Achieving Fast and High Throughput Data Exchange for Serverless Computing Systems via Switch-Native Serialization/Deserialization

Gonglong Chen[1], Baiyan Ke[3], Yuxin Xu[1,2], Shenghong Xiong[1,2], Haolin Pan[3],
Jiamei Lv[4], Wenxing Ge[5], Yuxi Wang[5], Chengzhong Xu[6], and Kejiang Ye[1*]
[1]Shenzhen Institutes of Advanced Technology, CAS. [2]University of CAS. [3]Foshan University.
[4]Zhejiang University. [5]Independent Researcher. [6]University of Macau.
Email: {*gl.chen2, yx.xu2, sh.xiong, kj.ye*}*@siat.ac.cn*,
{*baiyanke29, panpanhaolin, vincentge95, leahwang2009*}*@gmail.com*, *lvjm@zju.edu.cn*, *czxu@um.edu.mo*

***Abstract*—Serverless computing has emerged as a dominant cloud paradigm that enhances development efficiency and reduces operational costs. Serialization and deserialization are key operations in these systems. However, these processes may contribute up to 98% of the total latency, making the host *compute-bound*. Recent serialization-free approaches, such as RMMap, eliminate serialization overhead by directly mapping memory between functions in distributed serverless systems. Nevertheless, transferring raw memory produces large payloads and imposes significant network demands, rendering the process *bandwidth-bound*.**

To address these challenges, we propose HyperLink, a *network-assisted* system that unifies the benefits of both *compute-bound* and *bandwidth-bound* approaches. HyperLink offloads intensive serialization tasks (e.g., data compression) to programmable switches, while the host employs a lightweight memory mapping mechanism. It features a parallel striped storage module for efficient compression dictionary management, a latency-aware adaptive encoder that balances compression delays with compression ratios, and a dependency-gated parallel decoder that decouples packet-level decoding into block-level processing to enhance throughput. Experimental results show that HyperLink reduces latency by up to 68.12% and achieves 8.25× throughput compared to state-of-the-art solutions.

I. Introduction

Serverless computing is a dominant cloud paradigm that improves development efficiency and reduces operational costs [1–3]. The fundamental unit of the serverless computing system is the *function*. These small, user-defined code units execute on demand. Functions are orchestrated into *workflows* to support complex applications [3–8]. This abstraction allows developers to focus exclusively on application logic [1–4, 7, 9]. Recent studies indicate that serverless workflows can significantly boost development efficiency by 35% and decrease operational expenditures by nearly 50% [10].

Serialization and deserialization are critical components in serverless workflows, where functions typically execute in isolated environments or on physically distinct machines [4, 9, 11]. Since direct memory sharing between these functions is often restricted, in-memory objects generally require conversion into transferable byte streams (serialization) and subsequent reconstruction (deserialization). Prior research [4] indicates that this process can contribute significantly to execution latency, potentially accounting for 40% to 98% of the total delay in complex workflows. This overhead stems primarily from the substantial computation and memory copying involved.

To mitigate this bottleneck, several optimization strategies have been proposed [3, 12–17]. However, early approaches primarily targeted messaging [12, 13] or storage overhead [16, 17]. These methods largely overlooked the intrinsic costs of serialization and deserialization. In particular, the computational burden of integrated compression algorithms remains a critical issue. Serialization systems like Protobuf [18] often employ methods such as Gzip [19] or Snappy [20] to minimize data size. Yet, the overhead of these compression operations is substantial. Studies indicate that this overhead can dominate the end-to-end execution time [21].

Recently, a serialization-free approach, RMMap [4], has been proposed. It introduces an operating system primitive that maps memory directly between functions in distributed environments, eliminating the overhead of serialization in serverless workflows. However, transferring raw memory produces large payloads that impose significant bandwidth demands. In contrast, serialization methods such as Protobuf [18] support compression and achieve higher concurrent throughput. As detailed in Section II, the absence of data compression in RMMap [4] increases network contention, leading to sender throttling and higher end-to-end latency, which ultimately diminishes the performance advantages of bypassing serialization.

To overcome these performance issues, we propose an innovative approach, HyperLink, which employs a *network-assisted* mechanism for data exchange among containers. Unlike conventional *compute-bound* schemes [22, 23] that require heavy host-side serialization to reduce network bandwidth, or *bandwidth-bound* schemes [24] that rely on direct memory mapping and suffer from high transmission costs, HyperLink unifies the strengths of both. Specifically, HyperLink offloads resource-intensive serialization tasks (e.g., data compression [25]) to programmable network switches that reside along the communication path between container pairs, while the host maintains only a lightweight memory mapping mechanism. This hybrid design keeps host computational overhead low

*Corresponding author.

979-8-3195-1703-6/26/$31.00 © 2026 IEEE

and reduces network bandwidth usage through in-network processing. As a result, HyperLink alleviates congestion and minimizes processing latency in serverless workflows.

However, achieving the above *network-assisted* data exchange faces new challenges, as enabling efficient in-network data compression requires overcoming inherent hardware limitations and carefully balancing overall system performance.

Challenge 1: high-throughput dictionary buffer management. Efficient data compression in data exchange typically relies on maintaining a dynamic dictionary buffer to capture and store recent traffic history. This buffer serves as the foundation for identifying repeated patterns. However, programmable switches are subject to strict limitations on memory access frequency and parallelism. Standard register arrays typically permit only a single access per packet processing cycle, creating a bottleneck when updating or querying the buffer for high-speed data streams. To overcome this limitation, we propose a *parallel striped storage module* (Section III-B). By reorganizing conventional one-dimensional register arrays into a two-dimensional matrix through striping, our design enables simultaneous updates and lookups. This transformation substantially increases effective memory bandwidth, allowing dictionary operations to be executed at line rate.

Challenge 2: balancing compression ratio and processing latency under resource constraints. High compression ratios require identifying long matching patterns within the dictionary buffer. This task involves retrieving and comparing multiple candidate entries. However, in programmable switches with limited parallel reads, simultaneous candidate lookups for these candidates lead to resource contention. Conventional conflict resolution methods, such as additional recirculation rounds, cause unacceptable delays. Conversely, simply discarding conflicted candidates reduces the likelihood of finding long matches, degrading the compression ratio. To address this trade-off, we propose a *latency-aware adaptive encoder* (Section III-C). The encoder formulates candidate scheduling as a joint optimization problem. It employs a lightweight scheduling algorithm that decides which candidate entries to fetch immediately and which to process via recirculation or discard. This design balances recirculation delay and compression ratio, yields transmission savings, and maximizes throughput without compromising latency.

Challenge 3: overcoming sequential dependencies in dictionary-based packet decoding. Low latency packet decoding on programmable switching ASICs poses significant challenges. Traditional designs work on a per packet basis. Packet P_n does not start decoding until packet P_{n-1} is fully processed. The previous packet provides the dictionary data for restoration. Network jitter and out-of-order arrivals further complicate dependency management. To overcome these issues, we propose a *dependency-gated parallel decoder* (Section III-D). It shifts the processing granularity from packets to blocks. A block can be decoded as soon as the required data is available. A packet does not need to be completely decoded before later blocks can be handled. If the necessary data is not ready, the block is deferred to a subsequent cycle when resolution is more likely. This fine grained interleaved processing improves concurrency and reduces pipeline stalls.

