Honey, I Shrunk the Headers with Flow.ZIP

Yinda Zhang
University of Pennsylvania
Philadelphia, PA, USA

Liangcheng Yu
Microsoft Research
Redmond, WA, USA

Gianni Antichi
Politecnico di Milano and
Queen Mary University of London
Milan, Italy

Ran Ben Basat
University College London and
Broadcom
London, UK

Vincent Liu
University of Pennsylvania
Philadelphia, PA, USA

Abstract

Packet header overhead is a persistent source of inefficiency in packet-switched networks, reducing goodput and increasing network load. Trends like tunneling further increase this overhead, significantly impacting flow completion times. While, in principle, it is possible to compress these headers, existing methods require specialized hardware on every hop to compress/decompress the packet to/from custom header formats.

In this paper, we present Flow.ZIP, a backward-compatible header compression mechanism designed for existing data center networks. Our solution leverages a combination of last-hop network offload and MPLS support, both of which are ubiquitous in modern data center deployments. Flow.ZIP overcomes scalability limitations in these components by selectively and intelligently coordinating compression for a subset of flows. Doing so, Flow.ZIP achieves up to 58% reduction in average flow completion time on real-world data center workloads.

CCS Concepts

• **Networks** → **Network protocol design**; *Data center networks*; *Programmable networks.*

Keywords

Header Compression; Network Overhead, MPLS; VXLAN; Data Center Networks

ACM Reference Format:
Yinda Zhang, Liangcheng Yu, Gianni Antichi, Ran Ben Basat, and Vincent Liu. 2026. Honey, I Shrunk the Headers with Flow.ZIP. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3789240.3829104

1 Introduction

Since the advent of packet-switching, headers have remained a fundamental overhead in modern networks: they appear on every packet, reducing goodput and increasing load [1–3].

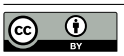
The potential overheads have only increased over time. An IPv6 header occupies 40 B, and in cloud networks that use tunneling for virtual networking and tenant isolation [4], common protocols like VXLAN [5] and NVGRE [6] layer multiple encapsulation headers, which can add an additional 20 or more bytes per packet. Table 1 shows aggregate header overheads for some common configurations; in many cases, these overheads can exceed 100 B.

A growing body of work [1, 2, 7] shows that increases in per-packet overheads—even modest ones—can have disproportionately large performance impacts. For instance, prior work [1] reports that adding 28 bytes of extra header metadata can raise flow completion time (FCT) by ∼10% under light network load, while removing 40 bytes can shorten FCT by ∼30% [3]. Our experiments on realistic data center workloads (Section 2) expand on those results, demonstrating that at higher loads like those regularly experienced in AI collective communication, the overhead introduced by IPv6 and TCP headers (60 B) increases the average FCT by up to 50% compared to an ideal case without header overhead. When additional tunneling (e.g., VXLAN) is applied, FCT increases by up to 2.7×. Similarly, recent proposals for Scale-Up Ethernet (SUE) [8, 9] highlight the inefficiency of modern packet headers for AI and HPC workloads, which consist largely of small memory load/store messages.

This disproportionate impact arises from the nonlinear relationship between FCT and network load. As an example of this effect, consider a queuing theoretic M/G/1 system. The average completion time scales as $\frac{1}{1-\rho}$, where ρ denotes the system load [10, 11]—a hyperbolic relationship. In the case of VXLAN, which can increase effective load by 13% (Section 2.1), a network with 80% initial load would suffer an approximately 2.9× increase in FCT according to this model. This is before other factors, such as congestion control and higher ECN marking rates [12, 13], pause-induced backoff [14], and increased per-packet processing overhead [1, 15].

The most direct approach to addressing these issues is to compress packet headers [16, 17]. There is a long line of work on this topic, including ROHC [18], CRTP [19], and IPHC [20]. Unfortunately, these generally operate at Layer-2, requiring Layer-3 devices to maintain per-flow compression state and to decompress and recompress headers at each hop to recover routing-critical fields. Modern switches [21] cannot support such per-flow state at datacenter scale due to limited on-chip memory [22, 23]. Moreover, the compression and decompression overhead, at billions of pps, is infeasible on today's switches. As a result, a general, practical, and deployable solution to packet header bloat remains elusive.

In this paper, we present Flow.ZIP, a header compression mechanism designed for practical and backward-compatible deployment



SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/2026/08
https://doi.org/10.1145/3789240.3829104

Encap	Layer Components	Overhead (B)
TCP	IPv6 + TCP	60
RoCE v2	IPv6 + UDP + IB.BTH	60
+NVGRE	IPv6 + GRE + Inner	122
+VXLAN	IPv6 + UDP + VXLAN + Inner	130

Table 1: Typical header overheads introduced by common L3/L4 encapsulation (using IPv6 as an example).

on existing cloud and data center networks. Unlike prior approaches that require modified switches or per-hop compression/decompression at every Layer 3 device, Flow.ZIP leverages existing protocols in the middle of the network and performs all packet manipulation at the edge.

Flow.ZIP's requirements are for a compressor at the sender side and a decompressor at the receiver side, implemented either in the host network stack or offloaded to last-hop hardware such as a DPU or a programmable switch. In the middle of the network, Flow.ZIP repurposes Multiprotocol Label Switching (MPLS) [24], a lightweight and widely supported protocol in modern routers [25–28]. Although primarily intended for traffic engineering in wide-area networks [28, 29], the MPLS header is attractive for its ubiquitous support and small footprint: only 4 bytes, i.e., one to two orders of magnitude smaller than the header combinations of Table 1.

Of course, direct, wholesale usage of MPLS is not feasible. The naive approach of assigning a unique MPLS label to every flow and replacing full packet headers with compact MPLS labels cannot be used at data center scales. Most commodity switches support only around 16K MPLS label entries per device [25, 26], while cloud and data center networks often carry millions of concurrent flows [30].

To address this limitation, Flow.ZIP implements a distributed cache of active labels. Given the long-tailed nature of traffic in cloud and data center traffic [12, 30, 31], Flow.ZIP's approach of dynamically compressing high-volume flows and aggressively reclaiming inactive labels results in substantial cumulative header-size reductions despite the extremely limited MPLS table capacity (e.g., 16K entries) of commodity switches [25–27].

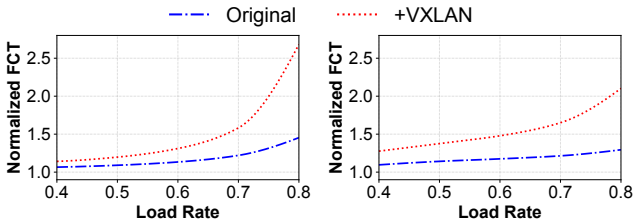
We implement a Flow.ZIP prototype[1] and evaluate it using both a hardware testbed and large-scale ns-3 simulations. We further augment our evaluation with a theoretical analysis of Flow.ZIP (Section 6) that characterizes the effectiveness of its selective compression policy under skewed traffic and closely matches our empirical results. Flow.ZIP reduces the average FCT by up to 58%, across diverse network configurations, traffic patterns (e.g., continuous flows, incast, etc.), protocol stacks (TCP and RDMA), and tunneling setups. Its control-plane overhead is also low: tracking multiple Tbps of traffic requires under 0.5 Gbps of control plane traffic.

Ethics. This work does not raise ethical issues.

2 Background and Motivation

We begin by examining the impact of packet headers on end-to-end performance (Section 2.1), then summarize existing header compression designs and explain why they remain impractical for general cloud and datacenter networks (Section 2.2).

[1]Code is available at https://github.com/eniac/Flow.ZIP



(a) TCP/DCTCP. **(b) RoCEv2/DCQCN.**

Figure 1: Impact of header overhead on average FCT, normalized to the ideal case without L3/L4 header overhead.

2.1 Packet Header Overhead

Most cloud and data center traffic is carried in IP, TCP, and UDP headers. Over the years, due to scale and isolation concerns, providers have moved toward larger headers, e.g., larger addresses like IPv6 [32] and tunneling protocols like VXLAN [5] and NVGRE [6]. As shown in Table 1, such encapsulation can push total L3/L4 overheads to 122–130 bytes, amounting to more than 15% of an average-sized 800-byte packet; much more for small packets (e.g., 256-byte memory load/store messages in Scale-Up Ethernet [8, 9]).

We quantify this effect through simulation of a FatTree topology [33] and representative production storage workload [14], as described in Section 8.1. Figure 1 compares the average flow completion time (i.e., the time from when the first packet of a flow is sent until the last packet is acknowledged [34]) across different network loads (i.e., the ratio of traffic to network capacity) under three configurations: (1) with standard 60 B IPv6 and TCP headers, (2) IPv6-based VXLAN tunneling (an additional 70 B), and (3) an idealized lower bound that assumes zero L3/L4 header overhead, i.e., only payloads are transmitted over raw Ethernet.

The results confirm that header overhead inflates FCT substantially, especially under high load. In the baseline configuration at 50% load, FCT increases by roughly 9% over the idealized configuration; at 80% load, the gap widens to 45%. VXLAN amplification makes the gap more pronounced—FCT reaches 2.7× the ideal baseline at 80% load.

As mentioned in Section 1, these disproportionate impacts can be partially understood using queuing theoretic models. For instance, consider an M/G/1 network [10], a simplified but widely used abstraction for analyzing FCT [11, 35, 36] that imposes no assumptions on the flow-size distribution. In these networks, even a small increase in load can cause an outsized increase in FCT, especially as the system approaches full utilization. This relationship is formalized below.

THEOREM 1. *In an M/G/1-PS queue, an increase of δ in load results in at least a $\frac{\delta}{1-\rho-\delta}$ relative increase in average completion time, where ρ is the original load rate.*

PROOF. The average completion time in an M/G/1-PS queue, from [10], is $T(\rho) = \frac{\alpha}{1-\rho}$, where α is a constant determined by service and arrival characteristics. Thus, the relative increase in completion time after a load increment of δ is

$$\frac{T(\rho+\delta)-T(\rho)}{T(\rho)} = \frac{\frac{\alpha}{1-\rho-\delta} - \frac{\alpha}{1-\rho}}{\frac{\alpha}{1-\rho}} = \frac{\delta}{1-\rho-\delta} \quad . \quad \square$$

This result highlights the nonlinear sensitivity of FCT to load: beyond the direct δ increase, the factor $\frac{1}{1-\rho-\delta}$ exhibits hyperbolic growth as the system nears saturation. For an average packet size of 800 bytes and a VXLAN header overhead of 130 bytes, the induced load increase δ is roughly 13% under an initial 80% load. Applying Theorem 1, this overhead can inflate FCT by approximately 2.9× relative to the ideal case. Real networks have more complex arrival rates and network topologies, but the intuition holds.

