In-Kernel Aggregation and Broadcast Acceleration for Distributed Communication

Jianchang Su
University of Connecticut, Storrs, CT, USA

Yifan Zhang
University of Connecticut, Storrs, CT, USA

Wei Zhang[1]
University of Connecticut, Storrs, CT, USA

Abstract

Broadcasting and aggregation dominate the communication overhead in distributed systems, from machine learning training to data analytics. Current acceleration approaches require specialized hardware (RDMA) or dedicated resources (DPDK), limiting their deployment in commodity clouds. However, we present a counter-intuitive alternative: rather than bypassing the kernel, we move operations *into* it using eBPF. While this imposes severe constraints including no floating-point, limited memory, and stateless execution, we show these restrictions paradoxically drive innovative protocol designs that yield unexpected benefits.

We introduce AggBox, which implements broadcast and aggregation operations entirely within eBPF's constrained environment. Our key innovations include stateless group acknowledgments for reliability, edge quantization for floating-point aggregation using only integer arithmetic, and tail-call chains that create virtual memory beyond eBPF's 512-byte stack limit. These designs emerge from and exploit the constraints rather than fighting them. AggBox achieves remarkable performance on commodity hardware: 84.5% reduction in broadcast latency, 43× speedup for MapReduce workloads, and 56.1% faster ML gradient aggregation, all without specialized NICs or dedicated cores. Beyond performance, our work demonstrates that constrained environments can drive fundamental innovation in protocol design, offering insights for future resource-limited and verified systems.

2012 ACM Subject Classification Networks → Cloud computing; Networks → In-network processing; Computer systems organization → Distributed architectures

Keywords and phrases eBPF, distributed communication, broadcast, aggregation, in-kernel processing, XDP

Digital Object Identifier 10.4230/OASIcs.NINeS.2026.13

Acknowledgements We thank our shepherd Nate Foster and the anonymous reviewers for their helpful comments. We thank Chameleon Cloud [13] for the development and evaluation infrastructure. This work is supported in part by the National Science Foundation under Award #2530909.

1 Introduction

In the era of big data and distributed computing, efficient communication plays a crucial role in the performance of various systems and applications [11, 29]. Two fundamental communication patterns in distributed systems are broadcasting and aggregation [30, 26, 7, 21]. Broadcasting involves disseminating data from one node to multiple nodes, while aggregation combines data from multiple nodes into a single result. These operations are commonly used in a wide range of scenarios, such as big data processing frameworks (e.g., Apache

[1] Corresponding author

Spark [36]), distributed machine learning [31], large-scale data analytics [6], and federated learning [18, 20].

Traditional approaches to accelerating distributed communication follow two paths: hardware specialization (RDMA, SmartNICs) or complete kernel bypass (DPDK). We challenge this dichotomy with a fundamentally different approach: what if we move operations *into* the kernel rather than around it? This counter-intuitive strategy exploits an overlooked opportunity: modern kernels provide programmable hooks (eBPF) that enable safe, verified code execution at line rate.

This approach raises a profound question: can we achieve the performance benefits of kernel bypass while maintaining the deployment simplicity of standard networking? We demonstrate that the answer is yes, but requires rethinking fundamental assumptions about protocol design, reliability mechanisms, and even numerical representation. Our key insight is that the constraints of verified kernel programming, including limited instructions, no floating-point, and stateless execution, force design innovations that yield unexpected benefits: simpler protocols, predictable performance, and automatic safety guarantees.

To address these challenges, we propose AggBox, a novel system that leverages eBPF (extended Berkeley Packet Filter) and kernel hooks (XDP and TC) to accelerate distributed communication by offloading broadcasting and aggregation operations to the kernel space. eBPF is a powerful technology that allows running custom code in the kernel space without modifying the kernel itself. It provides a safe and efficient way to extend the functionality of the kernel and optimize various aspects of the system. XDP (eXpress Data Path) and TC (Traffic Control) are two key components of the eBPF ecosystem that enable high-performance packet processing and traffic control.

By leveraging eBPF, XDP, and TC, AggBox can offload broadcasting and aggregation operations to the kernel space, reducing the overhead of user-kernel crossings and network stack traversals. This approach allows AggBox to perform these operations in a more efficient way, bypassing the conventional network stack and significantly reducing the communication overhead. As a result, AggBox aims to improve the performance of distributed systems and applications by providing a more streamlined and optimized communication mechanism.

The main contributions of this paper are as follows:

- We identify a new research direction: accelerating distributed communication through *constrained* kernel programming rather than unconstrained bypass, challenging the assumption that performance requires escaping kernel restrictions.
- We present novel protocol designs that turn eBPF's limitations into advantages: stateless reliability through group ACKs, fixed-layout headers with semantic overloading, and edge quantization for integer-only aggregation. These techniques are applicable to any verified or resource-constrained environment.
- We demonstrate that commodity deployments can achieve substantial performance gains (84.5% broadcast latency reduction, 43× MapReduce speedup) without specialized hardware, making high-performance communication accessible to cloud-native applications.

2 Background and Motivation

2.1 Distributed Communication Patterns

Broadcasting and aggregation are two fundamental communication patterns in distributed systems [10]. Broadcasting is used to disseminate data from one node to all other nodes in the system. It is commonly used in scenarios such as distributing configuration updates, synchronizing global parameters, and broadcasting intermediate results. On the other hand, aggregation is used to combine data from multiple nodes into a single result. It is often used in operations like sum, average, max, and min, which are essential for data analysis, machine learning, and federated learning tasks.

These communication patterns are prevalent in various distributed systems and applications. For example, Apache Spark [36], a popular big data processing framework, uses broadcasting for distributing read-only data and configuration, and it uses aggregation for implementing reduction operations and collecting results from worker nodes. In distributed machine learning, frameworks like TensorFlow [1] and PyTorch [22] rely on broadcasting and aggregation for synchronizing model parameters and gradients across multiple nodes, including emerging workloads such as large language model training [15, 16] and federated reinforcement learning [14]. Federated learning [3], a distributed machine learning paradigm that trains models on decentralized data, heavily relies on aggregation to combine local model updates from participating clients.