We prototyped HyperLink on a testbed with sixteen servers and two programmable switches. This setup demonstrates the system's ability to achieve low latency and high throughput simultaneously (Section V). We evaluated HyperLink using production traces from typical serverless machine learning prediction workflows [3, 6]. Additionally, we used microbenchmarks involving various Python data types [4]. The results show that HyperLink reduces latency by up to 55.79% and 68.12% compared to RMMap [4] and Protobuf [18]. Furthermore, HyperLink improves throughput by 4.12× and 8.25× over these state-of-the-art solutions.

This paper makes the following contributions:

- We analyze existing state transfer approaches for serverless workflows and identify critical limitations in current designs. In particular, *compute-bound* methods [22, 23] that rely on heavy serialization/deserialization incur excessive computational overhead, while *bandwidth-bound* approaches [26, 27] that directly transmit raw data, such as RMMap [4], lead to severe network congestion and reduced aggregation throughput. These design flaws hinder system scalability and performance (Section II).
- We propose HyperLink, an innovative system that accelerates data exchange in serverless workflows through a *network assisted* approach. HyperLink offloads resource intensive serialization tasks (e.g., data compression) to programmable switches while employing a lightweight memory mapping mechanism on the host. This design harnesses the benefits of *compute-bound* and *bandwidth-bound* methods to mitigate congestion and enhance throughput (Section III).
- We implement a HyperLink prototype on a testbed consisting of sixteen servers and two programmable switches and evaluate it using production traces. The experimental results demonstrate that HyperLink significantly enhances scalability and reduces end-to-end latency compared to state-of-the-art solutions (Sections IV and V).

II. Background and Motivations

This section provides a background on the benefits of serverless computing, and the role of serialization in serverless computing. We summarize the issues and problems of existing serialization approaches and its optimizations. We also review dictionary-based compression algorithms such as LZ4 [28] to enhance transmission efficiency.

A. The Benefits of Serverless Computing

Serverless computing abstracts the underlying infrastructure so that developers can focus solely on application logic. Platforms such as AWS Lambda [29], Microsoft Azure Functions [30], Alibaba Serverless Application Engine [31], and Huawei Cloud Functions [32] automate tasks like deployment, resource allocation, auto-scaling, and load balancing, eliminating the need to manage dedicated server clusters while

enabling efficient parallel function execution. Furthermore, serverless computing supports complex workflows for diverse applications such as financial trade validation [33], machine learning pipelines [3, 6], and large-scale data processing [34]. These benefits in scalability, cost efficiency through a pay-per-use model, and agile development cycles underscore its growing importance in modern cloud computing.

B. Serialization is the Key Component of Serverless Computing

Resource isolation via containers is a core feature of serverless computing. Because functions run in separate containers without shared memory, inter-function communication must rely on message passing or distributed storage [4]. In this setting, serialization and deserialization are indispensable operations that convert runtime objects into byte arrays for network transmission or storage and then reconstruct them at the receiving end. Without these processes, state transfer between containerized functions would be infeasible and the execution of workflows would cease.

C. Limitations of Traditional Serialization Approaches

Traditional serialization modules offer several benefits. They allow for data compression and facilitate the movement of data from memory to network packets to improve transmission efficiency [4, 35].

Problems. However, these techniques incur substantial computational overhead because they require traversing complex object graphs and performing extensive memory copying. For example, serializing a 3.2MB Python data frame may involve processing over 400,000 sub-objects and take on the order of 10ms [4]. This latency is particularly detrimental for ephemeral serverless functions and prevents the use of zero-copy mechanisms such as RDMA for efficient data sharing in distributed environments. Analysis of real-world workloads [4] shows that serialization can consume 42%-98% of the state transfer time in messaging-based workflows, making it a critical bottleneck in serverless performance.

D. The Issues of Recent Works

Recent approaches [4] have succeeded in reducing the computation cost associated with serialization. One notable example is RMMap [4], which reduces these overheads by directly exposing a producer's raw memory data to the consumer. Instead of transforming the state into a byte array, the producer registers its memory so that the consumer can map it directly into its own address space via RDMA.

Issues. However, transmitting raw memory data increases network load and congestion risk. We evaluated RMMap [4] against Protobuf [18] using an ML prediction workflow from ORION [3] that transmits 30MB per request (derived from a 10K-image MNIST dataset [36], corresponding to 240Mbps). Fig. 1 shows that at the request rates below 400RPS, RMMap achieves lower latency due to the absence of serialization overhead. However, above 400RPS, raw data transmission saturates the link and triggers host-side congestion control

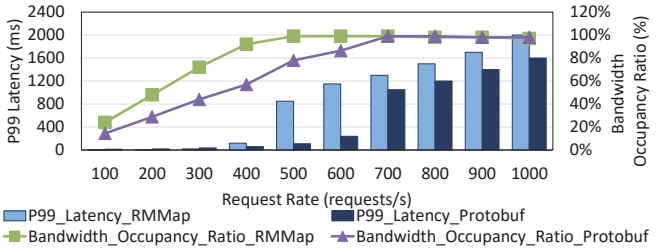

Fig. 1: The impact of RPS on the latency and bandwidth occupancy ratio.

Comp. Algo	FINRA		WordCount		ML-Train		ML-Predict		Complex.
	Comp. Ratio (%)	Comp. Latency (ms)	Comp. Ratio (%)	Comp. Latency (ms)	Comp. Ratio (%)	Comp. Latency (ms)	Comp. Ratio (%)	Comp. Latency (ms)	
Zlib	42.75	66	36.85	835	43.08	1109	42.92	795	Mid
Gzip	42.5	583	36.75	1008	42.61	2810	42.45	2018	Mid
Lzma	33.4	295	2.1	3574	37.53	13650	19.75	6661	High
Snappy	60.41	6	94.29	51	99.7	66	99.75	49	Low
Lz4	50.05	20	45.58	195	58.44	310	58.03	222	Low

TABLE I: Comparison of dictionary-based compression algorithms.

(e.g., DCTCP [37]). As a result, RMMap's latency deteriorates beyond that of Protobuf [18], ultimately increasing the overall transmission latency by a factor of approximately eight.

E. The Basis of Data Compression Algorithm

The categories of data compression algorithms. Data compression methods are either entropy encoding or dictionary-based [38]. Entropy encoding (e.g., Huffman coding [39], Gzip [19]) approximates Shannon's entropy [40] but relies on computations not supported by P4 switches [41]. Dictionary-based methods replace repeated substrings with short codes and are more hardware-friendly. We evaluated five dictionary algorithms on production datasets [4]: Zlib [42], Gzip [19], Lzma [43], Snappy [20], and LZ4 [28] (as shown in Table I). Although Zlib [42], Gzip [19], Lzma [43] compress well, their dependence on Huffman coding and prediction models hinders deployment on resource-limited hardware. Snappy's [20] simple byte-copying performs poorly on ML workloads, while LZ4 [28] offers a balanced tradeoff between compression efficiency and hardware feasibility, making it the preferred choice for real-time compression.