Beyond these effects, higher load also triggers more pause messages, increases ECN marking, and forces additional transport backoff [12–14], compounding FCT even further. These are on top of higher serialization/processing latency at each hop [1, 15] and increased bit-level corruption [37], which trigger retransmissions and uneven latency spikes.

Importantly, many of these effects can be triggered even without high average network utilization. As shown in Section 8.1, transient and localized load spikes caused by incast traffic can produce similar FCT inflation.

2.2 Existing Solutions and Limitations

Header compression is a promising technique for reducing header overheads without application modifications. Unfortunately, existing mechanisms (e.g., ROHC [18], CRTP [19], and IPHC [20]) largely require *every* Layer 3 device to:

- R1 Compress outgoing packets and decompress incoming packets to/from custom header formats;
- R2 Maintain per-flow state to reconstruct original packets.
- R3 Provide sufficient data-plane programmability to operate on that state at line rate.

As a concrete example, RFC 3095 [18] defines the classic Robust Header Compression (ROHC) scheme, in which the compressor and decompressor each maintain a copy of the original uncompressed header for every connection; packets then carry only a connection identifier and the delta from this reference header. To tolerate errors and packet reordering under differential compression, the two ends must periodically synchronize their per-flow state. Although effective in its intended bandwidth-constrained Layer-2 settings (e.g., wireless and IoT networks), ROHC does not scale to multi-terabit datacenter fabrics, which routinely carry millions of concurrent flows. Each ROHC flow state requires hundreds of bytes, far beyond what current switches can store given their limited on-chip memory (e.g., 10s MB) [22, 23]. Moreover, the required differential compression and decompression at billions of packets per second is unsupported by today's merchant-silicon switches, and would require fundamental switch redesign [16]. Other header compression algorithms experience similar limitations [20].

We note that jumbo frames can potentially mitigate the above issues by amortizing the overhead across larger payloads. However, this approach has several limitations: (1) not all networks support jumbo frames, particularly those found in enterprise data center networks [1] or those that go beyond TCP/IP (e.g., older RNICs support only 1–2 KB MTUs, and many recent RoCEv2 NICs are capped at 4 KB MTUs [38]); (2) jumbo frames can negatively impact short flows, as they introduce additional head-of-line blocking, which can degrade FCT [39]; and (3) there are still benefits to reducing header overheads, both for the larger frames and the smaller ones that hosts inevitably need to send.

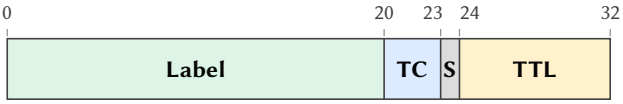


Figure 2: Format of the MPLS header (4 bytes)

The goal of this work is a general and practical solution to reduce header overhead that preserves compatibility with currently deployed network hardware.

3 MPLS: Constraints/Opportunities

We observe that modern routers, even those designed for data centers (e.g., Broadcom Tomahawk and Cisco G200) [40–43], almost universally provide native support for an alternative Layer-3 protocol in *Multiprotocol Label Switching (MPLS)* [24]. MPLS, in addition to being widely supported [25–27], also has the distinction of having among the shortest headers of any routable protocol at just 4 bytes, $\frac{1}{10}$ of IPv6.

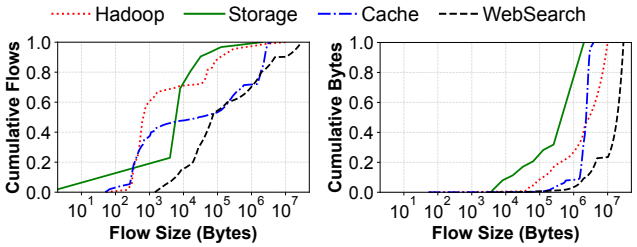
Header Format (Figure 2). An MPLS header contains:

(1) *Label [20 bits].* Used by switches to route packets. A packet may carry multiple MPLS headers (forming a stack), but the switch primarily examines the top label to make forwarding decisions and perform operations such as swapping, pushing, or popping MPLS headers.
(2) *Traffic Class [3 bits].* Records the packet's QoS priority or ECN information.
(3) *Bottom of Stack field [1 bit].* Indicates whether the current header is the last in the MPLS stack.
(4) *Time to Live [8 bits].* Functions similarly to the TTL field in IP, preventing packets from looping indefinitely.

Traditional MPLS operation. MPLS is primarily designed for traffic engineering in wide-area networks [28, 29, 44]. In the data plane, when an IP packet enters the MPLS domain, an ingress label edge router uses routing information to assign and prepend one or more MPLS headers to the packet. The labeled packet is then forwarded along a pre-established path toward its destination. At the egress point, another label edge router removes the MPLS headers and forwards the resulting IP packet using standard IP forwarding rules. In the control plane, MPLS is agnostic to the label distribution protocol, and so rules can be installed using a variety of mechanisms, both distributed and centralized [29, 45].

Strawman solution and its limitations. A natural strawman solution for lossless header compression is to use MPLS labels to fully replace the original packet headers, à la source routing. In this design, a unique label is assigned to each flow at the sender; switches perform routing based directly on the label, and the receiver reconstructs the full header using the label and a saved copy of the original header.

Unfortunately, this approach has a severe bottleneck in the form of the MPLS capacity of the intermediate commodity switches. The MPLS label field is only 20 bits wide, allowing for a theoretical maximum of about one million unique labels. In practice, commodity switches typically support only around 16K MPLS label entries



(a) Weighted by flows. (b) Weighted by bytes.

Figure 3: Cumulative distribution of flow sizes.

per device [25–27]. Given that modern data center networks often carry millions of concurrent flows [30], this constrained label space falls far short of what is required for header compression.

Opportunity. In this paper, we argue that universal compression is unnecessary and that *selective compression* is a more feasible path toward practical header compression.

Figure 3 provides intuition for why this is the case by presenting four real-world workload traces from production data centers [12, 14, 30]. These workloads reveal skewed distributions, with a small subset of flows accounting for a disproportionate share of total traffic volume. For example, in Hadoop [30], fewer than 3% of flows are larger than 1 MB, but this small fraction of flows accounts for over 70% of the total bytes transmitted. Similar patterns have also been observed in other data center environments [12, 31].

Compounding these benefits is the fact that establishing new connections (e.g., for both TCP and RDMA) is often expensive, so applications tend to reuse existing connections for multiple flows and tasks [36, 46], further concentrating traffic within a smaller set of active connections.

4 Design Overview

Flow.zip is a practical, general, and backward-compatible framework for selective header compression. Because a few flows dominate the overall traffic frequency and volume, focusing on them allows Flow.zip to reduce a significant portion of header overhead while leaving other flows untouched, preserving compatibility and minimizing the need for excessive label entries. In designing Flow.zip, we needed to solve several technical challenges:

(C1) How should we encode today's complex L3/4 headers into the limited space of the MPLS header in a way that preserves essential routing semantics, and without imposing significant complexity on end-hosts? (C2) How do we manage the lifecycle of labels given the severe limitations of modern switch memory capacity while maintaining consistency, correctness, and high compression efficiency? (C3) How can we ensure robustness and maintain correct behavior in the presence of network and controller failures?

Traditional uses of MPLS, e.g., traffic engineering, operate on much coarser granularity and over much larger timescales (to allow time for the scheduling algorithm to execute over WAN flows) [28, 47, 48]. Flow.zip is bottlenecked by label capacity rather than scheduling overheads, necessitating a different set of solutions to the above challenges.

Components. The high-level architecture of Flow.zip is shown in Figure 4. It consists of three main components.

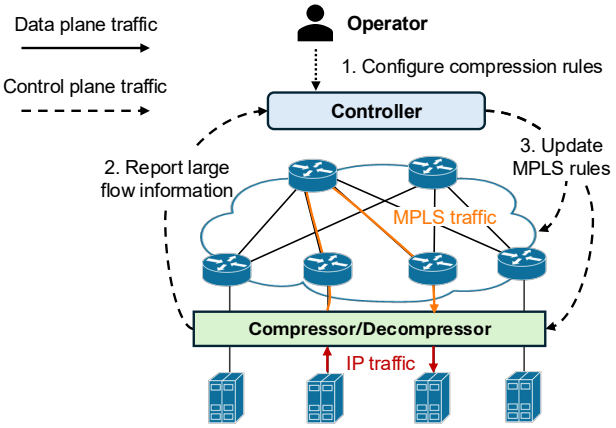


Figure 4: Architecture of Flow.zip.

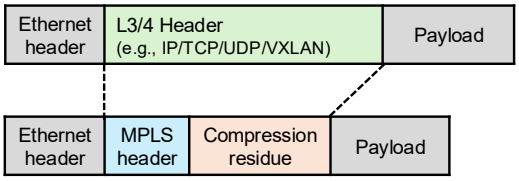


Figure 5: Compression in Flow.zip.

Switches: Flow.zip assumes that all switches in the network are MPLS-enabled and can be controlled remotely. The MPLS processing is on top of the existing IP routing configuration, which serves as the default and fallback routing mechanism.

Compressor/Decompressor: On the sender side, the compressor checks each packet and compresses it only if it matches a rule provided by the controller, replacing the original Layer 3/4 headers with the corresponding MPLS header. On the receiver side, the decompressor restores the original headers according to its decompression rules. Additionally, compressors monitor each flow's packet count and report flow statistics to the controller. These components can be implemented in software on the server or offloaded to hardware such as the NIC or last-hop switch, as is common for other features in modern cloud deployments [4, 49]. To minimize programmability requirements, Flow.zip employs simple, rule-based header replacement.

Controllers: Flow.zip assumes a logically centralized (but shardable) controller, similar to those commonly used in existing production networks [28, 47, 48, 50, 51]. Using the reported compressor statistics, the controller is responsible for identifying large flows that should be compressed and installing the corresponding rules on applicable switches, compressor, and decompressor. It is also responsible for actively managing label capacity and ensuring fault tolerance and consistency of the labels throughout their lifecycle.

