observe that the LEO counterpart without multi-path support starts to increase its median latency when system workload is above 5Mrps, outperforming the CPU-based DrTM-KV by ∼1.5×. The reason for the higher sustainable throughput is attributed to the efficient computation at accelerators. LEO outperforms both its variant and DrTM-KV by delivering stably low latency and higher sustainable throughput. Specifically, LEO delivers 1.2–2× higher sustainable throughput over its counterparts, as its system demonstrates higher computation capacity by introducing both CPU and accelerator for handling the incoming request streams. Meanwhile, LEO exhibits stably low median latency with the increasing of the offered load. The reason comes from its careful path-selection algorithm, which helps LEO to balance the load among multiple computation sources and mitigate the potential temporal overload.

For the YCSB-B workload, it contains both `GET` and `PUT` requests. For the `PUT` request, it can only be processed by the CPU, since there are currently no sophisticated mechanisms for maintaining data consistency at the accelerator. The scheduling in this workload is more complex as the placement constraints and the uniform workload regarding execution times. As shown in Figure 9b, both LEO and its variant deliver as much as 3× higher throughput than DrTM-KV, while LEO achieves the highest throughput. The reason aligns with the case in Figure 9a. This experiment further demonstrates the benefits of LEO's incorporation of CPU and accelerator and effective scheduling with balancing load among multiple paths.

[4]The SEC and CRC parts in Solar are handled by RDMA transport.

[5]We omit the procedure of actor migration for ease of emulation, but it can actually degrade performance.

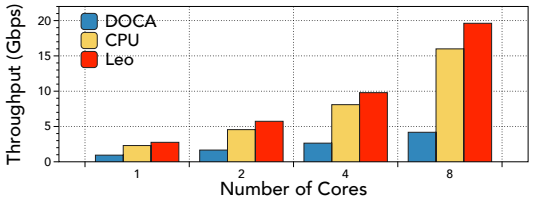
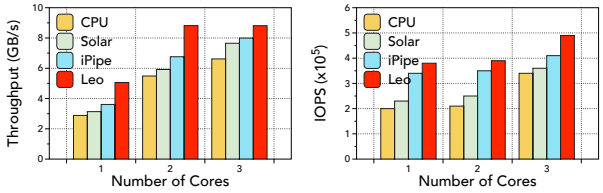


Fig. 10: Performance of distributed file system.



(a) Throughput of 64KB read. (b) IOPS of 4KB read.

Fig. 11: Performance of elastic block storage.

Distributed file system. We conduct the experiments by compressing the 1MB file at the requester and write to the responder for decompression. Figure 10 shows the throughput with an increasing number of cores on both sides. We find that DOCA performs the worst because its compression time is longer than that of the CPU, *i.e.*, 2.18ms vs 0.89ms, due to the fact that the compression firmware on BF-3 is less powerful than a single core at the server. When increasing the number of cores, both approaches show improved performance due to better decompression pipeline at the responder. Overall, LEO outperforms DOCA and CPU by 1.19–4.69× as the number of clients increases. The enhancement is attributed to LEO's effective utilization of hybrid resources.

Elastic block storage. We use the same experiment setting of [30] and compare the 64KB throughput and 4KB IOPS (I/O per second) of LEO with CPU, Solar, and iPipe. The results shown in Figure 11 reveal that Solar achieves better performance than CPU-based solution, as it reduces the processing latency via dataplane offloading. iPipe is better for migrating some functions to CPU according to the online end-to-end JCTs. It achieves better throughput and IOPS by 1.04–1.25× and 1.19–1.71×, respectively, than CPU and Solar with different number of cores. However, it performs worse than LEO as its migration signal, end-to-end JCT, can be affected by network variation. The fluctuation may result in a false migration from a good offloading placement to a worse one. Besides, we remind readers that we did not emulate the actor migration process, which could result in

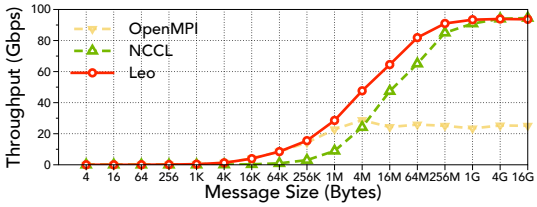
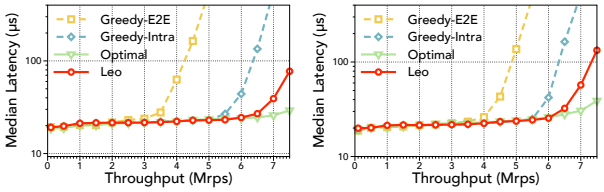


Fig. 12: Performance of DML allreduce.