The basis of the LZ4 algorithm. Fig. 2 shows the basic procedure of LZ4 [28], a byte-level variant of LZ77 [44]. LZ4 [28] uses a dictionary buffer and a hash chain to locate previously seen strings. A deeper hash chain increases the chance of finding longer matches, thus improving compression. During encoding, the engine computes the hash of the first four bytes to retrieve candidate positions (Step ①). If a match is found, it selects the longest match (Step ②) and outputs an offset-length tuple; if not, it outputs the literal bytes (Step ③). During decoding, the algorithm bypasses the hash table and uses memory copying to reconstruct the data (Step ④). This design reduces complexity and boosts throughput.

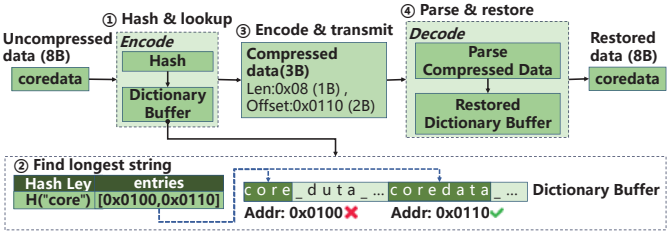

Fig. 2: The basic procedure of LZ4.

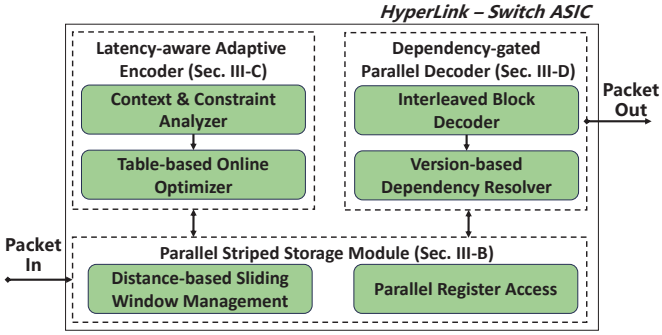

Fig. 3: The overview of HyperLink.

III. Design

A. Key Idea and Overview

As detailed in Section II, data exchange in serverless computing faces a fundamental trade-off. Conventional host-side serialization is *compute-bound*. It uses compression to save bandwidth but causes high CPU overhead. In contrast, direct memory mapping [4] is *bandwidth-bound*. It minimizes host processing while increasing transmission costs by sending raw data. To resolve this, we propose HyperLink, a *network-assisted* system. HyperLink offloads intensive serialization tasks (e.g., data compression) to programmable switches. The host retains only a lightweight memory mapping mechanism. This design simultaneously lowers host overhead and reduces bandwidth usage through in-network processing. Consequently, HyperLink minimizes latency and congestion in serverless workflows, unifying the strengths of both traditional approaches.

The system HyperLink comprises host modules and programmable switches for packet forwarding. On the host, we adopt the RMMap [4] strategy to directly map memory data. The primary focus of this paper is the programmable switch design. We demonstrate efficient in-network data compression to lower transmission overhead. Fig. 3 shows the core architecture of HyperLink, whose data plane consists of three key components.

1) Parallel striped storage module. It offers efficient dictionary management for LZ4 [28] on programmable switches by integrating two key submodules. The distance-based sliding window determines data freshness by comparing the current write pointer with candidate match positions to ensure they reside within the sliding window. Meanwhile, the parallel register access restructures the traditional one-dimensional register into a two-dimensional matrix, enabling concurrent

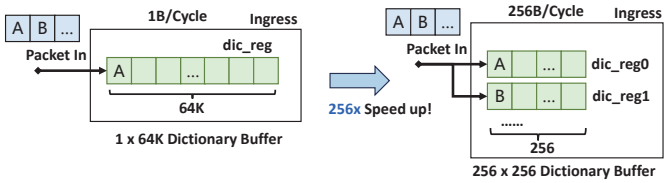

Fig. 4: The comparison of one-dimensional dictionary buffer and two-dimensional register dictionary buffer (N=256).

read and write operations. Together, they improve memory bandwidth and reduce latency while preserving forwarding performance.

2) Latency-aware adaptive encoder. It minimizes overall latency by dynamically balancing recirculation delays against transmission savings. The context and constraint analyzer collects and processes real-time system metrics, including candidate distances, register conflict information, and network congestion levels, to accurately quantify the compression potential and hardware constraints. In parallel, the table-based online optimizer leverages an offline pre-computed decision table to rapidly determine the optimal candidate retrieval strategy within a single pipeline stage. This coordinated design effectively meets hardware limitations while sustaining high compression efficiency and line-rate performance.

3) Dependency-gated parallel decoder. It boosts throughput by decoupling packet boundaries. Its interleaved block decoder concurrently processes blocks from multiple packets, filling idle pipeline stages. The version-based dependency resolver checks dictionary versions to either process valid blocks immediately or recirculate those with unmet data dependencies.

B. Parallel Striped Storage Module

According to Section II, the dictionary buffer is a core element of LZ4 [28] that serves as a sliding window caching historical data for pattern matching. A larger buffer increases the likelihood of finding repeated sequences and thus improves compression efficiency. We adopt a standard 64KB buffer, as specified by typical LZ4 settings [28], which fits within programmable switches such as Intel Tofino [41] that provide 60MB of SRAM. The buffer occupies merely 0.1% of the available space. Cross-packet data access is enabled through stateful registers in switch SRAM. However, a direct implementation using a single register array is constrained by hardware limitations that allow only one access per register per packet processing cycle [41]. Consequently, writing N bytes from a packet requires N cycles, reducing the effective bandwidth to $1/N$ of the line rate and increasing latency as subsequent packets must wait until the dictionary buffer is fully updated.

To address this bottleneck, we propose a *striped*-based buffer that reconfigures the standard one-dimensional register array (1×64K elements, one byte each) into a two-dimensional register array (256×256 elements, one byte each). This design enables the simultaneous loading of all N bytes

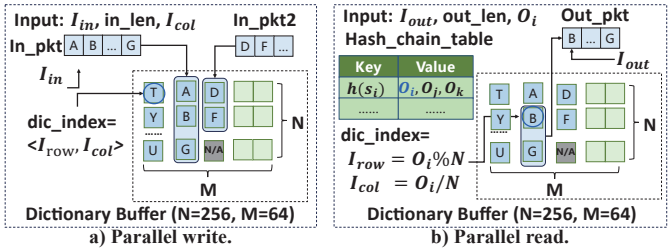

Fig. 5: The process of parallel reading from and writing to the striped dictionary buffer.

into the dictionary buffer within a single recirculation pass (as shown in the Fig. 4). Consequently, this approach increases memory bandwidth by a factor of N (e.g., N is set to 256 in our settings) and reduces encoding latency by approximately 98.4% compared to the single register array approach.