Compressed fields (Figure 5). Flow.zip targets the lossless compression of Layer 3/4 headers, including any encapsulation headers from protocols such as VXLAN and NVGRE, though the exact set of headers and their handling are configurable by the network operator (see Section 5.1).

In general, Flow.zip will categorize fields into one of four types: (1) Routing fields (e.g., the 5-tuple) are mapped one-to-one to an

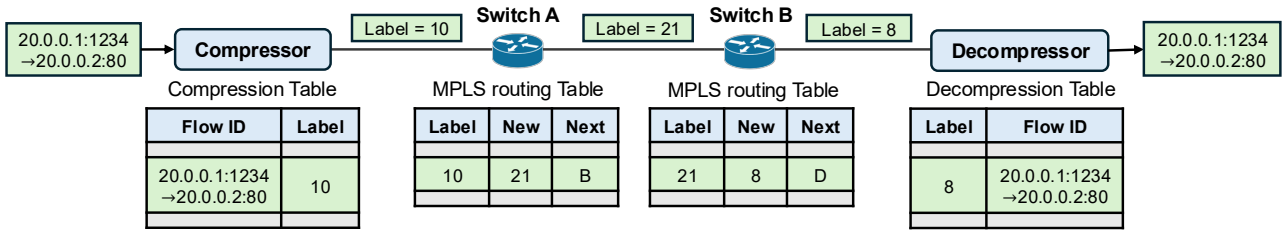


Figure 6: An example lifecycle of a packet in Flow.zip.

MPLS label. Additionally, certain IP header fields, such as QoS and TTL, are directly placed into the corresponding fields of the MPLS header. (2) Static fields, i.e., those that remain unchanged across packets in a given MPLS label, are not transmitted. Instead, they are implicitly restored during decompression using the associated MPLS label. (3) Dynamic fields, i.e., those that may vary within an MPLS label, such as payload length, are appended after the MPLS label as residual data. (4) Unused fields are ignored.

Notably, all of the above require only simple, rule-based header replacement (R3) of a handful of flows (R2) only at the compressor and decompressor (R1), minimizing the necessary programmability and ensuring compatibility with existing hardware.

Packet lifecycle (Figure 6). Figure 6 shows a simplified example of a compressed packet's lifecycle. The compressor opportunistically replaces the packet's original L3/L4 header stack with an MPLS header if and only if the label is ready for end-to-end routing and a matching compression table rule is found (Section 5.1 and Section 5.3). Transit switches route on the labels, making sure to swap on every hop to simplify label management and enable lazy deletion (Section 5.3). Finally, the decompressor reconstructs the original header using installed rules and residual data behind the MPLS header.

Scalability and correctness. Flow.zip's architecture is designed to scale to multi-terabit workloads with millions of concurrent flows (Section 8.1), which is well within the capacity of modern SDN controllers [50, 52]. In the end, however, Flow.zip is an optimization that 'fails open' (Section 5.4). If Flow.zip cannot keep up, it falls back to normal IP routing—the cost is efficiency, not correctness. To that end, Flow.zip maintains several invariants to guarantee correctness:

1. *Atomic activation/deactivation:* Compression, despite requiring the cooperation of every device on the network path, is activated/deactivated with a single operation on a single node. Packets never encounter intermediate states—their path either supports compression or not.
2. *Local label validity:* A naive design would attach a single MPLS label at the ingress and preserve it throughout the network. However, this approach requires coordinating global label usage when allocating state for new flows. Instead, labels in Flow.zip are managed on a hop-by-hop basis to improve scalability. As a result, label capacity becomes a per-device constraint rather than a network-wide limitation. With 16K labels, each device can compress up to 16K concurrent large flows, while the network as a whole can support a substantially larger number of flows.
3. *Checked dictionary compression:* To guarantee that the compression is truly lossless, the compressor matches outgoing packets with an expected header pattern. This goes beyond simply checking the packet's 5-tuple, Flow.zip verifies all routing and static fields to ensure that the decompressed packet will be identical to the original. Any discrepancies cause a fallback to IP routing.
4. *Soft state and IP fallback:* In Flow.zip, it is always safe to leave a packet uncompressed and rely on IP routing.

5 Flow.zip Header Compression

In this section, we present the detailed design of Flow.zip. We begin by describing how Flow.zip compresses each header field based on its category in the compression rules (Section 5.1) before explaining its key functions. For simplicity, we assume a pure Layer-3 data center, with extensions to other scenarios (e.g., middleboxes) left to Appendix B.

5.1 Compression Rules

The compressor compresses each packet according to a set of operator-defined rules that specify the category of each header field and how Flow.zip should handle it. While the specific rules are network-specific (e.g., whether IPv6 Flow Labels are used in routing), we present an example compression policy here, one that is also used in our evaluation.

As shown in Figure 7, Flow.zip divides header fields into four types: routing-relevant, static, dynamic, and ignored.

Routing-relevant fields: Routing-relevant fields are header fields that must be encoded, one-to-one, in the MPLS header to enable correct forwarding decisions. In most cases, this involves Flow.zip assigning a unique MPLS label to each new 5-tuple, but this can differ for other routing keys (e.g., source address, destination address, and flow label). The traffic class and TTL fields are directly transcribed between the IP and MPLS headers during compression/decompression.

Static fields: Static fields are header fields that remain unchanged across all packets associated with a given MPLS label. In our prototype, these fields include the IP version, Internet Header Length (IHL), and fragmentation-related fields. These fields are not transmitted in the compressed packet; instead, they are restored directly during decompression using the MPLS label as the key. Importantly, Flow.zip does not assume that all incoming IP packets conform to specific patterns. Instead, it performs a validation check before compression: a packet is only compressed if its static fields match

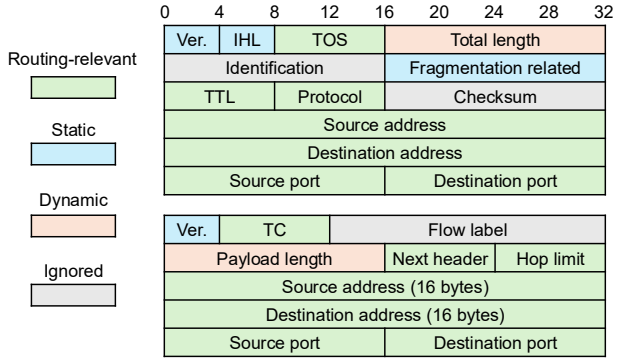


Figure 7: Compression of IPv4/IPv6 and TCP/UDP port numbers in Flow.zip.

the expected values. For simplicity, Flow.zip, by default, ignores packets that include IP options or enable fragmentation.

Dynamic fields: Dynamic fields are header fields that may vary across packets within the same MPLS label (e.g., the IPv6 payload length). These fields are placed in the compression residue, which follows the MPLS header, and are copied back to the original header during decompression.

Ignored fields: Finally, Flow.zip can safely ignore unused header fields, which in our prototype include the IP identification and flow label. At the decompressor, these fields are simply set to zero. Similarly, the checksum field is not needed because MPLS-based forwarding does not rely on per-hop checksum validation. If downstream devices still require a valid checksum, the decompressor can recompute and restore it to maintain packet validity.

Tunneling headers. Flow.zip also supports the compression of encapsulation headers introduced by tunneling protocols, following the same taxonomy described above. For example, in VXLAN, Flow.zip can compress the outer IP header, outer UDP header, VXLAN header, inner Ethernet header, inner IP header, and transport-layer port numbers. In our prototype, when using VXLAN, we extend the routing key from the standard 5-tuple to a richer tuple that includes the outer 5-tuple, the VXLAN Network Identifier (VNI), the inner Ethernet header, and the inner 5-tuple, all of which may be required to uniquely identify a flow in multi-tenant or virtualized environments. All other header fields are handled similarly to the non-tunneled case.

5.2 Packet Count Monitoring

As discussed in Section 3, Flow.zip leverages traffic skewness by assigning MPLS labels only to large, high-volume flows. To enable this, the compressor must monitor the flow-level packet counts. For IP packets, if the packet count exceeds a predefined threshold S (e.g., 100 packets in our evaluation), the compressor sends a 'new flow' notification to the controller to request rule installation. As analyzed in Theorem 2, Flow.zip is insensitive to the specific choice of this threshold. Even after compression, the compressor continues to track the flow and periodically reports statistics to the controller (e.g., every 100 ms). The controller uses these reports to prioritize MPLS label assignment for higher-volume flows, thereby maximizing overall compression efficiency, as detailed in Section 5.3.

We note that the mechanism for monitoring packet counts is orthogonal to the core of Flow.zip. If the compressor can estimate flow size in advance [35, 53], it can proactively send notifications to accelerate rule installation. Otherwise, Flow.zip benefits from standard tools and optimizations. For simplicity and backward compatibility, our prototype leverages packet sampling through sFlow [54], in line with prior controllers [55, 56]. Section 8.1 shows that the monitoring bandwidth overhead is negligible—below 0.5 Gbps even under 4 Tbps of user traffic. More complex techniques, such as sketch-based algorithms [57, 58], are also viable drop-in replacements but may require specialized hardware.

5.3 Controller Label Management

Path selection. The controller is responsible for managing all compression rules based on statistics reported by the compressors. When an uncompressed large flow is detected, Flow.zip attempts to find a target path for placement.

When possible, the controller keeps the compressed traffic on the same path as the original IP traffic to prevent reordering and isolate the impact of header compression. It can determine the path by correlating the flow with paths inferred from flow sampling in the middle of the network. As we target only the largest flows, this correlation can be done efficiently and with high probability. If strict flow stickiness is not possible, e.g., because the original path cannot be determined, other placements (including random) are possible. To that end, Flow.zip can be directly integrated with existing flow-scheduling systems like [56, 59] to route large flows along the least-loaded paths.

Rule installation. For a candidate flow and target path, P, the controller will attempt to install the MPLS path if MPLS label entries are available on all switches along P. If installation fails due to insufficient label capacity, the flow is deferred but not discarded: the compressor continues to report the flow in subsequent monitoring intervals as described in Section 5.2. Large flows, which generate more installation notifications, are prioritized and retried until resources become available.