The performance of distributed communication is often hindered by inefficiencies in the traditional socket-based communication stack. When a user-space application sends or receives data, it must make system calls and cross the user-kernel boundary, involving context switches and data copying between user and kernel space. Additionally, the data traverses multiple layers of the network stack, including the socket layer, TCP/IP layer, and the device driver, before reaching the network interface card (NIC). These user-kernel crossings and network stack traversals introduce significant latency, particularly for small and frequent data transfers. As the number of nodes and data size increase, this overhead becomes more pronounced, severely impacting the performance of distributed systems and applications [40].

2.2 eBPF and Kernel Hooks

eBPF (extended Berkeley Packet Filter) is a powerful technology allows running custom code in the kernel space without modifying the kernel itself [32]. It provides a safe and efficient way to extend the functionality of the kernel and optimize various aspects of the system. eBPF programs are written in a restricted C-like language and are verified for safety before being loaded into the kernel. They can be attached to various kernel hooks and events, such as network packet processing, system calls, and function entry/exit points.

XDP (eXpress Data Path) is an eBPF-based mechanism for high-performance packet processing [32]. It allows eBPF programs to be executed at the earliest point in the network stack, directly upon packet arrival from the NIC. By processing packets at such an early stage, XDP enables low-latency and high-throughput packet processing, making it suitable for use cases like packet filtering, load balancing, and network monitoring.

TC (Traffic Control) is another key component of the eBPF ecosystem that enables traffic control and packet manipulation [35]. It allows attaching eBPF programs to the traffic control subsystem of the kernel, enabling advanced packet classification, shaping, and scheduling. TC eBPF programs can inspect and modify packets based on various criteria, such as packet headers, protocol fields, and custom metadata. By leveraging eBPF, XDP, and TC, it is possible to offload certain networking operations to the kernel space, bypassing the conventional network stack and reducing the overhead of user-kernel crossings and network stack traversals. This makes eBPF a promising technology for optimizing distributed communication and improving the performance of various distributed systems and applications.

2.3 Motivation: Why Aggregation and Broadcast Matter

Recent studies show that communication accounts for up to 97% of distributed ML training time at 10 Gbps [31], with gradient aggregation being the primary bottleneck. In MapReduce workloads, shuffle operations (broadcast + aggregation) consume 30-50% of job completion time [36]. These patterns are fundamental: every distributed ML framework implements allreduce (aggregation + broadcast), and every data analytics system requires shuffle operations.

To understand where this overhead originates, we instrumented the entire packet path. We measured five components: (1) user-to-kernel transition at sender, (2) kernel protocol stack at sender, (3) network propagation, (4) kernel receive processing, and (5) kernel-to-user delivery.

Figure 1 reveals a striking asymmetry: sender-side processing takes 16.89 µs (12.3 µs user-kernel, 4.59 µs stack), while receiver-side takes 65.60 µs. This 4× difference stems from receive-side complexity: interrupt handling, packet validation, buffer allocation, and memory copying. In broadcast, this receiver overhead is paid by every node; in aggregation, it accumulates for every incoming packet.

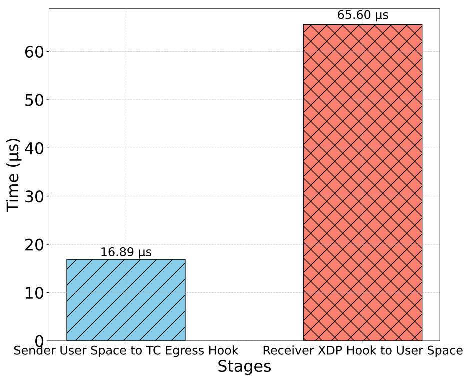


Figure 1 Detailed breakdown of packet processing latency

The micro-benchmark results demonstrate that over 80% of the end-to-end communication latency is attributed to user-kernel crossings and network stack traversals. In broadcasting scenarios, the sender-side overhead affects all receivers, while in aggregation, the receiver-side overhead accumulates for each incoming packet. These findings motivate the need for a more efficient approach that can bypass the conventional network stack while remaining practical for commodity deployments.

Table 1 compares various communication acceleration approaches. While specialized solutions like RDMA achieve the lowest latency, they require specific hardware and are difficult to deploy in cloud environments. DPDK offers good performance but requires dedicated CPU cores, making it unsuitable for shared infrastructure. AggBox addresses these limitations by providing significant performance improvements over standard TCP/IP while maintaining the ease of deployment and resource sharing capabilities essential for commodity cloud and edge deployments.

Table 1 Comparison of Communication Acceleration Approaches

Approach	Hardware	CPU	Latency	Deployment
TCP/IP	Standard	Shared	High	Easy
RDMA	Special NIC	Bypass	Lowest	Difficult
DPDK	Standard	Dedicated	Low	Moderate
AggBox	Standard	Shared	Medium	Easy

2.4 Fundamental Challenges: Rethinking Communication for Constrained Kernel Environments

Our work identifies a fundamentally new problem: how to implement high-performance collective operations within the severe constraints of verified kernel programming. Unlike traditional kernel bypass (RDMA/DPDK) that sidesteps the kernel entirely, we explore an orthogonal approach: moving critical operations *into* a restricted kernel environment. This raises unique research challenges:

The Instruction-Memory Trade-off. eBPF enforces a fundamental tension: programs are limited to 1M instructions with 512-byte stacks, yet must process unbounded data streams. Traditional algorithms fail here. We introduce a novel decomposition strategy using tail-call chains that partition aggregation into fixed-size chunks, effectively creating a "virtual stack" across program boundaries. This represents a new programming paradigm for resource-constrained environments.

Stateless Reliability. TCP's reliability assumes persistent connection state, which is impossible in eBPF's stateless execution model. We present a counter-intuitive insight: group-based acknowledgments with receiver-side state reconstruction can achieve reliability without sender-side connection tracking. Our protocol leverages the fact that datacenter packet loss is bimodal (either very low or complete failure), making group ACKs surprisingly effective.