1) Distance-based sliding window management: Since the dictionary functions as a sliding window buffer, it is critical to distinguish between fresh data and stale data. Storing complete timestamps is memory-prohibitive. Instead, we propose a lightweight distance-based data freshness check mechanism. It verifies whether a candidate match retrieved from the hash table remains within the active window of size W (e.g., 64KB in our settings).

When processing an incoming byte stream, the system maintains a 32-bit write pointer O_{curr} that marks where new data is stored. When a hash lookup occurs, a candidate match position O_{match} is retrieved from the hash chain. The system computes the distance $\Delta = O_{curr} - O_{match}$ to verify the match. A match is valid if $\Delta \leq W$. If $\Delta > W$, the candidate is stale because its corresponding data has been overwritten. This mechanism enables lazy eviction and avoiding costly operations to explicitly delete old entries from the hash table.

2) Parallel register access: To achieve high-throughput dictionary management within the Tofino ASIC [41], we design a parallel access mechanism that maps a linear data stream to N independent physical register arrays, enabling scatter-write or gather-read of variable-length data in a single packet processing cycle.

Concurrent write. As shown in Fig. 5(a), when a packet arrives, its payload is stored in the PHV as a byte array indexed by I_{in} with length in_len. This data is written into a dictionary buffer organized as N registers (typically $N = 256$) within one cycle. Each byte is placed at a unique coordinate $\langle I_{\text{row}}, I_{\text{col}} \rangle$, where I_{row} (0 to $N-1$) identifies the register and I_{col} the position within it. Due to hardware limitations that disallow variable indexing of the PHV [41], a fixed mapping is employed (e.g., byte at $I_{\text{in}} = 0$ maps to register 0). For packets shorter than N bytes, unfilled positions are padded with a placeholder (e.g., `0xff`). A global pointer I_{col} tracks write progress across registers.

Concurrent read. The read module extracts a contiguous string from the dictionary buffer to an output buffer. During encoding, a hash value computed from the packet data queries a hash chain, returning one or more absolute indices O_i indicating candidate string locations in the dictionary (Fig. 5(b)).

The absolute index O_i is converted into the striped coordinate system $\langle I_{\text{row}}, I_{\text{col}} \rangle$:

$$I_{\text{row}} = O_i \bmod N \quad \text{and} \quad I_{\text{col}} = \frac{O_i}{N}. \tag{1}$$

Since N is a power of two, these operations reduce to bit-slicing: $I_{\text{row}} = O_i[\log_2 N - 1 : 0]$, $I_{\text{col}} = O_i[31 : \log_2 N]$. For example, when $N = 256$, $I_{\text{row}} = O_i[7:0]$, $I_{\text{col}} = O_i[31:8]$. The read module accepts three inputs: the starting output index I_{out}, the required length out_len, and the global starting index O_i. The bytes at the computed coordinates are gathered from the dictionary buffer and sequentially copied into the output buffer beginning at I_{out}.

C. Latency-aware Adaptive Encoder

As detailed in Section III-B, the dictionary buffer is distributed over N independent registers (for example, $N = 256$). When a packet arrives, a hash chain generates four candidate positions $S = \{s_1, s_2, s_3, s_4\}$. This corresponds to the LZ4 search depth of four [28], a parameter chosen to balance compression ratio with implementation efficiency [28]. As described in Section II, verifying a match requires reading four sequential bytes from each candidate. However, the Tofino ASIC [41] permits only one read per register per cycle, leading to conflicts when candidate accesses overlap. For example, if candidate s_1 reads registers 13 to 16 and candidate s_2 reads registers 15 to 18, registers 15 and 16 are accessed twice, preventing simultaneous fetching. This constraint necessitates a scheduling mechanism to resolve such conflicts.

1) Metrics modeling: We model the decision-making process as an optimization problem executed for every incoming data packet. **The optimization goal** is to minimize the total latency

$$\min \quad L = L_{proc} + L_{trans}, \tag{2}$$

which quantifies the trade-off between the processing delay incurred by recirculation and the transmission delay associated with sending uncompressed data.

The processing latency is modeled as

$$L_{proc} = \tau \cdot \max\Big(0, \max_{j:\, d_j \geq 2} \{d_j - 1\}\Big), \tag{3}$$

where τ is the delay per recirculation round. Let $D = \{d_1, d_2, d_3, d_4\}$ with each $d_j \in \{0,1,2,3,4\}$. Here, $d_j = 0$ means candidate j is fetched immediately (Round 0). $d_j = 1$ indicates that candidate j is dropped. $d_j = 2$ means candidate j is fetched in Recirculation Round 1. $d_j = 3$ means fetching in Recirculation Round 2, and $d_j = 4$ means fetching in Recirculation Round 3. A candidate scheduled for recirculation incurs an additional delay of $(d_j - 1)\tau$. The maximum delay corresponds to three rounds.

The transmission latency is defined as

$$L_{trans} = \beta \cdot \Big(U - \Delta^*\Big) + \gamma \cdot P_c\Big(G_{code}, E[L^*], B_{util}\Big), \tag{4}$$

where U is the uncompressed data unit size, Δ^* is the effective compression gain determined by the candidate providing the

maximum expected match length, β converts the compression gain into transmission latency savings, γ is a weighting coefficient, and P_c represents the probability of triggering congestion. In our offline evaluations, a congestion state is defined as one in which the instantaneous packet drop rate exceeds 10% [45], this condition is used to calibrate P_c.

Estimation of expected match length $E[L]$: For each candidate j, the expected match length $E[L_j]$ is derived from its candidate distance $Q_{dist}(j)$. Empirical analysis indicates that $E[L_j]$ decreases as $Q_{dist}(j)$ increases [46]. This relationship is modeled as $E[L_j] = f\big(Q_{dist}(j)\big)$ where $f(\cdot)$ is a monotonically decreasing function calibrated offline through statistical analysis. The effective expected match length used in the optimization is then defined as $E[L^*] = \max_{j\in T}\Big\{E[L_j]\Big\}$, where $T = \{j \mid d_j \neq 1\}$, i.e., the set of candidates that are not dropped. The compression gain for candidate j is defined as $\Delta_j = \max\{0, E[L_j] - L_{th}\}$, for a given threshold L_{th}, and the overall compression benefit is expressed as $\Delta^* = \max_{j\in T}\Big\{\Delta_j\Big\}$.

Model input: The input parameters combine real-time measurements and offline calibrations. In real time, the candidate distance Q_{dist} measures the dictionary index gap, where smaller values imply higher expected match lengths $E[L_j]$ due to temporal locality [38]. The six-bit conflict graph G_{code} encodes register overlaps using the condition $\text{Conflict}(A,B) = (S_A \leq S_B + 3) \wedge (S_B \leq S_A + 3)$, with each candidate occupying registers S through $S+3$. Additionally, the switch port bandwidth utilization B_{util} is monitored in real time, and the uncompressed data unit size U is fixed by design. Offline, parameters such as the recirculation latency τ and the transmission coefficients β and γ are calibrated.