The controller initiates rule installation by pushing MPLS forwarding rules to the switches and corresponding compression/decompression rules to the involved compressor and decompressor. To ensure consistency and correctness, the controller follows a strict ordering: while the switches and decompressor can be configured in parallel, the compressor is always the *last* component to install rules (and the *first* to remove them). This guarantees that no compressed packet is transmitted before the network is ready to handle it.

Rule deletion. While a straightforward approach is to apply the inverse process to delete rules after flows that dip under the threshold, this results in high control-plane overhead and rule flapping.

Instead, for switches and decompressors, Flow.zip *never* deletes rules; rather, old rules are implicitly overwritten only when their labels are reassigned to new flows. Doing so merges deletions and installations into a single operation, avoiding nearly half of the signaling messages of the strawman implementation and allowing temporarily inactive flows to resume compression without requiring reinstallation. Compressors also take a lazy approach to deletion, deleting rules only when the label needs to be reclaimed. In

a Flow.zip deployment with properly configured routing-relevant, static, dynamic, and ignored field classifications, leaving stale rules in the network is always safe.

Overwritten rules are chosen by the controller using a process similar to Linux's two-list algorithm for approximating LRU in page frame reclamation [60]. Briefly, flows are marked as inactive if flow counter increments remain low for a predefined TTL (e.g., 100 ms) after being added to the list, and are then lazily evicted.

5.4 Fault Tolerance

For robustness, Flow.zip adheres to classic network design principles like fate-sharing, soft-state, and the guarantee of failing open. Below, we discuss the main failure scenarios and how Flow.zip handles them safely.

Unexpected input at the compressor. Within a flow, some packets may violate the compression rule (e.g., packets with fragmentation or IP options). As described in Section 5.1, the compressor applies compression only when a packet fully matches the predefined rule and can be safely encoded. Any packet that does not satisfy the rule is forwarded in its original, uncompressed form, handled by the original routing logic.

Compressor failure. If a compressor fails and loses its local state, user traffic remains unaffected. Without a valid compression rule, the host transmits packets using standard IP headers, which is always safe. Decompression and forwarding rules in the network are harmless in the absence of compressed traffic. The compression table is soft-state and can be reconstructed after recovery.

Controller failure. If the controller fails, potential impacts can be split into three categories. Previously compressed flows will continue to be compressed until the TTLs age them out of their compression tables. Uncompressed flows will not transition to compression until the controller recovers. Finally, partially installed flows will remain uncompressed—the ordering discipline in Section 5.3 essentially makes compressor rule installation an atomic commit. All remain correct while the controller recovers and rebuilds its soft state.

Switch, link, or decompressor failures. If a switch, forwarding path, or decompressor fails, there are a few options. One option, if there are alternative paths, is to leverage existing MPLS fault-tolerance mechanisms, e.g., FRR [61], to tolerate a small number of failures at the cost of label inflation. Another is to emulate other centralized network architectures and rely on the controller to detect and tear down failed paths at the compressor. As there are only a small number of paths to monitor relative to the broader network, they can be subject to increased scrutiny. Finally, we note that end-host-driven mechanisms like PRR [62] are highly complementary to Flow.zip—small changes to static or routing-relevant fields will cause the connection to appear as a new flow.

6 Theoretical Analysis

We model Flow.zip using an M/G/1-PS queue as in Theorem 1.

Pareto flow-size model. Following prior work [63, 64], we model flow sizes (in packets) using a Pareto distribution. Let X denote the number of packets in a flow. With skewness parameter $\alpha > 1$ and minimum flow size $x_m = 1$, the flow-size distribution has complementary CDF

$$\Pr_{\text{flow}}[X \geq x] = \left(\frac{x_m}{x}\right)^{\alpha} \tag{1}$$

Based on the flow-size distribution in packets, the corresponding packet-level distribution is:

$$\Pr_{\text{pkt}}[X \geq x] = \left(\frac{x_m}{x}\right)^{\alpha-1} \tag{2}$$

where $\Pr_{\text{pkt}}[X \geq x]$ denotes the fraction of packets belonging to flows with at least x packets.

The Pareto distribution is widely used to model network traffic because it captures the empirical observation that a small fraction of flows contribute a disproportionately large share of packets [63, 64]. A smaller α indicates that more packets originate from large flows. For example, with $\alpha = 1.3$, ∼25% of packets belong to flows larger than 100 packets; with $\alpha = 1.1$, this increases to roughly 63%. Empirical datacenter traces (Figure 3) commonly exhibit $\alpha < 1.1$.

Insensitivity to packet threshold. Let τ denote the packet threshold and γ the additional delay before installed rules take effect. That is, Flow.zip requests rule installation after the first τ packets, and a flow begins being compressed after $\tau + \gamma$ packets. We assume that all subsequent packets are compressible and that Flow.zip can compress all flows whose size exceeds the threshold.

Theorem 2. *The fraction of packets Flow.zip compresses is*

$$R = \frac{1}{\alpha} \cdot \Pr_{\text{pkt}}[X \geq \tau + \gamma] \tag{3}$$

Proof. A flow of size X contributes $(X - (\tau+\gamma))_+$ compressed packets. Thus, based on the Pareto distribution,

$$\begin{aligned} R &= \frac{\mathbb{E}[(X - (\tau+\gamma))_+]}{\mathbb{E}[X]} \\ &= \frac{x_m^{\alpha}/[(\alpha-1)(\tau+\gamma)^{\alpha-1}]}{\alpha x_m/(\alpha-1)} \\ &= \frac{1}{\alpha} \cdot \Pr_{\text{pkt}}[X \geq \tau + \gamma] \end{aligned}$$

□

Interpretation. This theorem shows that the fraction of packets compressed depends primarily on the share of traffic contributed by flows larger than $\tau + \gamma$ packets. For example, when $\tau + \gamma = 200$, Flow.zip compresses 54% of all packets while operating on only 0.29% of flows for $\alpha = 1.1$. When the traffic distribution is more skewed ($\alpha = 1.05$), the compressed packet fraction increases to 73%, while still touching only 0.38% of flows. These results illustrate why Flow.zip can achieve substantial packet-level compression while acting on only a tiny fraction of total flows.

Theorem 2 also implies that the compressed packet ratio R is insensitive to the choice of the compression threshold τ, once the additional delay γ is accounted for. We assume $\gamma = 100$, corresponding to a 10 Gbps throughput, an 80 μs installation delay, and 1000-byte packets. Under these conditions, with $\alpha = 1.05$, Flow.zip compresses 72% of packets when $\tau = 200$. Reducing the threshold to $\tau = 50$ increases the compressed fraction only slightly, to 74%, a marginal gain of just 2%. This insensitivity aligns with our empirical results:

across a wide range of threshold values (50–200 packets), overall performance improvements remain nearly unchanged.

Potential benefits to FCT. We compare the FCT of Flow.zip, T_{zip}, with that of the original traffic, T_{orig}. Let ρ_{orig} be the original network load and ρ_{zip} the load under Flow.zip. Similarly, let S_{zip} denote the transmitted flow size under Flow.zip, and S_{orig} the uncompressed flow size. We normalize packet size such that a fully compressed packet (excluding header overhead) has unit size. We define β as the additional header-overhead ratio in the original traffic, such that $S_{\text{orig}} = (1+\beta)\cdot S_{\text{ideal}}$, where S_{ideal} denotes the flow size if all packets were fully compressed with zero header overhead.

Theorem 3. *For a flow with X packets, the relative FCT under Flow.zip satisfies*

$$\frac{T_{\text{zip}}}{T_{\text{orig}}} = \frac{S_{\text{zip}}}{S_{\text{orig}}} \cdot \frac{1-\rho_{\text{orig}}}{1-\rho_{\text{zip}}}, \tag{4}$$

where the flow size ratio

$$\frac{S_{\text{zip}}}{S_{\text{orig}}} = \begin{cases} 1, & \text{if } X < \tau+\gamma, \\ \dfrac{(\tau+\gamma)\beta + X}{(1+\beta)X}, & \text{otherwise.} \end{cases} \tag{5}$$

and the effective network load under Flow.zip is

$$\rho_{\text{zip}} = \left(1 - \frac{R\beta}{1+\beta}\right)\cdot \rho_{\text{orig}}, \tag{6}$$

Proof. For the M/G/1-PS queue [10, 11], the expected flow completion time scales linearly with: (1) the flow size X, and (2) the queuing factor $\frac{1}{1-\rho}$. In other words,

$$\frac{T_{\text{zip}}}{T_{\text{orig}}} = \frac{\frac{S_{\text{zip}}}{1-\rho_{\text{zip}}}}{\frac{S_{\text{orig}}}{1-\rho_{\text{orig}}}} = \frac{S_{\text{zip}}}{S_{\text{orig}}} \cdot \frac{1-\rho_{\text{orig}}}{1-\rho_{\text{zip}}},$$

To compute the flow-size ratio, note that under Flow.zip, only the first $\tau+\gamma$ packets retain the full header and thus incur the original overhead. All remaining packets use the compressed header and achieve the ideal per-packet size. Hence, if $X < \tau+\gamma$, then $S_{\text{zip}} = S_{\text{orig}}$. Otherwise,

$$\frac{S_{\text{zip}}}{S_{\text{orig}}} = \frac{(\tau+\gamma)\beta + X}{(1+\beta)X}$$

Finally, as Flow.zip removes the header overhead for a fraction R of packets, the resulting aggregate network load is therefore

$$\rho_{\text{zip}} = R\cdot\frac{\rho_{\text{orig}}}{1+\beta} + (1-R)\cdot\rho_{\text{orig}} = \left(1-\frac{R\beta}{1+\beta}\right)\cdot\rho_{\text{orig}}.$$

□

Interpretation. This theorem shows that the relative FCT of Flow.zip decomposes into two components. The first component, $(1-\rho_{\text{orig}})/(1-\rho_{\text{zip}})$, reflects system-wide queueing effects and does not depend on the individual flow size. For flows with fewer than $\tau+\gamma$ packets, this term fully determines the relative FCT. To illustrate, consider a concrete setting where Flow.zip compresses $R = 80\%$ of packets. For TCP/IPv6, the header overhead is $\beta \approx 7.4\%$. In this case, the original network load is $\rho_{\text{orig}} = 80\% \times (1+\beta) = 85.9\%$, and from Equation (6), the load under Flow.zip becomes $\rho_{\text{zip}} = 81.2\%$. This yields a relative FCT $T_{\text{zip}}/T_{\text{orig}} \approx 75\%$, consistent with our evaluation where Flow.zip achieves a 20–30% FCT reduction under an 80% offered load. When VXLAN encapsulation is used, where the header overhead is larger ($\beta \approx 16.0\%$), the same calculation gives $T_{\text{zip}}/T_{\text{orig}} \approx 41\%$. This aligns with our empirical results showing that Flow.zip reduces FCT by roughly 58% under 80% load with VXLAN encapsulation. The second component, $S_{\text{zip}}/S_{\text{orig}}$, captures per-flow size reduction and grows more beneficial as the flow becomes larger, again matching our evaluation trends.