Co-designing Protocols with Verification Constraints. eBPF's verifier rejects programs it cannot prove safe, forcing us to rethink protocol design. We discovered that traditional packet formats with variable-length fields cause verification failures. Our solution uses fixed-layout headers with semantic overloading, where the same bytes serve different purposes based on operation type, enabling complex protocols within verification bounds.

Floating-Point in Integer-Only Kernels. Aggregating ML gradients requires floating-point math, forbidden in eBPF. Rather than falling back to user space, we demonstrate that quantization at the network edge with careful precision management can maintain ML convergence while enabling pure-integer kernel processing. This challenges the assumption that ML requires floating-point communication.

These constraints force us to fundamentally rethink distributed communication design, yielding insights applicable beyond eBPF to any resource-constrained or verified execution environment.

3 Design

AggBox is designed to optimize broadcasting and aggregation operations in distributed systems by leveraging eBPF and kernel hooks. The main goals of AggBox are to:

G1 Reduce communication overhead by minimizing user-kernel crossings and network stack traversals.
G2 Provide a scalable solution that can handle moderate-scale clusters common in cloud deployments.
G3 Ensure compatibility and reliability with existing network infrastructures and protocols.

AggBox optimizes communication in distributed systems by leveraging the differences between two main eBPF hook points: TC and XDP.

For (**G1**), AggBox uses TC hooks to clone and modify packet buffers for broadcasting, reducing user-kernel crossings and improving performance by minimizing latency and resource consumption. For (**G2**), AggBox utilizes XDP hooks for data aggregation. Positioned immediately after the NIC driver receives the packet, XDP hooks enable low-latency aggregation in kernel space, reducing user-space intervention and processing overhead. To achieve (**G3**), AggBox implements a retransmission mechanism to handle packet loss, ensuring reliable communication without modifying existing network infrastructure.

3.1 Overview

AggBox consists of in-kernel components for broadcasting and aggregation, and a user-space runtime library that provides APIs to interact with these components. The overall architecture of AggBox is shown in Figure 2.

The user-space runtime library bridges user-space applications and in-kernel components, exposing the APIs listed in Table 2.

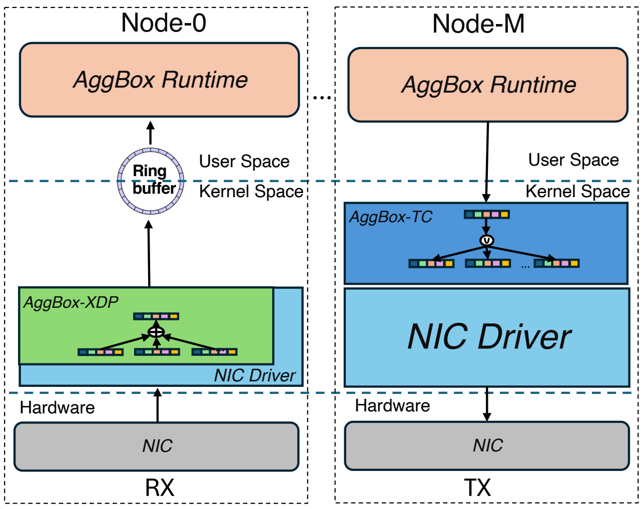


Figure 2 System architecture of AggBox

These APIs provide a convenient and efficient way for user-space applications to interact with the in-kernel broadcasting and aggregation components of AggBox.

3.2 Protocol Design

In this section, we first present the design of the communication protocols used in AggBox for broadcasting and aggregation operations. The protocol design aims to provide an efficient and lightweight mechanism for data transmission while leveraging the capabilities of eBPF and kernel hooks.

Function	Description
`append_machine_config(ip, port)`	Adds a new machine configuration to the eBPF map, specifying the IP address and port of the destination machine.
`broadcast(data, ip_list)`	Broadcasts the given data to the specified list of IP addresses, converting the IP list into an index mask for the in-kernel broadcasting component.
`send_aggregated_data(data)`	Sends data to be aggregated to the in-kernel aggregation component, formatted according to the aggregation protocol in § 3.2.
`get_aggregated_data()`	Retrieves the aggregated data from the in-kernel aggregation component, using the eBPF ring buff for efficient transfer from kernel to user space.

Table 2 User-space APIs for Broadcasting and Aggregation

Broadcast Protocol. The broadcast protocol in AggBox is designed to efficiently disseminate data from one node to multiple nodes in the system. To achieve this, we propose a custom UDP-based protocol optimized for fast and scalable broadcasting.

The broadcast packet format builds upon the standard UDP packet structure with additional fields to facilitate efficient broadcasting. The packet format is as follows:

```
[UDP Header][MAGIC_NUMBER][MASK] [SEQ_NUM][PAYLOAD]
```

The `MAGIC_NUMBER` is a 4-byte predefined value used by the eBPF program attached to the TC hook to quickly identify and process broadcast packets without deep packet inspection. The `MASK` is a 4-byte bitmask representing the set of destination nodes to which the packet should be broadcasted. The eBPF program performs a bitwise AND operation between this mask and the node index to determine the target nodes for broadcasting. The `SEQ_NUM` field is a 4-byte sequence number used for retransmission and acknowledgment purposes. It enables reliable broadcasting by allowing the sender to track the status of each packet and retransmit lost packets if necessary. The `PAYLOAD` contains the actual data to be broadcasted, and it is structured to fully utilize the maximum transmission unit (MTU) to optimize network utilization and avoid fragmentation.

Aggregation Protocol. The aggregation protocol in AggBox is designed to efficiently combine data from multiple nodes into a single result. The aggregation packet format is defined as follows:

```
[UDP Header][MAGIC_NUMBER][KEY][SEQ_NUM] [FRAG_OFFSET] [AGGR_OP]
[LAST_FRAGMENT] [PAYLOAD]
```

The `MAGIC_NUMBER` is used to identify aggregation packets. The `KEY` field is a unique identifier for the sender, allowing the aggregation program to differentiate between data from different sources, This is important for federated learning aggregation tasks where each client is assigned a different weight. The `SEQ_NUM` field is a sequence number used for retransmission and acknowledgment purposes, ensuring reliable delivery of aggregation packets.