Model output: The optimization yields the optimal decision vector D^*. Each candidate's decision is encoded as a 3-bit value for controlling the switch data plane. The final decision vector comprises 12 bits for all four candidates.

2) Solving the problem: Solving the non-linear optimization problem defined in Eq. (2) involves floating-point computations and iterative search procedures that are computationally infeasible for the switch ASIC at line rate. To bridge the gap between the theoretical model and hardware constraints, we adopt a hybrid architecture: offline pre-computation of optimal policies and online table-based execution.

State space discretization and online optimization table design: We discretize the continuous state into a compact 10-bit key for constant-time Match-Action Table lookups. The key comprises three fields: (1) a 6-bit conflict topology G_{code} representing candidate conflicts; (2) a congestion level B_{state} obtained by quantizing bandwidth utilization into discrete levels (e.g., two bits for Low, Medium, High, Critical); and (3) a quality profile Q_{class} mapping candidate distance Q_{dist} to priority classes. The composite key $\mathscr{K} = G_{code} \parallel B_{state} \parallel Q_{class}$ yields a state space of 1024 entries while occupying negligible SRAM.

Offline solver and online execution: A Python-based offline solver enumerates all $\mathscr{K}$ permutations, rebuilds conflict constraints, and estimates congestion probability P_c using calibrated parameters β, γ, τ. It evaluates all valid decision vectors D, selects D^* minimizing Eq. (2), and encodes it as a 3-bit action ID installed in the switch's scheduler at initialization. At runtime, the data plane computes hash indices, detects register conflicts to form G_{code}, and reads the traffic meter for B_{state}. These features are bit-packed into a query for the scheduler, which returns the precomputed D^* in a single pipeline stage.

Algorithm 1: Latency-aware adaptive encoder in data plane.

Input : Candidate distance Q_{dist}, Bandwidth utilization ratio B_{util}, Packet payload H
Output: Decision vector $D^* = meta.decision\_mask$

```
// 1) Feature extraction.
meta.B_state ← net_state(B_util) ;
meta.Q_class ← classifier_table.apply(Q_dist) ;
idx[1..4] ← find_hash_indices(H) ;
bit⟨1⟩ c_12 = conflict(idx[1], idx[2]) ;
bit⟨1⟩ c_13 = conflict(idx[1], idx[3]) ;
/*..... Repeat for all pairs*/;
meta.G_code = {c_12, c_13, ..., c_34} ;
// 2) Model application
meta.key = {meta.G_code, meta.B_state, meta.Q_class} ;
meta.decision_mask ← scheduler.apply(meta.key) ;
/*Determining registers' action based on
  meta.decision_mask*/;
```

D. Dependency-gated Parallel Decoder

Traditional LZ4 decoder [28] enforces packet-level serialization. Decoding for packet P_n starts only after packet P_{n-1} is fully processed. The previous packet provides the dictionary data for restoration. In programmable switching ASICs [41], this dependency creates a significant bottleneck [25, 47]. Packets propagate stage by stage in each packet processing cycle. For example, a Tofino [41] ASIC pipeline contains 12 stages. While one packet is processed at stage one, the subsequent packet enters stage zero. Following the traditional LZ4 design wastes pipeline resources and lowers processing efficiency.

1) Interleaved block-level processing: To overcome pipeline underutilization in serial packet processing, we propose a decoding architecture that decouples execution from packet boundaries. Our design enables concurrent decoding by shifting processing from the packet level to the block level. Encoded blocks from different flows or packets are decoded in parallel. This approach fully utilizes pipeline stages that would otherwise remain idle. If a block requires data that is not yet available, the packet is recirculated for later processing. This block level decoding achieves significant performance improvements over conventional packet level processing.

2) Version-Based Dependency Resolution: If a packet P_N references dictionary data carried by a preceding packet P_{N-1}, but P_{N-1} has not yet arrived, reading the dictionary memory would yield invalid or uninitialized data. To resolve these

dependencies without stalling the high-speed pipeline, we implement a *version-based dependency resolution mechanism*.

Every dictionary element carries a metadata field, `VersionID`. At the beginning of a flow, all `VersionID` values are initialized to zero. After a packet finishes writing its decoded payload to the dictionary, it updates the corresponding `VersionID` with its own `SequenceID`. When a packet later accesses a dictionary element, it compares the stored `VersionID` with the expected `Ref_SequenceID` carried in its header. If a packet P_N references dictionary data that should have been provided by a preceding packet P_{N-1} and P_{N-1} has not yet arrived, the entry in the dictionary may be invalid. The version-based mechanism prevents this error and allows the high-speed pipeline to continue without stalling.

IV. Implementation

We prototype HyperLink using CPU servers and programmable switches. On the CPU servers, RMMap is used for direct memory mapping, and we omit further details since it follows the standard implementation [4]. In this section, we focus on how compression is offloaded to programmable switches. The implementation consists of over 1.3K lines of Python for the switch control plane and over 5.2K lines of P4 for the programmable switch data plane.

Data plane. *1) The parallel striped storage module* Incoming payloads are segmented into fixed-size blocks and stored in a two-dimensional register array. Fixed read lengths are enforced, with unused bytes padded as needed. *2) Latency-aware adaptive encoder.* A precomputed decision table in the data plane uses a 10-bit key (i.e., 6-bit conflict vector, a 2-bit congestion level, and a 2-bit quality profile) to yield a 12-bit decision vector that directs immediate fetching, recirculation, or dropping of candidate data. *3) Dependence-gated parallel decoder.* A version lock table, keyed by sequence ID, maintains data consistency. An invalid flag in the packet header supports high-concurrency decoding. Due to Tofino's parsing limitations [41], packets larger than approximately 440bytes are fragmented into multiple sub-packets for cyclic processing, avoiding the overhead of sender-side segmentation.

Control plane. The control plane monitors performance metrics and status reports from the data plane via high-speed APIs, detecting anomalies like high recirculation, register conflicts, or versioning issues. Upon detection, it initiates exception handling by reconfiguring table entries or issuing alerts. Thus, while the data plane processes packets rapidly, the control plane ensures overall system stability.

Adaptive compression switching. A congestion-aware adaptive compression is adopted in HyperLink to achieve the lowest forwarding latency. When the host's congestion control (e.g., DCTCP [37]) detects congestion, HyperLink appends an instruction in the IP option field to trigger compression. Upon detecting this flag, the switch initiates the compression algorithm; if absent, no compression is performed. This design maintains low transmission latency across diverse conditions.

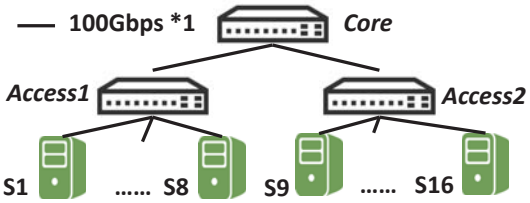

Fig. 6: Testbed. S denotes the server, Access and Core denote the switch.