7 Implementation

We implement a prototype of Flow.zip on a hardware testbed. The servers are equipped with Mellanox ConnectX-4 25 Gbps NICs, Intel Xeon Silver 4110 CPUs @ 2.10 GHz, and 64 GB of memory. The code will be open source upon publication.

Flow.zip controller. We implement the controller logic on the control-plane CPU with around 500 lines of C code. As described in Section 5.3, the controller receives flow statistics from the compressor and sends control commands to install rules on switches, decompressors, and compressors. These control-plane operations are asynchronous and remain off the critical path; therefore, they do not impact data-plane performance metrics such as FCT.

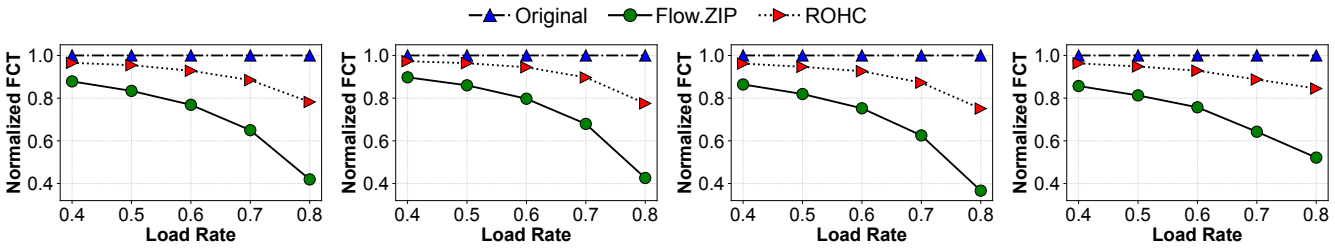
Flow.zip compressor/decompressor. We implement the Flow.zip compressor and decompressor in P4-16 to demonstrate cross-platform compatibility, as P4 is supported by both programmable NICs [65–67] and switches [21]. Due to hardware availability constraints, we deploy the compressor/decompressor to two Wedge100BF-32X ToR switches directly connected to the source/destination in a 'bump-in-the-wire' configuration. Because the compressor/decompressor logic fits entirely within the switch hardware pipeline without requiring additional packet-processing stages, the added data-plane latency is negligible. This deployment also represents a lower bound on the performance of Flow.zip, as a SmartNIC would not be bottlenecked by uncompressed throughput on the server-ToR link.

Critically, as described in Section 5, the implementation requires only three additional tables—a compression table, a decompression table, and an sFlow table for monitoring packet counts. It does not rely on any hardware-specific externs. While our prototype targets hardware, Flow.zip can also be implemented in software, for example, within the host networking stack or as an extension to a software proxy such as Envoy [68]. A software-based deployment would offer greater flexibility (e.g., alternative monitoring or control policies), at the cost of increased CPU overhead.

Hardware resource usage: On a Wedge100BF-32X [21], the resulting program takes 14.69% of SRAM and 3.85% of hash bits, primarily due to the compression and decompression tables, each containing 16K entries. It also uses 2.08% of the Meter ALU for the sFlow packet count implementation. Other compute resources (e.g., 10.94% gateway usage and 4.43% VLIW actions usage) are used to evaluate conditional logic (e.g., determining whether a packet is IPv4 or IPv6, TCP or UDP) and to copy fields accordingly.

8 Evaluation

We evaluate Flow.zip through a combination of testbed experiments and large-scale ns-3 simulations [69]. Our evaluation focuses on the following three key questions:



(a) Hadoop. (b) Storage. (c) Cache. (d) WebSearch.

Figure 8: Normalized FCT across different workloads for TCP with VXLAN. Flow.ZIP reduces FCT by up to 58% compared to the uncompressed baseline, and by up to 46% compared to the ROHC baseline.

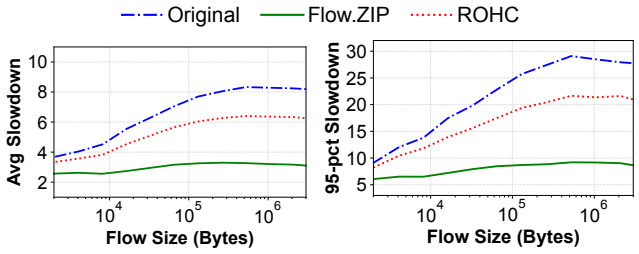
- How does Flow.ZIP impact average and tail FCTs across flow sizes?
- Is Flow.ZIP robust across diverse settings (e.g., TCP, RDMA, and tunneling) and workloads (including different traffic patterns, flow size distributions, and incast conditions)?
- What is the control-plane overhead of Flow.ZIP?

8.1 Simulation

Topology. We simulate a FatTree [33] topology consisting of 9 core switches and 6 pods. Each pod contains 3 aggregation switches and 3 top-of-rack (ToR) switches, resulting in a total of 18 aggregation switches, 18 ToR switches, and 216 servers. All links operate at 100 Gbps, leading to a 4:1 oversubscription ratio, consistent with prior work [70–72]. Each link has a propagation delay of 1 µs, leading to a maximum base round-trip time (RTT) of 12 µs. Switch buffer sizes are configured according to the bandwidth-buffer ratio of Intel Tofino switches [21], following prior work [71, 72]. Specifically, each switch is provisioned with approximately 2 MB of buffer for up to 600 Gbps of user traffic. These buffers are shared across all ports on the switch. We use DCTCP [12] for TCP traffic and DCQCN [13] for RoCEv2 traffic. For RoCEv2, Priority Flow Control (PFC) is enabled to ensure lossless transmission. Traffic is load-balanced using Equal-Cost Multi-Path [73] routing based on the 5-tuple. One of the 216 servers is reserved as the controller, and no user traffic is routed through it.

Workloads. We evaluate Flow.ZIP using four realistic workloads. The realistic workloads are derived from production data centers: the Hadoop workload [30], the Cache workload [30], the Storage workload [14], and the WebSearch workload [12]. Their overall flow size distributions are shown in Figure 3. Flows are generated following a Poisson arrival process, consistent with prior work [1, 14]. To further assess Flow.ZIP's performance, we generate incast traffic similar to prior work [14, 71, 72]. Incast traffic is prevalent in applications ranging from traditional distributed storage [14], web search [12], to modern Mixture-of-Experts models training [74]. In our incast setup, we designate the first server in the topology as the receiver. 16 senders are selected uniformly at random from different racks and simultaneously transmit data to the receiver. The incast traffic load is less than 2% of the network capacity.

Flow.ZIP configuration. The detailed compression rules used by Flow.ZIP are described in Section 5.1. We assume the compressor and decompressor are implemented in the server's NIC, with the default packet threshold for label assignment at 100. By default, we limit the number of MPLS label entries per switch to 16K, aligning with the capabilities of commodity switches [25–27]. To isolate the effect of header compression, we ensure that MPLS-compressed traffic follows the same path as the original uncompressed IP traffic.



(a) Average slowdown. (b) 95th percentile slowdown.

Figure 9: Slowdown under 80% for Hadoop.

Baselines. We compare Flow.ZIP against two baselines:

(1) Original: The original, unmodified traffic without any form of header compression.

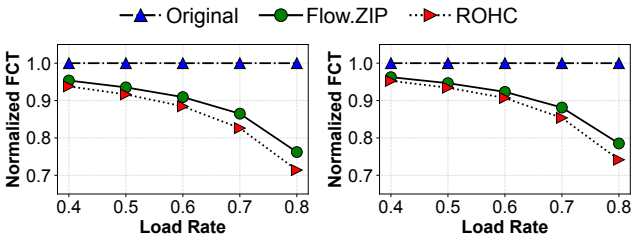
(2) Robust Header Compression (ROHC): We also compare against ROHC [18]. Per Section 2.2, ROHC is a classic solution that (*a*) performs header compression and decompression at *every* intermediate Layer-3 device, and (*b*) maintains and manipulates state for *every* flow in the network[2].

8.1.1 Flow Completion Time for TCP

This section presents the flow completion time (FCT) and slowdown for TCP/IPv6 traffic, both with and without tunneling, under various network load levels and workloads. Slowdown is defined as the ratio between a flow's FCT in the presence of background traffic and its FCT when running alone. To exclude initialization and termination effects, we measure FCT and slowdown for user traffic generated between 20% and 80% of the simulation duration. All reported FCTs are normalized to the uncompressed baseline. We begin with results under tunneling.

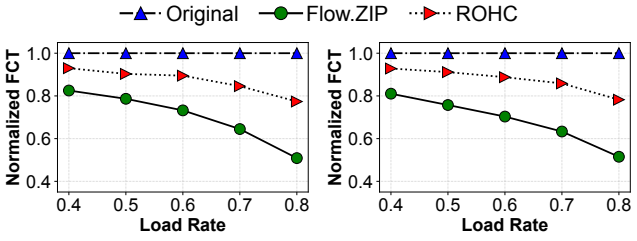
FCT under tunneling (Figure 8). As shown in the figure, Flow.ZIP significantly reduces FCT relative to the baseline, with consistent

[2]Note that recent implementations of ROHC for data centers require programmable switches in conjunction with FPGAs in a fashion that would result in half-duplex throughput for the compressor and decompressor (halving effective throughput) [16]. To maintain this as an upper bound, we assume hardware that supports full-duplex compression/decompression.



(a) Hadoop (b) Storage

Figure 10: FCT for TCP without tunneling.



(a) Hadoop. (b) Storage.