The `FRAG_OFFSET` field indicates the starting position of the packet data within the complete data stream in the aggregated result. This allows AggBox to handle large data streams that need to be fragmented into multiple packets. The `LAST_FRAGMENT` field is a flag that indicates whether the current packet is the last fragment of the data stream.

The `AGGR_OP` field specifies the aggregation operation to be performed on the data. AggBox defines an enumeration type for the supported aggregation operations, which currently include:

- `SUM`: Computes the sum of the values.
- `COUNT`: Counts the number of values.
- `MAXIMUM`: Finds the maximum value.
- `MINIMUM`: Finds the minimum value.
- `AVERAGE`: Calculates the average of the values.

The `FRAG_OFFSET` and `LAST_FRAGMENT` fields enable the handling of fragmented data streams, ensuring that the complete data is aggregated correctly. The runtime library's `send_aggregated_data()` function formats the data according to this protocol before sending it to the in-kernel aggregation component. The eBPF program attached to the XDP hook processes the aggregation packets, extracts the relevant fields, and performs the specified aggregation operation on the data.

3.3 In-kernel Broadcasting

The in-kernel broadcasting component leverages the power of eBPF and TC hooks to efficiently disseminate data packets to multiple destinations. It utilizes the custom broadcasting protocol described in § 3.2 to facilitate efficient packet processing and forwarding.

When the `broadcast()` function is called in user space, the runtime library converts the IP list into an index mask and passes it with the data to the in-kernel broadcasting component. The eBPF program attached to the TC hook processes the broadcast packets, extracts the index mask, and determines the destination nodes based on the mask. It checks the `MAGIC_NUMBER` field to identify broadcast packets. If identified, the program extracts the `INDEX_MASK` field to determine the destination nodes.

Next, the program retrieves the next-hop information (IP and MAC addresses) from an eBPF map corresponding to the destination IP addresses obtained by the `bpf_fib_lookup` helper function. This function performs a lookup in the kernel's FIB (Forwarding Information Base) tables to determine the next-hop details for each destination IP address.

Next-hop Handling. To handle the next-hop information for broadcasting, AggBox utilizes the `bpf_fib_lookup` API to retrieve the MAC addresses corresponding to the destination IP addresses. The system employs an eBPF map to store the mapping between IP addresses and their associated MAC addresses. To ensure that the MAC addresses are up to date, AggBox leverages the `bpf_timer` facility to periodically refresh the MAC information. A typical refresh interval is set to 1 hour, but this can be adjusted based on the specific requirements of the deployment environment.

The eBPF program responsible for broadcasting uses the `bpf_fib_lookup` API to retrieve the next-hop MAC address for each destination IP. It then updates the packet headers with the appropriate MAC addresses before cloning and forwarding the packets using the `bpf_clone_redirect()` helper function. By utilizing the `bpf_fib_lookup` API and the `bpf_timer` facility, AggBox eliminates the need for user-space intervention and communication via Netlink sockets to update the MAC addresses. This approach streamlines the next-hop handling process and reduces the overhead associated with maintaining up-to-date MAC information in the eBPF map.

Retransmission Mechanism. AggBox implements a user-space retransmission mechanism to handle packet loss during broadcasting operations. The broadcasting protocol includes a sequence number field (`SEQ_NUM`) in the packet header, as described in § 3.2. This field tracks the order of packets and detects missing ones.

On the sender side, the runtime library maintains a map of sequence numbers and corresponding broadcast packets. When a packet is sent, the sender starts a timer and waits for an acknowledgment (ACK) from the receivers. The ACK contains the sequence number of the received packet, indicating successful delivery. If the sender does not receive an ACK within a specified timeout, it assumes the packet is lost and retransmits it.

On the receiver side, the runtime library tracks received sequence numbers and sends an ACK back to the sender for each received packet. The receiver maintains a buffer for out-of-order packets. If a gap in sequence numbers is detected, indicating missing packets, the receiver waits for them to arrive. If the missing packets do not arrive in time, the receiver informs the sender of all successfully received packets, even if they arrived out of order. This enables the sender to retransmit only the missing packets, improving retransmission efficiency.

AggBox also provides a group-based acknowledgment mechanism to reduce the overhead associated with acknowledging every single packet. The packets sent by each sender are logically divided into groups of a configurable size (e.g., 1024 packets). The receiver sends an ACK for each group of packets, rather than individual packets. This approach significantly reduces the number of ACKs sent while still ensuring reliable transmission.

Our group-based retransmission appears suboptimal, resembling Go-Back-N rather than selective repeat. This is deliberate and reveals a key insight: protocol efficiency depends on environment assumptions. eBPF's memory constraints (512KB max map size) make per-packet state impossible for large transfers. Group tracking (1024 packets/group) reduces memory by 1024×.

Crucially, this design exploits datacenter network characteristics where loss is bimodal, either near-zero (<0.01%) or total failure. With p=0.0001 loss rate, the probability of multiple losses in a 1024-packet group is $1-(1-p)^{1024} \approx 0.097$, and having exactly one loss is 0.088. Thus, 90% of groups need no retransmission, and most others need just one packet. This makes our "inefficient" protocol surprisingly effective in practice.

To further optimize the retransmission process, AggBox incorporates rate negotiation and aggregation node selection mechanisms. By determining the sending rate of senders and the aggregation topology based on available bandwidth and aggregator capacity, AggBox proactively minimizes the likelihood of packet loss due to network congestion or aggregator overload. This helps reduce the need for retransmissions and enhances the efficiency of the reliable transmission mechanism.

Interoperability and Fallback Behavior. While AggBox employs a custom protocol for optimized performance, it is designed with interoperability in mind. The protocol is built on standard UDP, ensuring compatibility with existing network infrastructure including switches, routers, and firewalls that handle UDP traffic. The custom header fields (MAGIC_NUMBER, MASK, SEQ_NUM) are encapsulated within the UDP payload, making AggBox packets indistinguishable from regular UDP traffic at the network layer.