V. Evaluation

We evaluate HyperLink through a testbed experiment, and compare it against two of the most relevant existing solutions, i.e., Protobuf [18], RMMap [4].

Testbed deployment. As shown in Fig. 6, the testbed comprises sixteen servers, two programmable switches equipped with a Tofino ASIC [41], and one commercial switch. The commercial switch acts as the core, while the programmable switches function as access switches running HyperLink to support network serialization. Each server is equipped with a 48-core AMD CPU, a 100Gbps NIC, and 300GB of memory. All links between servers and switches, as well as between switches, operate at 100Gbps.

Workloads setup. The workload employs eight servers on the left (see Fig. 6) hosting 128 partition containers and eight servers on the right hosting 128 prediction containers, with data flowing from partitions to predictions. In the ML prediction workflow, a pre-trained model processes 30MB of input images divided into 128 segments in parallel; the outputs are then merged to produce the final result. Each request entails a 240Mbps data transfer, and varying the RPS simulates different levels of concurrent traffic. For the microbenchmarks, we follow the methodology of RMMap [4] to construct various data types to evaluate the performance of existing works.

Results reveal that:

- HyperLink achieves the lowest latency: it reduces latency by up to 55.79% and 68.12% relative to RMMap [4] and Protobuf [18] under high RPS, respectively.
- HyperLink achieves highest throughput, and it significantly improves the throughput by 4.12× and 8.25× compared to RMMap [4] and Protobuf [18]. HyperLink consistently produces stable network transmissions, exhibits the lowest network jitter, and maintains the highest proportion of flows that achieve minimal flow completion times.

A. Testbed Experiments

Latency. Fig. 7 shows that over a range of 0 to 2400RPS, HyperLink achieves lower P99 latency than both RMMap [4] and Protobuf [18], reducing latency by up to 55.79% and 68.12% respectively. At 400RPS, the aggregated bandwidth of RMMap [4] nears 100Gbps with a bottleneck between the access and core switches. Under these conditions, RMMap [4] triggers sender-side congestion control that slows down data transmission rate, enlarging the latency. Protobuf [18] incurs high CPU overhead from compute-intensive serialization and

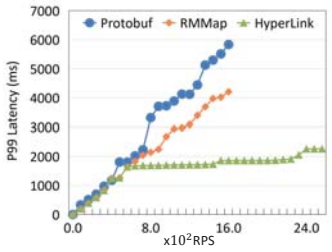

Fig. 7: [Testbed] P99 latency.

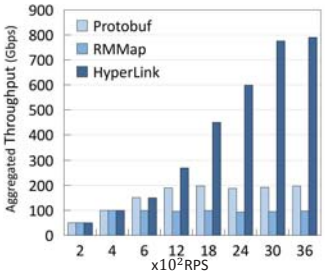

Fig. 8: [Testbed] Aggregated throughput.

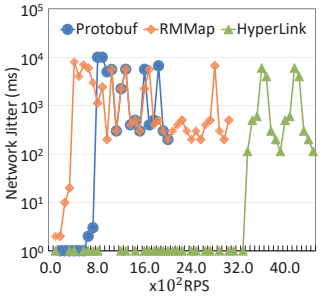

Fig. 9: [Testbed] Jitter.

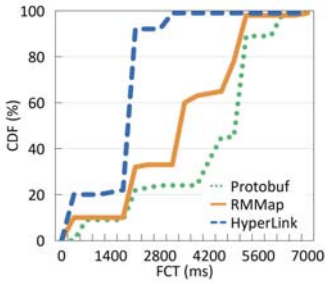

Fig. 10: [Testbed] Flow completion time.

deserialization. As RPS increases beyond 800RPS, rising CPU utilization and, above 420RPS, approaching the link's bandwidth threshold further increase Protobuf's [18] latency due to network congestion. In contrast, HyperLink employs in-network data compression with fast, lightweight encoding and decoding on a programmable switch to maintain low transmission latency.

Throughput. Fig. 8 shows the throughput results under various concurrent request scenarios. Experimental results indicate that HyperLink consistently achieves higher throughput than both Protobuf [18] and RMMap [4]. In particular, HyperLink improves throughput by up to 4.12× relative to RMMap [4] and by up to 8.25× compared to Protobuf [18]. When the RPS exceeds 400, the transmission link approaches congestion, leading to packet loss, retransmissions, and sender rate throttling that lower RMMap's [4] throughput. In contrast, HyperLink leverages in-network computation to reduce the transmitted data volume by around 88.1%, thereby increasing effective throughput by nearly eight times. Although Protobuf [18] also employs compression, its compute-intensive serialization and deserialization prolong the transmission time and result in lower throughput.

Jitter. Fig. 9 shows the jitter results for different concurrent request scenarios. Jitter is defined as the absolute difference between per-flow delay measurements and their average. The experiments show that HyperLink consistently achieves lower network jitter than Protobuf [18] and RMMap [4]. HyperLink reduces jitter by up to 98.2% compared to RMMap [4] and up to 99.3% compared to Protobuf [18]. When the transmission data exceeds the link capacity, the link undergoes repeated congestion cycles. The sender throttles its rate, which increases jitter. For RMMap [4], the RPS threshold is nearly 400. The

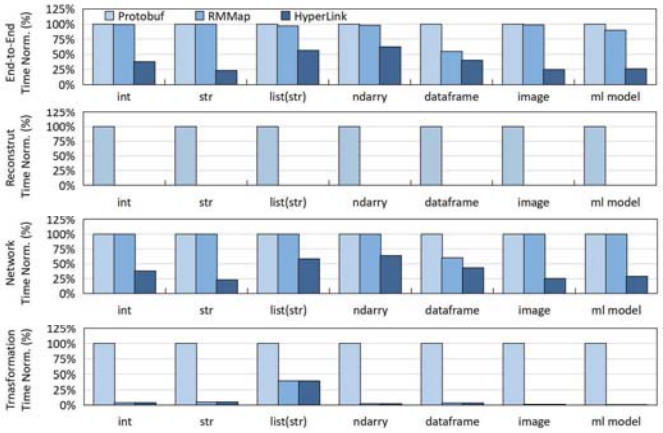

Fig. 11: [Microbenchmark]. Latency breakdown.

thresholds are nearly 798 for Protobuf [18] and 3300 for HyperLink. The efficient compression in HyperLink enables it to tolerate a higher number of concurrent requests and to maintain a stable latency up to 3300RPS.

Flow completion time. Fig. 10 shows the flow completion time results under various concurrent request scenarios. We evaluated HyperLink, RMMap, and Protobuf, across 41 scenarios with RPS ranging from 100 to 4100 in increments of 100RPS, with each scenario running for one minute. The flow completion times from all scenarios were aggregated into a distribution plot. The experimental results indicate that HyperLink consistently achieves lower flow completion times than both Protobuf [18] and RMMap [4]. HyperLink reduces the average flow completion time by up to 47.7% compared with RMMap [4] and by up to 57.8% compared with Protobuf [18].