Figure 11: Normalized FCT for RoCEv2 with VXLAN.

gains across different workloads. In particular, Flow.ZIP reduces FCT by up to 58% compared to the uncompressed baseline and by up to 46% compared to the ROHC baseline. ROHC is less effective in this setting because it compresses only the outer tunnel header, leaving inner headers uncompressed, thereby limiting its benefit under tunneling.

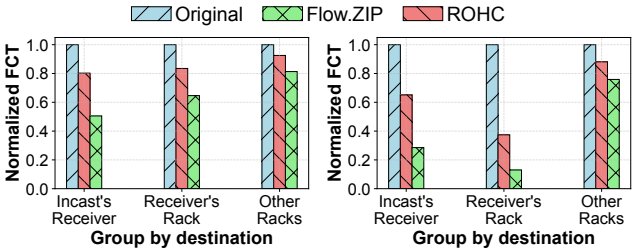
Slowdown across different flow sizes (Figure 9). We next examine flow slowdown as a function of flow size. The results show that the benefits of Flow.ZIP increase with flow size, consistent with the analysis in Section 6. For flows smaller than 10 KB, Flow.ZIP reduces average slowdown by approximately 43% and the 95th-percentile slowdown by about 53% relative to the uncompressed baseline. For flows larger than 1 MB, the reductions increase to approximately 62% and 68% for the average and 95th percentile, respectively.

FCT without tunneling (Figure 10). We also evaluate FCT in the absence of VXLAN tunneling. At a 70% network load, Flow.ZIP achieves an FCT reduction of approximately 14%, increasing to about 23% under an 80% load. Without tunneling, Flow.ZIP delivers FCT performance comparable to ROHC, typically within a 7% margin. ROHC attains slightly lower FCT in this scenario because it compresses additional TCP header fields (e.g., TCP sequence and acknowledgment numbers). However, ROHC relies on per-hop compression and decompression, which is incompatible with modern data center network architectures, as discussed in Section 2.

8.1.2 Flow Completion Time for RoCEv2

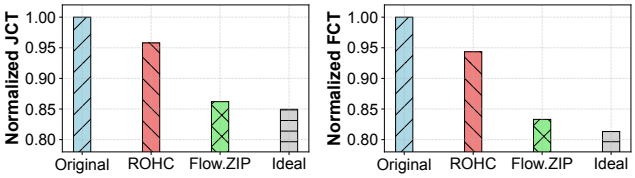
This section presents the flow completion time (FCT) for RoCEv2/IPv6 traffic. Non-tunneling results are in Appendix A.

FCT under tunneling (Figure 11). We find that the performance advantage of Flow.ZIP is consistent across RoCEv2 and TCP traffic. For tunneled RoCEv2 traffic, Flow.ZIP reduces FCT by up to 49% relative to the uncompressed baseline and by up to 34% compared to ROHC.



(a) Hadoop. (b) Storage.

Figure 12: Incast + 40% load traffic.



(a) Job Completion Time. (b) Flow Completion Time.

Figure 13: Collective Communication.

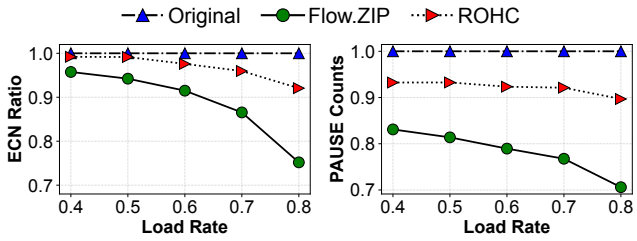
Incast traffic (Figure 12). In these experiments, we inject incast traffic on top of 40% load background traffic. We group flows by their destination and report normalized FCT ratios. For traffic destined to other racks, the normalized FCT under Flow.ZIP remains around 80%, consistent with the results observed at 40% load in Figure 11. In contrast, Flow.ZIP substantially reduces FCT for incast traffic as well as for traffic destined to servers in the same rack as the incast receiver. Specifically, Flow.ZIP reduces the FCT of incast traffic by up to 71%. These results demonstrate that even when the overall network load is moderate, Flow.ZIP effectively mitigates transient, localized congestion induced by incast traffic.

Collective communication (Figure 13). Following prior work [75], we simulate an All-to-All communication workload representative of MoE training. Specifically, within a block of 36 servers, each server sends 8MB of data to all 36 servers in another block. Figure 13a reports the job completion time (JCT), measured as the interval between the start of the first flow and the acknowledgment of the last flow. Flow.ZIP reduces JCT by approximately 14% compared to the uncompressed baseline and by 10% compared to ROHC. Moreover, the gap between Flow.ZIP and the idealized case without header overhead is less than 2%. Figure 13b shows the average flow completion time (FCT) across all flows. Compared to the uncompressed baseline, Flow.ZIP reduces average FCT by approximately 17%, and by 12% relative to ROHC.

8.1.3 Deep Dive

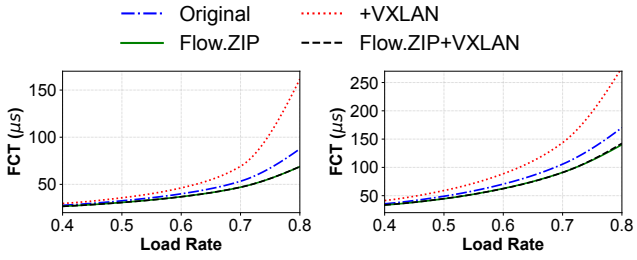
This section provides a deeper analysis of how Flow.ZIP mitigates network congestion and improves FCT. The compression rates of Flow.ZIP are deferred to Appendix A.

Congestion signals (Figure 14). Figure 14a reports the ECN marking ratio for TCP/IPv6 traffic under tunneling, defined as the fraction of ECN-marked packets among all packets and normalized to the uncompressed baseline. Compared to the baseline, Flow.ZIP reduces the ECN marking ratio by up to 25%, and by up to 18% relative to ROHC. Figure 14b shows the number of PAUSE messages



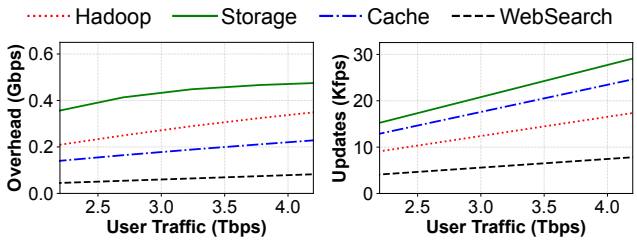
(a) ECN Marking Ratio. (b) Number of PAUSE Msg.

Figure 14: Congestion signals under Hadoop workload.



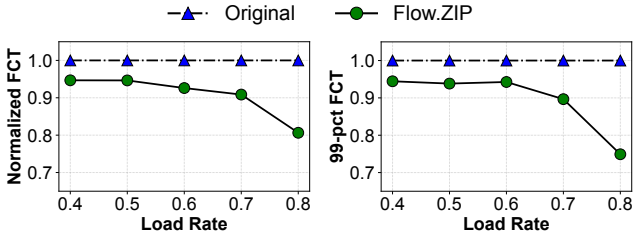
(a) TCP/DCTCP. (b) RoCEv2/DCQCN.

Figure 15: FCT across different header types.



(a) Monitoring overhead. (b) Updates overhead.

Figure 16: Overhead in control plane.



(a) Average. (b) 99th-percentile.

Figure 17: Normalized FCT under varying load rates.

for RoCEv2/IPv6 traffic under tunneling. Flow.ZIP reduces PAUSE events by up to 29% compared to the baseline, and by up to 21% relative to ROHC. Together, these results indicate that reducing header size alleviates in-network congestion, which in turn improves flow completion times.

FCT across different headers (Figure 15). Figure 15 compares the average FCT under the Storage workload across different header configurations. '*Original*' denotes traffic without tunneling, while '*+VXLAN*' denotes traffic with VXLAN tunneling. 'Flow.ZIP' and 'Flow.ZIP +VXLAN' represent the corresponding configurations with Flow.ZIP compression enabled. These results highlight the nonlinear sensitivity of FCT to network load, as discussed in Section 2.1. For example, under DCTCP with VXLAN tunneling, the average FCT increases from $\approx$ 30 μs at 40% load to $\approx$ 160 μs at 80% load. After applying Flow.ZIP, the average FCT at 80% load decreases to $\approx$ 60 μs. Notably, after applying Flow.ZIP, the FCT under tunneling becomes comparable to that of the non-tunneled case. This is because the additional tunnel-header overhead is effectively removed from packets and encoded into MPLS headers, reducing network load and mitigating congestion.

8.1.4 Control Plane Overhead

We also analyze the control-plane overhead of Flow.ZIP under diverse workloads. Figure 16a presents the monitoring overhead, defined as the traffic incurred when collecting large-flow statistics across the network. The X-axis represents the total amount of user traffic generated in the network. Even under heavy workloads with 4 Tbps of user traffic and millions of concurrent flows (e.g., the Storage workload), the monitoring overhead remains below 0.5 Gbps ($\approx$0.01%). For workloads dominated by large flows (e.g., WebSearch), the overhead is even lower—below 0.1 Gbps under 4 Tbps of traffic. Figure 16b shows the update rate of Flow.ZIP, i.e., the number of flows requiring rule installation, updates, or deletions per second. Even for 4 Tbps user traffic, Flow.ZIP processes fewer than 30 K updates per second, which can be easily handled by a single control-plane server. For comparison, modern production control planes can sustain millions of flow updates per second [50, 52].

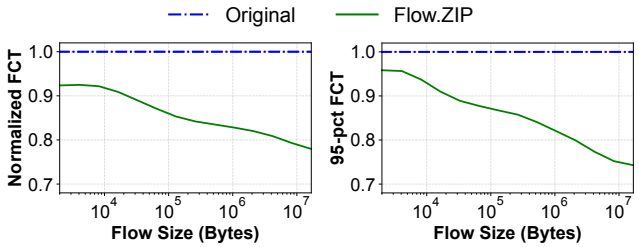
8.2 Testbed Experiments

Implementation details are provided in Section 7. We deploy Flow.ZIP on a small-scale testbed consisting of three servers. Each server generates IPv6 DCTCP flows based on the WebSearch workload distribution and sends traffic to the others, following the setup described in [53]. All three servers are connected to a first switch (i.e., the last-hop switch), where the compressor and decompressor are implemented. This switch is connected to a second switch, which is responsible for forwarding both IP and MPLS traffic.