(a) YCSB-C workload. (b) YCSB-B workload.
Fig. 13: Effectiveness of LEO multi-path optimization.

even worse performance with frequent migrations. Overall, LEO demonstrates the highest performance among all, thanks to its better resource utilization with multi-path and intra-host JCT signal for load balancing. Specifically, with LEO, the throughput of 64KB read increases by 1.14–1.75× given the varying number of CPU cores, while the IOPS of 4KB read also exhibits a rise of 1.1–1.9×.

Distributed machine learning. Allreduce is a key communication pattern in distributed machine learning, with varying message sizes representing different sizes of gradients and parameters. We compare the allreduce throughput of LEO, NCCL, and OpenMPI with different message sizes. As shown in Figure 12, LEO consistently delivers the highest throughput in all message sizes. For large messages (≥1MB), LEO shows a substantial throughput improvement over OpenMPI. Compared to NCCL, LEO's throughput is on average 4.08× higher for messages ranging from 4KB to 1GB. Other message sizes perform similarly. OpenMPI performs worst among all due to the low processing capability of the CPU. The benefit of LEO is attributed to its use of idle compute resources through its multi-path design, allowing it to saturate the link faster than NCCL.

C. LEO *Deep Dive*

Effectiveness of path selection algorithm. We compare the heuristic algorithm adopted in LEO with two greedy algorithms, *i.e.*, selecting path with minimal end-to-end JCTs (Greedy-E2E) and minimal intra-host JCTs (Greedy-Intra), as well as the optimal LP algorithm in the key-value store application. We do not predict the request sequence but directly pre-record the history of the request sequence and solve the LP problem offline for each step to obtain the performance of the optimal solution. The results are shown in Figure 13. We observe that Greedy-E2E performs the worst as it can misselect paths due to network fluctuation between hosts. Greedy-Intra performs better since it only uses the intra-host JCT for selection. However, this algorithm operates in a greedy manner but overlook the temporal queuing delay effect, *i.e.*, the communication toll, caused by the current path decision (similar to k=1 algorithm illustrated in §III-B1), and therefore performs poorly over a horizon of requests. LEO demonstrates 1.18–2.17× performance gain over the two greedy algorithms, stemming from both the intra-host JCT to reflect path status and the consideration of communication toll. Besides, we also observe that LEO achieves comparable performance to the optimal solution, which knows the whole request sequence in advance. This observation reveals the effectiveness of the heuristic algorithm in LEO.

LEO framework overhead. We seek to understand the framework overhead of LEO. We measure the CPU utilization of the heuristic algorithm, the processing time of algorithms and analyze the header overhead. We find that LEO can run the heuristic algorithm for µs-level applications, with an average cost of 10.3% CPU utilization. We also report the path selection time for one request. Specifically, the LP solver takes as much as 2.4ms, and the heuristic algorithm consumes only ∼1.5µs, demonstrating its applicability in real-time path selection. For the message header, it takes up as much as 4 bytes for each response message. The overhead depends on the response size, ranging from 1.6×10^{-5} for DFS with 256KB chunks to 5.9% for KVS with 64B values in the case studies.

VI. RELATED WORK

Network-software co-design of INC. Systems with network-software co-design for INC have been illustrated in §II-A.

PINC frameworks. There have been several frameworks [14], [43], [47] proposed recently to facilitate PINC application development. These works are generic to serve PINC hardware rather than MINC hardware.

MINC frameworks. iPipe [25] automates fine-grained actor migration between CPU and on-path smartNICs. Floem [31] is a smartNIC compiler that eases the developers' programming effort. However, these frameworks are customized for smartNICs only, but do not support other PCIe devices such as GPUs and FPGAs. Moreover, they either do not support hybrid resource utilization or perform poor load balancing when inter-host network fluctuates (§V-B).

VII. CONCLUSION

This paper presents LEO, a generic and efficient communication framework for MINC applications. Experiment results with four case studies over three testbeds have demonstrated the genericity and efficiency of LEO. We hope LEO could serve as a stepping stone for the development of MINC applications and inspire research for MINC.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITF ACCESS, NSFC 62062005, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Kai Chen is the corresponding author.

REFERENCES

[1] Intel tofino p4-programmable ethernet switch asic that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2023.

[2] Intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, 2023.

[3] Nvidia bluefield networking platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2023.