For mixed deployments where some nodes run AggBox and others do not, the system provides graceful fallback behavior. Non-AggBox endpoints can communicate using standard socket APIs since the underlying transport is UDP. AggBox receivers process packets without the magic number as regular application data, forwarding them to user space through the standard network stack.

The group-based acknowledgment mechanism is a performance optimization rather than a protocol requirement. When communicating with legacy systems or in scenarios where per-packet reliability is preferred, AggBox can operate in per-packet ACK mode at the cost of increased overhead. This flexibility allows incremental deployment, enabling organizations to adopt AggBox on a subset of nodes and expand coverage over time without disrupting existing applications.

3.4 In-kernel Aggregation

The in-kernel aggregation component utilizes eBPF and XDP hooks to perform data aggregation directly in the kernel space. It leverages ring buffers (`BPF_MAP_TYPE_RINGBUF`) to efficiently transfer aggregated data between the kernel and sender user space. When the `send_aggregated_data()` function is called in the user space, the runtime library formats the data according to the aggregation protocol described in § 3.2 and sends it to the in-kernel aggregation component.

The eBPF program attached to the XDP hook processes the aggregation packets, extracts the relevant fields (`KEY`, `AGGR_OP`, `PAYLOAD`), and performs specified aggregation operation on data. The aggregated results are stored in eBPF maps, efficiently accessed by the user space using `get_aggregated_data()` function. Similar to broadcasting, AggBox also implements a retransmission mechanism for aggregation to handle packet loss, as described in § 3.3.

Handling Floating-Point Operations. eBPF prohibits floating-point arithmetic, yet ML workloads require gradient aggregation. Rather than falling back to user space (defeating our purpose), we exploit an insight from quantized training research: neural networks are surprisingly robust to reduced precision [26, 17, 37].

Our approach performs quantization at the network edge: senders multiply floating-point values by a scaling factor (e.g., 10^8) and round to 32-bit integers. The kernel performs pure integer aggregation. Receivers divide by the factor and worker count to recover floats. This is not merely a workaround but actually reduces network bandwidth (32 vs 64 bits) and leverages the fact that gradient magnitudes in modern optimizers (Adam, RMSprop) typically range from 10^{-4} to 10^{-2}, fitting comfortably in scaled integer representation.

For example, with 10,000 workers and gradient value 0.001: quantized value = $0.001 \times 10^8 = 100,000$. After aggregation: $100,000 \times 10,000 = 1,000,000,000$ (well within int32). Recovery: $1,000,000,000/10^8/10,000 = 0.001$. The quantization error ($< 10^{-8}$) is negligible compared to the inherent noise in stochastic gradient descent, which actually aids convergence through implicit regularization.

User-space Backup. In cases where the aggregation logic is too complex to be handled entirely in the kernel, even with function chaining, AggBox provides a user-space backup mechanism. This mechanism also helps to avoid wasting DMA resources on transferring packets that cannot be fully processed in the kernel.

AggBox predefines a set of aggregation operators based on the protocol design. If an incoming packet does not match any of these predefined operators, the XDP program will redirect the packet to the user-space runtime by AF_XDP for further processing. For example, if the aggregation logic involves complex operations like matrix multiplication or non-linear transformations, the XDP program will detect this and hand over the processing to the user space. Additionally, if the XDP runtime detects that the current network conditions or the data to be aggregated persist for an extended period, exceeding a predefined threshold, it will transfer the processing to the user-space runtime. This ensures that the system can handle various situations and prevent the XDP program from being overloaded or stalled. In such cases, the packet transfer utilizes AF_XDP sockets, benefiting from kernel bypass to enhance performance.

AggBox can detect DMA resource utilization by monitoring the status and length of DMA queues, handling interrupts, utilizing performance counters and metrics, managing buffer occupancy, and using eBPF programs to gather real-time data. The user-space backup mechanism processes these packets and performs the aggregation operation using the full capabilities of the user-space environment. By selectively offloading complex aggregation operations to the user space, AggBox ensures optimal utilization of DMA resources and maintains compatibility with a wide range of aggregation scenarios.

3.5 Scalability Considerations

While our experiments demonstrate performance up to 30 senders, AggBox is designed to scale beyond our testbed limitations:

Hierarchical Aggregation. For large-scale deployments, AggBox supports multi-level aggregation trees where intermediate nodes perform partial aggregation, reducing load on the root aggregator. This approach can scale to thousands of nodes by limiting fan-in at each level.

Load Distribution. The INDEX_MASK mechanism supports up to 32 destinations per packet. For larger broadcasts, we employ a tree-based dissemination where each node forwards to a subset of children, enabling broadcasts to arbitrary numbers of nodes.

Memory Management. To handle high packet rates from many senders, we implement adaptive buffer management that dynamically adjusts ring buffer sizes based on arrival rates, preventing buffer overflow under load.

Theoretical analysis shows that with 10 Gbps links and 1KB packets, a single aggregator can handle up to 1.25M packets/second. With our measured per-packet processing time, this supports approximately 300 concurrent senders before queuing delays become significant. For larger deployments, hierarchical aggregation becomes necessary.

4 Implementation

We prototyped AggBox with about 7K LOCs in C++, including two eBPF modules totaling about 500 LOCs. AggBox is built on top of BOAD [27] with additional implementation of aggregation features. Our implementation targets Linux kernel 5.15+ with eBPF support, tested on Ubuntu 22.04 LTS. We use libbpf for eBPF program loading and management, with CO-RE (Compile Once - Run Everywhere) for kernel version compatibility. The sender uses the Packet Capture (Pcap) [23] library to craft and inject packets.

Our baseline implementation uses standard UDP sockets with the same protocol headers and retransmission logic as AggBox, but processes all packets in user space. This ensures fair comparison by isolating the impact of kernel offloading. The baseline is optimized with SO_REUSEPORT for multi-core scaling, sendmmsg/recvmmsg for batch processing, and CPU affinity for NUMA-aware operation.