Each packet from a sender first passes through the first switch for compression, then through the second switch for forwarding, and finally returns to the first switch for decompression before being delivered to the destination server. The links between the servers and the first switch are 25 Gbps, while the link between the two switches are 10 Gbps. Uncompressed IP serves as the baseline for these experiments due to lack of hardware support for ROHC.

FCT across different load rates (Figure 17). The FCT is normalized to the baseline (no compression). As shown in the figure, Flow.ZIP reduces the average FCT of all flows by up to 20%, and the 99th percentile FCT by up to 25%. While the improvements are smaller in the testbed compared to simulation, this is primarily due to additional software overhead, such as kernel stack latency and user-space processing, which contributes to overall FCT in the testbed environment. Nonetheless, the results demonstrate that Flow.ZIP remains effective in reducing FCT even under real-world conditions with non-negligible system overhead.

FCT across different flow sizes (Figure 18). We also measure the normalized FCT across different flow sizes under an 80% load.



(a) Average. (b) 95th-percentile.

Figure 18: Normalized FCT under varying flow sizes.

The results show that the performance benefit of Flow.ZIP increases with flow size. For flows larger than 10 MB, Flow.ZIP reduces the average FCT by up to 23% and the 95th percentile FCT by up to 26%. Even for small flows under 100 KB, Flow.ZIP still provides around 10% improvement, primarily due to reduced network load.

9 Discussion and Future Work

Limitations of Flow.ZIP. Like other entropy-based encoding methods (e.g., Huffman coding [76]), Flow.ZIP compresses headers by assigning fewer bits to more frequent patterns. Consequently, Flow.ZIP is most effective in low-entropy scenarios, where a small number of flows dominate; its benefits diminish when traffic is composed entirely of short, distinct flows. However, as discussed in Section 3, such scenarios are uncommon in practice due to the high overhead of establishing TCP/RDMA connections. Flow.ZIP also inherits constraints from the MPLS format. In particular, the MPLS traffic class field is limited to 3 bits, compared to 8 bits in IPv6, restricting the granularity of priority encoding. Flow.ZIP can partially mitigate this limitation by assigning different MPLS labels to differentiate packets with different priorities within the same flow.

Scalability of Flow.ZIP. Flow.ZIP can compress up to 16K concurrent large flows per device using 16K labels. Recent RDMA measurements report fewer than 10 synchronized bursts per rack [77], suggesting that label exhaustion is unlikely in practice. Even when the number of concurrent large flows exceeds the per-device label capacity, Flow.ZIP can still improve performance by compressing packets from the supported set of flows.

MPLS mechanism compatibility. As discussed in Section 3, although MPLS is widely deployed in WAN [28, 29, 44], it is less common in modern data-center networks. Importantly, Flow.ZIP does not require exclusive use of the MPLS label space. In deployments where MPLS labels are already used for other purposes, label-space partitioning can be employed to reserve a subset of labels for Flow.ZIP while maintaining isolation. Our evaluation shows that Flow.ZIP uses label space efficiently: reducing the per-device label capacity from 16K to 4K results in less than a 2% FCT gap. Consequently, Flow.ZIP can coexist with other MPLS-based mechanisms while consuming only a modest portion of the label space.

Extending beyond data centers. Although Flow.ZIP targets scale-out data center networks, its design can generalize to other environments. In networks without centralized control, Flow.ZIP could integrate with existing MPLS signaling protocols such as RSVP [29] and LDP [45] to manage MPLS labels in a distributed manner. We leave the exploration of these extensions to future work.

10 Related Work

Header compression. As discussed in Section 2, previous header compression techniques—such as ROHC, 6LoWPAN, CRTP, and IPHC [17–20]—were designed for low-bandwidth Layer 2 networks, such as wireless networks. These approaches require each packet to be decompressed and re-compressed at every Layer 3 device to make routing decisions, making them impractical for modern data center switches.

Several prior efforts [78–80] have explored applying traditional compression schemes (e.g., ROHC) to MPLS networks. These approaches assume traffic is already MPLS-encapsulated and that MPLS forwarding rules and label assignments remain fixed. They compress MPLS/IP headers into an MPLS/ROHC representation, but do not modify MPLS routing semantics or label allocation. In contrast, compression in modern data centers requires a fundamentally different design. Flow.ZIP targets today's data centers, which are typically not MPLS-based, and repurposes MPLS labels as both routing identifiers and compact encodings of L3/L4 header context. This design enables header compression without requiring a pre-existing MPLS deployment.

Traffic engineering. Prior work [28, 29, 44] uses MPLS for traffic engineering in wide area networks by grouping flows and assigning MPLS labels based on QoS requirements. These methods preserve the full IP/TCP headers, appending them after the MPLS label without compression. In contrast, Flow.ZIP repurposes MPLS for data center networks to enable practical, efficient header compression.

Other sources of network overhead. Beyond header overhead, several well-studied factors contribute to inefficiency in modern data center networks, e.g., congestion within the host network [81, 82] or software stack inefficiencies (e.g., data copying) [83, 84]. While these issues arise from end-host behavior or protocol-layer constraints, Flow.ZIP targets a complementary dimension: header overhead in the data plane, reducing redundant per-packet metadata to improve overall network efficiency.

11 Conclusion

Despite prior efforts, packet header compression remains impractical for modern data centers, leaving header overhead a persistent source of inefficiency. In this work, we take a backward-compatible approach for today's large-scale data center hardware. By repurposing standard MPLS and applying lightweight rule-based compression at the last hop, Flow.ZIP safely reduces header overhead and significantly improves flow completion times under real-world workloads.

Acknowledgments

We thank the anonymous reviewers for their thorough comments and feedback. This work was funded in part by NSF grant CCF-2326606 and by a Google Research Scholar award.

References

[1] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. PINT: probabilistic in-band network telemetry. In *SIGCOMM '20*, pages 662–680. ACM, 2020.

[2] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *NSDI 2021*, pages 991–1010. USENIX Association, 2021.

[3] Kalapriya Kannan and Subhasis Banerjee. Scissors: Dealing with header redundancies in data centers through SDN. In *CNSM 2012*, pages 295–301. IEEE, 2012.

[4] Daniel Firestone, Andrew Putnam, and Sambrama Mundkur et al. Azure accelerated networking: Smartnics in the public cloud. In *NSDI 2018*, pages 51–66. USENIX Association, 2018.

[5] Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. https://datatracker.ietf.org/doc/html/rfc7348.

[6] Nvgre: Network virtualization using generic routing encapsulation. https://datatracker.ietf.org/doc/html/rfc7637.

[7] Yinda Zhang, Liangcheng Yu, Gianni Antichi, Ran Ben Basat, and Vincent Liu. Enabling silent telemetry data transmission with invisiflow. In *NSDI 2025*. USENIX Association, 2025.

[8] Scale-up network header (sunh). https://datatracker.ietf.org/doc/draft-herbert-sunh/.

[9] Scale-up ethernet framework specification. https://docs.broadcom.com/doc/scale-up-ethernet-framework.

[10] Natarajan Gautam. Analysis of queues. *CRC Press, LLC, Boca Raton, Florida, United States*, 10:2222496, 2012.

[11] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI 2012*, pages 253–266. USENIX Association, 2012.

[12] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM 2010*, pages 63–74. ACM, 2010.

[13] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM 2015*, pages 523–536. ACM, 2015.

[14] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In *SIGCOMM 2019*, pages 44–58. ACM, 2019.

[15] Rémi Oudin, Gianni Antichi, Charalampos Rotsos, Andrew W. Moore, and Steve Uhlig. OFLOPS-SUME and the art of switch characterization. *IEEE J. Sel. Areas Commun.*, 36(12):2612–2620, 2018.

[16] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *NSDI 2021*, pages 571–592. USENIX Association, 2021.

[17] Ipv6 over low-power wireless personal area networks (6lowpans): Overview, assumptions, problem statement, and goals. https://datatracker.ietf.org/doc/rfc4919/.

[18] The robust header compression (rohc) framework. https://datatracker.ietf.org/doc/html/rfc4995.

[19] Compressing ip/udp/rtp headers for low-speed serial links. https://datatracker.ietf.org/doc/html/rfc2508.

[20] Ip header compression. https://datatracker.ietf.org/doc/html/rfc2507.

[21] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/.

[22] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *NSDI 2022*, pages 1345–1358. USENIX Association, 2022.

[23] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: enabling state-intensive network functions on programmable switches. In *SIGCOMM '20*, pages 90–106. ACM, 2020.

[24] Multiprotocol label switching architecture. https://datatracker.ietf.org/doc/html/rfc3031.

[25] 10 tb/s stratadnx™ jericho2 ethernet switch series. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690.

[26] Cisco catalyst 9400 series switch data sheet. https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-9400-series-switches/nb-06-cat9400-ser-data-sheet-cte-en.html.

[27] Bcm56980 12.8 tb/s multilayer switch. https://docs.broadcom.com/doc/56980-DS.

[28] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *SIGCOMM 2013*, pages 3–14. ACM, 2013.

[29] Resource reservation protocol (rsvp). https://datatracker.ietf.org/doc/html/rfc2205.

[30] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM 2015*, pages 123–137. ACM, 2015.

[31] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM 2009*, pages 51–62. ACM, 2009.

[32] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. Plb: congestion signals are simple and effective for network load balancing. In *SIGCOMM 2022*, pages 207–218, 2022.

[33] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM 2008*, pages 63–74. ACM, 2008.

[34] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *Comput. Commun. Rev.*, 36(1):59–62, 2006.

[35] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[36] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI 15*, pages 455–468. USENIX Association, 2015.

[37] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *SIGCOMM '17*, page 362–375. Association for Computing Machinery, 2017.

[38] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *SIGCOMM 2018*, pages 313–326. ACM, 2018.

[39] Feixue Han, Qing Li, Jianer Zhou, Hong Xu, and Yong Jiang. APS: adaptive packet sizing for efficient end-to-end network transmission. In *IWQoS 2022*, pages 1–10. IEEE, 2022.