[4] Nvidia collective communication library. https://developer.nvidia.com/nccl, 2023.

[5] Pci-sig specifications. https://pcisig.com/specifications, 2023.

[6] Virtex ultrascale+ fpga. https://www.xilinx.com/products/boards-and-kits/device-family/nav-virtex-ultrascale-plus.html, 2023.

[7] Doca compress. https://docs.nvidia.com/doca/sdk/doca+compress/index.html, 2024.

[8] Infiniband specification. https://www.afs.enea.it/asantoro/V1r1_2_1.Release_12062007.pdf, 2024.

[9] Time-stamp service. https://docs.nvidia.com/networking/display/mlnxofedv571020/time-stamping#src-2396584992_TimeStamping-time-stampingservice, 2024.

[10] Time-stamping service. https://docs.nvidia.com/networking/display/mlnxofedv473290/time-stamping, 2024.

[11] Vhdl and fpga terminology. https://vhdlwhiz.com/terminology/simulation/, 2024.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, et al. Azure accelerated networking:smartnics in the public cloud. In *NSDI*, pages 51–66, 2018.

[14] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *SIGCOMM*, pages 435–450, 2020.

[15] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *OSDI*, pages 463–479, 2020.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *NSDI*, pages 35–49, 2018.

[17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP*, pages 121–136, 2017.

[18] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, , et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, pages 514–528, 2020.

[19] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. Atp: In-network aggregation for multi-tenant learning. In *NSDI*, volume 21, pages 741–761, 2021.

[20] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *SOSP*, pages 137–152, 2017.

[21] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *SIGCOMM*, pages 1–14, 2016.

[22] Wenxue Li, Junyi Zhang, Yufei Liu, Gaoxiong Zeng, Zilong Wang, Chaoliang Zeng, Pengpeng Zhou, Qiaoling Wang, and Kai Chen. Cepheus: accelerating datacenter applications with high-performance roce-capable multicast. In *HPCA*, 2024.

[23] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, cheap and in control with switchkv. In *NSDI*, pages 31–44, 2016.

[24] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in rdma servers. In *NSDI*, pages 15–29, 2023.

[25] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *SIGCOMM*, pages 318–333. 2019.

[26] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. In-network aggregation with transport transparency for distributed training. In *ASPLOS*, pages 376–391, 2023.

[27] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *FAST*, volume 19, pages 143–157, 2019.

[28] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for rdma in datacenters. In *NSDI*, pages 357–371, 2018.

[29] CPLEX User's Manual. Ibm ilog cplex optimization studio. *Version*, 12(1987-2018):1, 1987.

[30] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *SIGCOMM*, pages 753–766, 2022.

[31] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas E Anderson. Floem: A programming system for nic-accelerated network applications. In *OSDI*, volume 18, pages 663–679, 2018.

[32] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *SIGCOMM*, 2011.

[33] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *NSDI*, 2021.

[34] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *SOSP*, pages 740–755, 2021.

[35] Xinchen Wan, Kai Chen, and Yiming Zhang. Dgs: Communication-efficient graph sampling for distributed gnn training. In *ICNP*, 2022.

[36] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. Scalable and efficient full-graph gnn training for large graphs. In *SIGMOD*, 2023.

[37] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. Rat-resilient allreduce tree for distributed machine learning. In *APNet*, pages 52–57, 2020.

[38] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards domain-specific network transport for distributed dnn training. In *NSDI*, 2024.

[39] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. Fpganic: An fpga-based versatile 100gb smartnic for gpus. In *ATC*, pages 967–986, 2022.

[40] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. Srnic: A scalable architecture for rdma nics. In *NSDI*, pages 1–14, 2023.

[41] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path smartnic for accelerating distributed systems. In *OSDI*, 2023.

[42] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *OSDI*, pages 233–251, 2018.

[43] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. Clickinc: In-network computing as a service in heterogeneous programmable data-center networks. In *SIGCOMM*, pages 798–815, 2023.

[44] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *SIGCOMM*, pages 325–338, 2015.

[45] Junxue Zhang, Wei Bai, and Kai Chen. Enabling ecn for datacenter networks with rtt variations. In *CoNEXT*, pages 233–245, 2019.

[46] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. Liteflow: towards high-performance adaptive neural networks for kernel datapath. In *SIGCOMM*, pages 414–427, 2022.

[47] Bohan Zhao, Wenfei Wu, and Wei Xu. Netrpc: Enabling in-network computation in remote procedure calls. In *NSDI*, pages 199–217, 2023.