In the broadcasting module, we use the `bpf_clone_redirect` helper function to efficiently clone and forward packets to multiple destinations, minimizing overhead. The aggregation module, however, faces eBPF's 512-byte stack size limit. To handle large packets, we employ `bpf_tail_call` and split processing across multiple functions, each handling up to 256 bytes, recursively processing the packet pieces by `bpf_tail_call`.

Our user-space runtime library interacts with the eBPF modules. It initializes and configures the eBPF modules, constructs packet headers and payloads, interacts with eBPF maps for data exchange, and provides a high-level API for broadcasting and aggregation operations. The source code of AggBox is available at `https://github.com/cloud-sys-lab/AggBox`.

5 Evaluation

In this section, we present an extensive evaluation of AggBox, demonstrating its performance gains in terms of reduced latency for broadcasting operations and improved throughput for aggregation operations in distributed systems. We also evaluate AggBox's performance in real-world applications, including Parameter Servers and MapReduce. Our evaluation aims to answer the following questions:

1. How does AggBox broadcast perform compared to the baseline implementation in terms of broadcast latency with varying number of receivers? (§ 5.1)
2. What is the impact of packet size on the broadcast latency of AggBox and the baseline implementation? (§ 5.1)
3. How does AggBox perform in terms of aggregation throughput with varying data sizes and computation costs? (§ 5.2)
4. How does AggBox perform in real-world applications with different computational characteristics? (§ 5.1, § 5.2 and § 5.2)

Testbed Configuration. Our testbed uses Chameleon Cloud [13] with identical nodes: dual Intel Xeon Gold 6242 CPUs (32 cores, 2.8GHz), 192GB RAM, and Mellanox ConnectX-5 10 Gbps NICs. All nodes run Ubuntu 22.04 LTS with kernel 5.15.0-75, with hyperthreading disabled and CPU frequency scaling set to performance mode. Network configuration uses jumbo frames (9000 bytes MTU) and interrupt coalescing disabled for accurate latency measurements. For broadcast experiments, we use one sender and multiple receivers; for aggregation, multiple senders and one receiver.

Baseline Implementation. Our baseline uses optimized UDP socket programming with: (1) SO_REUSEPORT for multi-core scaling, (2) sendmmsg/recvmmsg for batch processing, (3) memory-mapped packet sockets where applicable, (4) CPU affinity and NUMA-aware memory allocation. This represents a well-tuned traditional approach, not a naive implementation. The baseline implements identical protocols and retransmission logic as AggBox, differing only in processing location (user vs kernel space).

Measurement Methodology. For broadcast experiments, we measure the latency as the time difference between the first packet sent by the sender and the last packet received by the receivers in the cluster. For aggregation experiments, we measure the latency as the time difference between the first packet sent by any of the senders and the moment the receiver obtains the final aggregated result. This methodology allows us to accurately assess the performance of AggBox in terms of reducing communication latency.

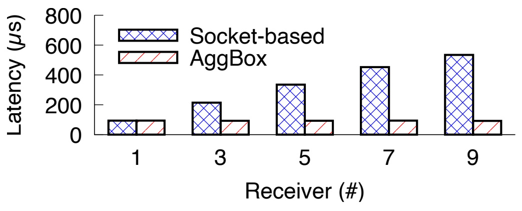
In our aggregation evaluation, we employ bursty sending of data, where the senders transmit packets in short bursts at a high rate. This bursty transmission pattern is common in distributed systems and can quickly saturate the network bandwidth, making it a challenging scenario for efficient communication.

5.1 Broadcast Experiments

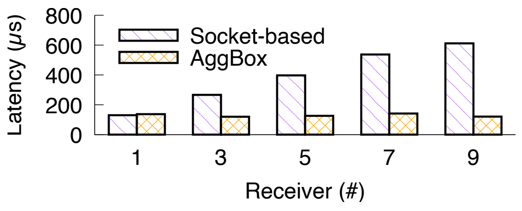
Impact of Receiver Number. We evaluate the broadcast latency of AggBox compared to a baseline implementation using traditional UDP sockets. The packet size is set to the Maximum Transmission Unit (MTU) to ensure optimal network utilization.

Figure 3(a) presents the average broadcast latency for both AggBox and the baseline implementation with a varying number of receivers. The results demonstrate that AggBox consistently achieves lower latency compared to the baseline, regardless of the number of receivers. The performance gain becomes more pronounced as the number of receivers increases, indicating the scalability of our in-kernel broadcasting approach. For instance, with 9 receivers, AggBox achieves an average broadcast latency of 92.05 µs, while the baseline implementation exhibits a latency of 534.22 µs, resulting in a significant reduction of 82.8%.

To further illustrate the robustness of AggBox, Figure 3(b) shows the 99th percentile broadcast latency. Even under high percentile latency, AggBox maintains its performance advantage over the baseline implementation, ensuring a consistently low latency for broadcasting operations. With 9 receivers, AggBox achieves a 99th percentile latency of 120.72 µs, while the baseline implementation exhibits a latency of 611.71 µs, resulting in a reduction of 80.3%.



(a) Average broadcast latency



(b) 99th percentile broadcast latency

Figure 3 Comparison of broadcast latencies

Impact of Packet Size. Now, we investigate the impact of packet size on the broadcast latency. We fix the number of receivers to 9 and vary the packet size from 64 bytes to 1024 bytes. Figure 4 presents the results, comparing AggBox with the baseline implementation.