B. Microbenchmark Experiments

Fig. 11 presents the end-to-end transmission delay for different data types and shows the delay breakdown for each module of HyperLink, RMMap [4], and Protobuf [18]. In this experiment, the transformation time for HyperLink and RMMap [4] is the period required to mark memory as copy-on-write. For Protobuf [18], the transformation time equals the serialization time. Network time represents the delay during data transmission. For HyperLink, network time also includes the delay for data compression and decompression, which is performed on programmable network switches. Reconstruct time is defined as the remote mapping time for HyperLink and RMMap [4] and as the deserialization time for Protobuf [18]. The experiments were conducted at 1000RPS.

Results show that HyperLink consistently achieves the lowest end-to-end delay across all data types. For the int data type, HyperLink reduces the overall delay by up to 62.14% compared with Protobuf [18] and by up to 61.83% compared with RMMap [4]. This gain stems from the high redundancy of int data, which enables LZ4 to achieve high compression ratios and significantly reduce transmitted data volume. In network transmission, HyperLink incurs up to 61.84% lower delay than RMMap [4] for the int data type, as RMMap [4] suffers from

link congestion and sender throttling at 1000RPS. In the transformation and reconstruction phases, HyperLink reduces delay by up to 99.89% for the machine learning model data type compared with Protobuf [18], since heavy serialization and deserialization under high concurrency significantly increase CPU overhead, whereas HyperLink employs lightweight direct memory mapping on the host side.

VI. Related Work

Optimizing serverless computing. Serverless optimizations fall into three categories. The first accelerates function startups by reducing cold-start delays through remote forking and sandboxing (e.g., MITOSIS [12], SAND [13], Catalyzer [14], SOCK [15]), but still incurs significant serialization costs. The second enhances stateful functions with fault tolerance via logging and transactional mechanisms (e.g., Halfmoon [16], Beldi [17]), introducing additional logging and serialization overhead. The third addresses cold-start and variability in serverless DAGs while relying on conventional serialization (e.g., ORION [3]). In contrast, HyperLink reduces the dominant serialization overhead by offloading *compute-intensive* compression to programmable switches, maintaining only lightweight memory mapping on the host.

Optimization of (de)serialization in serverless computing. Prior work employs hardware-based methods [22, 23, 35, 48, 49] or system-level approaches [25, 50–54]. Hardware solutions (e.g., Cereal [35], Cornflakes [23], Naos [22], RpcNIC [48], TOPPINGS [49]) offload CPU tasks using accelerators or NICs, but require extra devices on every server, which is cost-prohibitive at scale. In contrast, HyperLink upgrades only a limited number of programmable access switches, each costing nearly half as much as a server [55]. System-level approaches (e.g., ZOT [50], TeraHeap [51], Gerenuk [52], OPrime [25], DShuffle [53], CXLfork [54]) optimize processing solely on the server, whereas HyperLink leverages both server and network resources by offloading compression to switches while keeping lightweight memory mapping on the server.

In-network data compression. Only a few approaches achieve in-network compression. Flightplan [56] and Loom [57] compress packet headers using simplified Van Jacobson compression [58] and Bloom filter techniques [59]. THC [60] and SwitchTop-*k* [61] compress gradient tensors in distributed deep learning. In contrast, HyperLink targets lossless compression of payload data in serverless scenarios.

VII. Conclusion and Future Works

This paper introduces HyperLink, a *network-assisted* system that accelerates data exchange in serverless computing by combining *computation*-bound and *bandwidth*-focused methods. HyperLink offloads data compression to programmable switches and uses lightweight memory mapping on the host. Its design features a parallel storage module for dictionary management, a latency-aware adaptive encoder, and a dependency-gated parallel decoder. Prototyped on sixteen servers and two programmable switches, HyperLink reduces latency by up to 68.12% and boosts throughput by a factor of eight compared to current solutions.

In the future, there are several directions to explore. First, improving the compression ratio by merging data across consecutive packets. Merging the head of a later packet with the tail of its preceding packet may reveal longer matching strings. Second, incorporating a priority mechanism for operators. This mechanism enables high-priority flows to be compressed earlier, thus preserving the efficiency of more valuable flows.

Acknowledgment

This work is supported by the National Key R&D Program of China (No. 2025YFE0204100), Science and Technology Development Fund of Macao S.A.R (FDCT) under number 0074/2025/AMJ, Guangdong Basic and Applied Basic Research Foundation (No. 2026A1515011754, No. 2023B1515130002), Key Research and Development and Technology Transfer Program of Inner Mongolia Autonomous Region (2025YFHH0110), Shenzhen Basic Research Program (No. JCYJ20250604183035046, No. KJZD20230923113800001).

References

[1] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proceedings of USENIX ATC*, 2020.

[2] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *Proceedings of USENIX ATC*, 2021.

[3] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in *Proceedings of USENIX OSDI*, 2022.

[4] F. Lu, X. Wei, Z. Huang, R. Chen, M. Wu, and H. Chen, "Serialization/deserialization-free state transfer in serverless workflows," in *Proceedings of ACM EuroSys*, 2024.

[5] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Boki: A stateful serverless workflow engine," in *Proceedings of ACM SOSP*, 2022.

[6] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM SoCC*, 2019.

[7] M. Shahrad, J. Balkind, and D. Wentzlaff, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys*, vol. 54, no. 10s, 2021.

[8] Z. Zhuang, W. Liu, Q. Qian, Z. Li, C. Xie, X. Xiong, Z. Wu, Y. Zhou, and C. Yang, "FaaSFlow: Enable efficient workflow execution for Function-as-a-Service," in *Proceedings of ACM SoCC*, 2021.

[9] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "ServerlessLLM: Low-Latency serverless inference for large language models," in *Proceedings of USENIX OSDI*, 2024.

[10] Datadog, "The state of serverless 2021," 2021. [Online]. Available: https://www.datadoghq.com/state-of-serverless/

[11] Y. Luo, X. Ru, K. Liu, L. Yuan, M. Sun, N. Zhang, L. Liang, Z. Zhang, J. Zhou, L. Wei *et al.*, "Oneke: A dockerized schema-guided llm agent-based knowledge extraction system," in *Proceedings of ACM WWW*, 2025.

[12] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen, "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," in *Proceedings of USENIX OSDI*, 2023.

[13] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance serverless computing," in *Proceedings of USENIX ATC*, 2018.

[14] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of ACM ASPLOS*, 2020.