[40] Broadcom first to deliver 64 ports of 100ge with tomahawk ii 6.4tbps ethernet switch. https://investors.broadcom.com/news-releases/news-release-details/broadcom-first-deliver-64-ports-100ge-tomahawk-ii-64tbps.

[41] Cisco silicon one g200 data sheet. https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/silicon-one-g200-ds.html.

[42] Vamsi Addanki, Maciej Pacut, and Stefan Schmid. Credence: Augmenting datacenter switch buffer sharing with ML predictions. In *NSDI 2024*, pages 613–634. USENIX Association, 2024.

[43] Weitao Wang, Dingming Wu, Sushovan Das, Afsaneh Rahbar, Ang Chen, and T. S. Eugene Ng. RDC: energy-efficient data center network congestion relief with topological reconfigurability at the edge. In *NSDI 2022*, pages 1267–1288. USENIX Association, 2022.

[44] Srikanth Kandula, Dina Katabi, Bruce S. Davie, and Anna Charny. Walking the tightrope: responsive yet stable traffic engineering. In *SIGCOMM 2005*, pages 253–264. ACM, 2005.

[45] Ldp specification. https://datatracker.ietf.org/doc/html/rfc5036.

[46] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, et al. Empowering azure storage with RDMA. In *NSDI 2023*, pages 49–67. USENIX Association, 2023.

[47] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *NSDI 22*, pages 325–338. USENIX Association, 2022.

[48] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, August 2013.

[49] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *SIGCOMM 2021*, pages 194–206. ACM, 2021.

[50] Andrew D. Ferguson, Steve D. Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Mühe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's software-defined networking control plane. In *NSDI 2021*, pages 83–98. USENIX Association, 2021.

[51] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM 2015*, pages 183–197. ACM, 2015.

[52] Teemu Koponen, Martín Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and

Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI 2010*, pages 351–364. USENIX Association, 2010.

[53] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM 2018*, pages 221–235. ACM, 2018.

[54] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. *RFC*, 3176:1–31, 2001.

[55] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. Openflow: enabling innovation in campus networks. *Comput. Commun. Rev.*, 38(2):69–74, 2008.

[56] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. In *SIGCOMM 2011*, pages 254–265. ACM, 2011.

[57] Yinda Zhang, Peiqing Chen, and Zaoxing Liu. Octosketch: Enabling real-time, continuous network monitoring over multiple cores. In *NSDI 2024*. USENIX Association, 2024.

[58] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *ACM SIGCOMM 2021*, pages 207–222. ACM, 2021.

[59] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI 2010*, pages 281–296. USENIX Association, 2010.

[60] Shaochang Liu and Jie Ren. Assessing page reclamation mechanisms for linux. In *SC Workshops 2025*, pages 1762–1769. ACM, 2025.

[61] A framework for ip and mpls fast reroute using not-via addresses. https://datatracker.ietf.org/doc/html/rfc6981.

[62] David Wetherall, Abdul Kabbani, Van Jacobson, Jim Winget, Yuchung Cheng, Charles B. Morrey III, Uma Moravapalle, Phillipa Gill, Steven Knight, and Amin Vahdat. Improving network availability with protective reroute. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 684–695, New York, NY, USA, 2023. Association for Computing Machinery.

[63] Vern Paxson. End-to-end internet packet dynamics. In *Proceedings of the ACM SIGCOMM 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, September 14-18, 1997, Cannes, France*, pages 139–152. ACM, 1997.

[64] Mark Crovella and Azer Bestavros. Self-similarity in world wide web traffic: Evidence and causes. In *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 160–169. ACM, 1996.

[65] Amd pensando networking. https://www.amd.com/en/products/accelerators/pensando.html.

[66] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *NSDI 2023*, pages 1469–1487. USENIX Association, 2023.

[67] Esnet smartnic hardware design. https://github.com/esnet/esnet-smartnic-hw.

[68] Cloud-native high-performance edge/middle/service proxy. https://github.com/envoyproxy/envoy.

[69] Network simulator 3. https://www.nsnam.org/.

[70] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa H. Ammar, Ellen W. Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and WAN traffic aggregates. In *SIGCOMM '20*, pages 735–749. ACM, 2020.

[71] Vamsi Addanki, Oliver Michel, and Stefan Schmid. Powertcp: Pushing the performance limits of datacenter networks. In *NSDI 2022*, pages 51–70. USENIX Association, 2022.

[72] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. ABM: active buffer management in datacenters. In *SIGCOMM '22*, pages 36–52. ACM, 2022.

[73] Analysis of an equal-cost multi-path algorithm. https://datatracker.ietf.org/doc/html/rfc2992.

[74] Zhenghang Ren, Yuxuan Li, Zilong Wang, Xinyang Huang, Wenxue Li, Kaiqiang Xu, Xudong Liao, Yijun Sun, Bowen Liu, Han Tian, Junxue Zhang, Mingfei Wang, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. Enabling efficient GPU communication over multiple nics with fuselink. In *OSDI 2025*, pages 91–108. USENIX Association, 2025.

[75] Yiran Lei, Dongjoo Lee, Liangyu Zhao, Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, Hyeonseong Choi, Liangcheng Yu, Arvind Krishnamurthy, Justine Sherry, and Eriko Nurvitadhi. FAST: an efficient scheduler for all-to-all GPU communication. In *NSDI 2026*, pages 2515–2531. USENIX Association, 2026.

[76] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[77] Soudeh Ghorbani, Yimeng Zhao, Srikanth Sundaresan, Ying Zhang, Yijing Zeng, Abhigyan Sharma, Prashanth Kannan, and Cristian Lumezanu. Congestion patterns in a large-scale RDMA datacenter. In *IMC 2025*, pages 944–951. ACM, 2025.

[78] Protocol extensions for header compression over mpls. https://datatracker.ietf.org/doc/html/rfc4901.

[79] Requirements for header compression over mpls. https://datatracker.ietf.org/doc/html/rfc4247.

[80] William A. Flanagan. Header compression across entire network without internet protocol saves bandwidth and latency. In *3rd IEEE 5G World Forum, 5GWF 2020, Bangalore, India, September 10-12, 2020*, pages 281–285. IEEE, 2020.

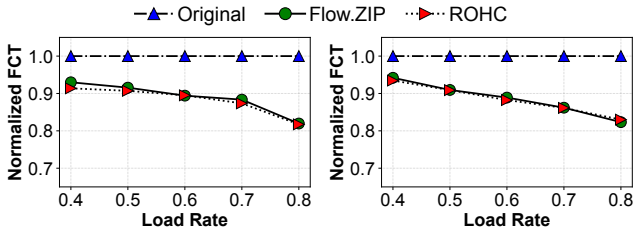
[81] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *ACM SIGCOMM 2023*, pages 275–287. ACM, 2023.

[82] Xinhao Kong, Jiaqi Lou, Wei Bai, Nam Sung Kim, and Danyang Zhuo. Towards a manageable intra-host network. In *HOTOS 2023*, pages 206–213. ACM, 2023.

[83] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: cpu-efficient remote storage access with i10. In *NSDI 2020*, pages 127–140. USENIX Association, 2020.

[84] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *SIGCOMM '22*, pages 767–779. ACM, 2022.

[85] Liangcheng Yu, Xiao Zhang, Haoran Zhang, John Sonchack, Dan R. K. Ports, and Vincent Liu. Beaver: Practical partial snapshots for distributed cloud services. In *OSDI 2024*, pages 233–249. USENIX Association, 2024.



(a) Hadoop. (b) Storage.

Figure 19: Normalized FCT for RoCEv2.

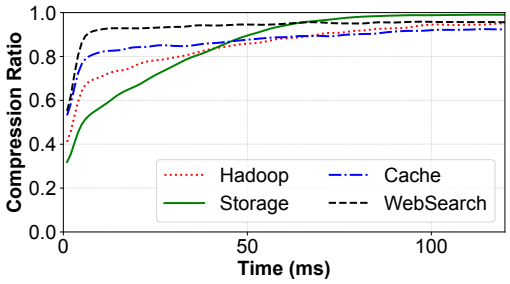


Figure 20: Compression ratio over time under 80% load.

Appendix

Appendices are supporting material that has not been peer-reviewed.

A Evaluations

FCT without tunneling (Figure 19). As shown in the figure, at a 70% load, Flow.ZIP reduces FCT by up to 14%, and at an 80% load, by up to 18%. Flow.ZIP also achieves FCT performance comparable to ROHC, typically within a 5% gap. This similarity arises because, for RoCEv2 traffic, both Flow.ZIP and ROHC compress the same header components (e.g., IPv6/UDP) while leaving the InfiniBand BTH header uncompressed.

Compression rate (Figure 20). We further examine the compression rate of Flow.ZIP over time under different workloads, which is defined as the ratio of user packets compressed in the network. The convergence speed varies across workloads. For workloads dominated by large flows (e.g., WebSearch), Flow.ZIP compresses over 90% of packets within 10 ms. Even in workloads with many small flows (e.g., Storage), it still achieves over 80% compression within 50 ms. These results demonstrate Flow.ZIP's efficiency in rapidly compressing the majority of user packets.

B Supporting middleboxes

Data-center traffic often traverses middleboxes such as load balancers, NATs, firewalls, and gateways that inspect or modify packet headers [22, 85]. To ensure compatibility with these devices, one straightforward approach is to have the middlebox decompress incoming MPLS packets, perform processing, and then recompress the packets before forwarding them.

In many cases, however, active decompression and recompression at the middlebox are unnecessary. Because compression is applied only to established flows and each flow's label mapping remains fixed throughout its lifetime, the controller can proactively install MPLS forwarding rules that preserve the intended middlebox functionality. For middleboxes that only inspect packet headers without modifying them (e.g., packet filters), if the relevant header fields are encoded in Flow.ZIP labels, the controller can directly install rules that match on the corresponding MPLS labels. For middleboxes that modify packet headers or insert additional header fields, the controller can update the corresponding decompression rules at the decompressor so that packets are correctly reconstructed without requiring decompression at the middlebox.

Note that Flow.ZIP performs header compression only within a data-center network. Packets destined for remote data centers or external networks are decompressed at the data-center gateway and restored to their original format before leaving the Flow.ZIP domain, consistent with conventional MPLS deployments. Conversely, packets entering the Flow.ZIP domain can be compressed at the ingress gateway before traversing the data-center network.