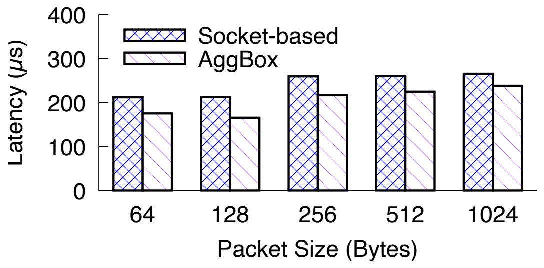
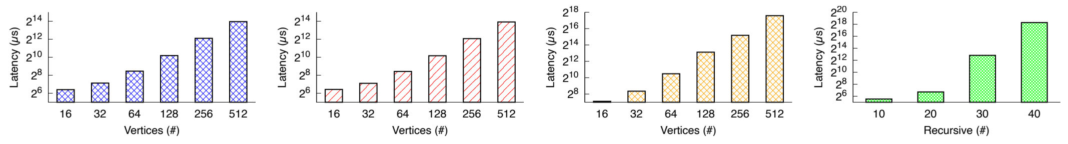


Figure 4 Broadcast latency with varying packet sizes

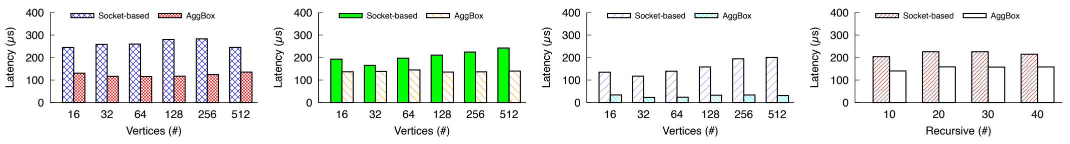
AggBox maintains consistently lower broadcast latency across all packet sizes. For 1024-byte packets, AggBox achieves 238.07 µs versus the baseline's 265.34 µs (10.3% reduction). While AggBox's latency remains stable (165.55-238.07 µs), the baseline shows increasing latency with packet size, demonstrating the impact of user-space processing overhead. This stability highlights the efficiency of kernel-space processing in handling varying data volumes.

Application-level Performance. To assess AggBox's real-world impact, we integrate our in-kernel broadcasting optimizations into four distributed applications: Breadth-First Search (BFS), Depth-First Search (DFS), PageRank, and Fibonacci computation. For BFS, DFS, and PageRank, input graphs vary from 16 to 512 vertices, generated using a random graph function to ensure diverse structures. For Fibonacci, the number of recursive calls ranges from 10 to 40.



(a) BFS **(b)** DFS **(c)** PageRank **(d)** Fibonacci

Figure 5 Computation time for various applications with different input sizes



(a) BFS **(b)** DFS **(c)** PageRank **(d)** Fibonacci

Figure 6 Broadcast latency for various applications with different input sizes

Figure 5 shows computation times for each application with varying inputs. As expected, computation time increases with input size, highlighting different computational characteristics. BFS and DFS range from microseconds for small graphs to milliseconds for larger ones (512 vertices). PageRank's times reach hundreds of milliseconds for the largest graphs. Fibonacci's computation time grows exponentially with the number of recursive calls.

Figure 6 compares broadcast latency between AggBox and the baseline. AggBox consistently achieves lower latency across all applications and input sizes. This performance gain is particularly notable when computation times are small, as communication overhead becomes a dominant factor.

For example, in BFS and DFS with small input graphs (16 vertices), computation time is in microseconds while broadcast latency is in hundreds of microseconds. Here, AggBox's optimizations significantly reduce latency. For BFS with 16 vertices, AggBox achieves a latency of 130.31 µs, compared to 244.87 µs for the baseline, a 46.8% reduction. For DFS with 16 vertices, AggBox achieves 136.78 µs versus 192.67 µs for the baseline, a 29.0% reduction.

As input sizes increase and computation times grow, the impact of broadcast latency on overall performance lessens. However, AggBox still outperforms the baseline. In PageRank with 512 vertices, AggBox achieves 31.10 µs latency compared to 200.36 µs for the baseline, an 84.5% reduction. For Fibonacci with 40 recursive calls, AggBox achieves 158.22 µs latency, while the baseline is 214.44 µs, a 26.2% reduction.

These experiments demonstrate AggBox's real-world benefits by reducing latency in diverse distributed applications. By optimizing kernel-level broadcasting, AggBox enhances communication efficiency and scalability, leading to faster execution times and better performance. Significant gains in scenarios with shorter computation times highlight the critical importance of optimizing communication for application performance.

5.2 Aggregation Experiments

Impact of Data Size. We evaluate the aggregation throughput of AggBox compared to the baseline implementation (Socket-based) using different data sizes. We use four machines to send data and aggregate it on one machine. Figure 7 shows the results.

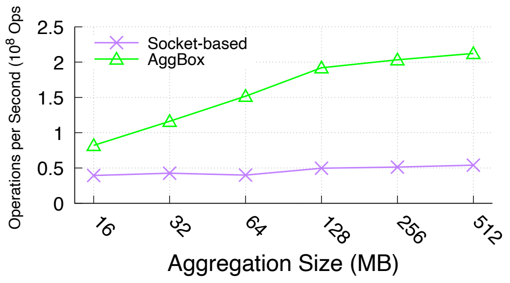


Figure 7 Data size vs. aggregation throughput

AggBox achieves higher aggregation throughput across all data sizes. At 512 MB, AggBox reaches 212,288,843 ops/s versus the baseline's 54,025,990 ops/s (3.9× improvement). The widening performance gap with larger data sizes demonstrates how kernel-space aggregation eliminates the memory copy and context switch overhead that increasingly burdens user-space processing.

Impact of Computation Cost. We evaluate the impact of computation cost on the performance of AggBox and the baseline implementation. We use XDP to add all received matrices and perform GEMM (general matrix multiplication) in the user space for AggBox, while the baseline performs both addition and GEMM in the user space. We use 4 senders and vary the matrix size (N) and measure the average time taken for the aggregation operation. Figure 8 shows the results.