[15] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning

with Serverless-Optimized containers," in *Proceedings of USENIX ATC*, 2018.
[16] S. Qi, X. Liu, and X. Jin, "Halfmoon: Log-optimal fault-tolerant stateful serverless computing," in *Proceedings of ACM SOSP*, 2023.
[17] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *Proceedings of USENIX OSDI*, 2020.
[18] Google, "Protocol buffers - Google's data interchange format," 2024, language-neutral, platform-neutral extensible mechanism for serializing structured data. [Online]. Available: https://github.com/protocolbuffers/protobuf
[19] GNU Project, "Gzip," 2024, general file compression tool using DEFLATE algorithm; Git mirror at https://github.com/mirror/gzip. [Online]. Available: https://www.gnu.org/software/gzip/
[20] Google, "Snappy: A fast compressor/decompressor," 2024, fast compression library optimized for speed (250+ MB/s compress, 500+ MB/s decompress). [Online]. Available: https://github.com/google/snappy
[21] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus prime: Accelerating data transformation in servers," in *Proceedings of ACM ASPLOS*, 2020.
[22] K. Taranov, R. Bruno, G. Alonso, and T. Hoefler, "Naos: Serialization-free RDMA networking in java," in *Proceedings of USENIX ATC*, 2021.
[23] D. Raghavan, S. Ravi, G. Yuan, P. Thaker, S. Srivastava, M. Murray, P. H. Penna, A. Ousterhout, P. Levis, M. Zaharia, and I. Zhang, "Cornflakes: Zero-copy serialization for microsecond-scale networking," in *Proceedings of ACM SOSP*, 2023.
[24] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang, S. Zhang, M. J. Fernandez, S. Gandham, and H. Zeng, "Rdma over ethernet for distributed training at meta scale," in *Proceedings of ACM SIGCOMM*, 2024.
[25] Z. Chen, Y. Feng, S. Liu, H. Song, H. Zhou, T. Yun, W. Xu, T. Pan, and B. Liu, "Optimusprime: Unleash dataplane programmability through a transformable architecture," in *Proceedings of the ACM SIGCOMM*, 2024.
[26] S. Chen, R. Jiang, D. Yu, J. Xu, M. Chao, F. Meng, C. Jiang, W. Xu, and H. Liu, "KVDirect: Distributed disaggregated LLM inference," *arXiv preprint arXiv:2501.14743*, 2024.
[27] J. Lou, X. Kong, J. Huang, W. Bai, N. S. Kim, and D. Zhuo, "Harmonic: Hardware-assisted RDMA performance isolation for public clouds," in *Proceedings of the USENIX NSDI*, 2024.
[28] Y. Collet *et al.*, "Lz4: Extremely fast compression algorithm," 2024, official repository; LZ4 home page at https://www.lz4.org/. [Online]. Available: https://github.com/lz4/lz4
[29] AWS, "Aws lambda," 2023, accessed: 2023. [Online]. Available: https://aws.amazon.com/lambda
[30] Microsoft, "Azure functions," 2023, accessed: 2023. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/
[31] A. Cloud, "Alibaba serverless application engine," 2023, accessed: 2023. [Online]. Available: https://www.aliyun.com/product/aliware/sae
[32] Huawei, "Huawei cloud functions," 2023, accessed: 2023. [Online]. Available: https://developer.huawei.com/consumer/en/agconnect/cloud-function/
[33] AWS, "Finra adopts aws to perform 500 billion validation checks daily," 2023. [Online]. Available: https://aws.amazon.com/solutions/case-studies/finra-data-validation/
[34] J. Kim and K. Lee, "Practical cloud workloads for serverless faas," in *Proceedings of the ACM SoCC*, 2019.
[35] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee, "A specialized architecture for object serialization with applications to big data analytics," in *Proceedings of ACM/IEEE ISCA*, 2020.
[36] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving," in *Proceedings of USENIX ATC*, 2019.
[37] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM*, 2010, pp. 63–74.
[38] K. Sayood, *Introduction to Data Compression*, 5th ed. Cambridge, MA: Morgan Kaufmann, 2017.
[39] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of IEEE IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
[40] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.
[41] Intel, "Intel intelligent fabric processors," Online: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors.html, 2024.
[42] J. loup Gailly and M. Adler, "zlib: A massively spiffy yet delicately unobtrusive compression library," 2024, official repository; zlib home page at https://zlib.net/. [Online]. Available: https://github.com/madler/zlib
[43] L. Collin *et al.*, "Xz utils," 2024, official repository; XZ Utils home page at https://tukaani.org/xz/. [Online]. Available: https://github.com/tukaani/xz
[44] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
[45] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Congestion control for Web real-time communication," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2629–2642, oct 2017.
[46] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the TCP congestion avoidance algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997.
[47] Y. Yang, L. He, J. Zhou, X. Shi, J. Cao, and Y. Liu, "P4runpro: Enabling runtime programmability for rmt programmable switches," in *Proceedings of the ACM SIGCOMM*, 2024.
[48] J. Zhang, H. Huang, X. Chen, X. Li, J. Zhao, M. Liu, and Z. Wang, "RpcNIC: Enabling efficient datacenter RPC offloading on PCIe-attached SmartNICs," in *Proceedings of IEEE HPCA*, 2025.
[49] S. Li, H. Lu, T. Wu, M. Yu, Q. Weng, X. Chen, Y. Shan, B. Yuan, and W. Wang, "TOPPINGS: CPU-assisted, rank-aware adapter serving for LLM inference," in *Proceedings of USENIX ATC*, 2025.
[50] M. Wu, S. Wang, H. Chen, and B. Zang, "Zero-Change object transmission for distributed big data analytics," in *Proceedings of USENIX ATC*, 2022.
[51] I. G. Kolokasis, G. Evdorou, S. Akram, C. Kozanitis, A. Papagiannis, F. S. Zakkak, P. Pratikakis, and A. Bilas, "TeraHeap: Reducing memory pressure in managed big data frameworks," in *Proceedings of ACM ASPLOS*, 2023.
[52] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu, "Gerenuk: thin computation over big native data using speculative program transformation," in *Proceedings of ACM SOSP*, 2019.
[53] C. Ding, S. Li, K. Lu, T. Yao, D. Wang, H. Wu, J. Wan, Z. Tan, and C. Xie, "DShuffle: DPU-Optimized shuffle framework for large-scale data processing," in *Proceedings of USENIX ATC*, 2025.
[54] C. Alverti, S. Psomadakis, B. Ocalan, S. Jaiswal, T. Xu, and J. Torrellas, "CXLFork: Fast remote fork over cxl fabrics," in *Proceedings of ACM ASPLOS*, 2025.
[55] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Proceedings of the USENIX NDSS*, 2020.
[56] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo, "Flightplan: Dataplane disaggregation and placement for p4 programs," in *Proceedings of USENIX NSDI*, 2021.
[57] J. Zhang, Y. Gao, S. Wen, T. Pan, and T. Huang, "Loom: Switch-based cloud load balancer with compressed states," in *Proceedings of IEEE ICNP*, 2021.
[58] V. Jacobson, "Compressing TCP/IP headers for low-speed serial links," 1990.
[59] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, p. 422–426, 1970.
[60] M. Li, R. B. Basat, S. Vargaftik, C. Lao, K. Xu, M. Mitzenmacher, and M. Yu, "THC: Accelerating distributed deep learning using tensor homomorphic compression," in *Proceedings of USENIX NSDI*, 2024.
[61] Y. Li, J. Huang, J. Liu, Z. Li, W. Jiang, and J. Wang, "SwitchTop-k: Scaling top-k compression on programmable switches," in *Proceedings of ACM KDD*, 2025.