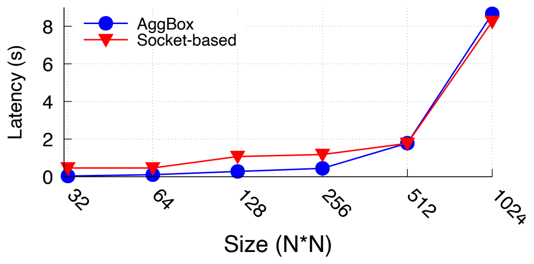
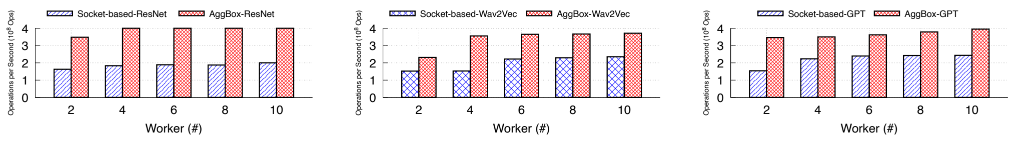


Figure 8 Computation cost vs. aggregation time

As matrix size increases, computation dominates and the performance gap narrows. At matrix size 1024, AggBox takes 8.3 seconds versus the baseline's 8.2 seconds, a negligible difference. This expected behavior occurs because AggBox optimizes communication, not computation. When GEMM operations dominate runtime, kernel-space aggregation provides minimal benefit as both systems are bound by user-space computation.

Parameter Servers Performance. We evaluate the performance of AggBox in parameter servers using three different models: ResNet-34 [9] (vision), GPT-2 [25] (text), and Wav2Vec [2] (audio). Figure 9 shows the performance comparisons between AggBox and the baseline (socket-based) for each model.



(a) ResNet34 **(b)** Wav2Vec **(c)** GPT-2

Figure 9 Performance comparisons of different models

AggBox consistently outperforms the baseline across all three models, with performance gaps widening as workers increase. The improvements stem from kernel-space aggregation of model updates, eliminating user-kernel transitions during gradient synchronization. AggBox's consistent gains across vision (ResNet34), text (GPT-2), and audio (Wav2Vec) models demonstrate its applicability to diverse ML workloads.

To quantify the reduction in training time achieved by AggBox, we calculate the percentage improvement over the baseline for each model and number of workers. Table 3 shows the results.

Table 3 Reduction in aggregation time for parameter servers

Model	Reduction in aggregation time (%)				
	2 workers	4 workers	6 workers	8 workers	10 workers
ResNet34	53.3%	54.2%	55.2%	56.1%	54.6%
Wav2Vec	51.7%	53.1%	54.3%	55.2%	56.1%
GPT-2	55.5%	36.2%	33.9%	34.4%	36.9%

The results show that AggBox achieves significant reductions in aggregation time across all models and worker configurations. For ResNet34 and Wav2Vec, the reduction in aggregation time ranges from 51.7% to 56.1%, indicating that AggBox can effectively accelerate the aggregation process by more than half. For GPT-2, the reduction in aggregation time is slightly lower but still substantial, ranging from 33.9% to 55.5%.

These findings highlight the practical impact of AggBox in accelerating distributed training workloads. By reducing the aggregation time, AggBox enables faster iterations and experimentation, which is crucial in the development and deployment of machine learning models. The significant time savings achieved by AggBox can lead to improved productivity and faster time-to-market for machine learning applications.

MapReduce Performance. We evaluate the performance of AggBox in a MapReduce scenario, using a word counting job with different input data sizes (2 million to 10 million words) and varying numbers of mappers (2 to 10). Figure 10 shows the results, comparing the performance of AggBox against the baseline implementation.

AggBox significantly outperforms the baseline across all configurations. With 2 mappers and 4 million words, AggBox achieves 126.25 million words/s versus the baseline's 5.72 million words/s (22× improvement). Performance scales with more mappers: at 10 mappers and 10 million words, AggBox reaches 257.66 million words/s compared to 5.98 million words/s for the baseline (43× improvement).

The performance gains stem from AggBox's kernel-space aggregation eliminating user-kernel crossings during the reduce phase. Our MapReduce workload is network-bound since the reduce operations (word count summation) are simple, making communication the bottleneck. The baseline's flat throughput across configurations indicates saturation from

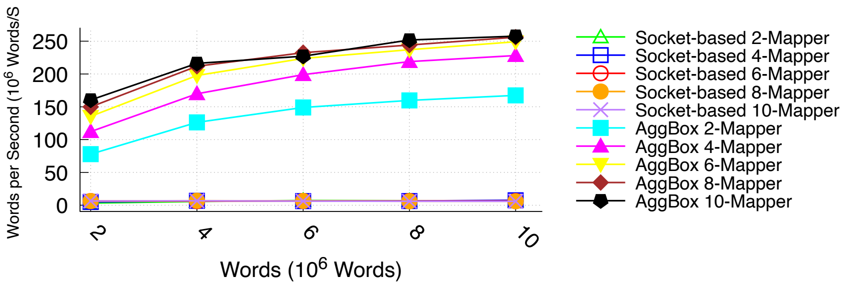


Figure 10 MapReduce performance comparison

user-space processing overhead, while AggBox's kernel processing efficiently handles the high packet rates from bursty mapper outputs. This demonstrates AggBox's effectiveness for network-bound workloads where communication dominates computation.

Overhead analysis We evaluate the CPU usage of AggBox on isolated CPUs when using different numbers of senders for aggregation operations. Figure 11 shows the CPU usage as the number of senders increases from 2 to 30.

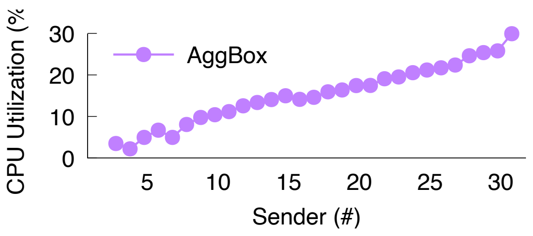


Figure 11 CPU utilization for aggregation operations

The results demonstrate that AggBox maintains low CPU usage even as the number of senders increases. When the number of senders is 2, the CPU usage is only 1.38%, indicating that AggBox can efficiently handle aggregation operations from a small number of senders. As the number of senders increases to 10, the CPU usage remains relatively low at 6.70%. This shows that AggBox can accommodate a moderate number of senders without incurring significant overhead.

Even when the number of senders reaches 30, the CPU usage is still manageable at 29.92%. This demonstrates AggBox's ability to scale and handle aggregation operations from a large number of senders while maintaining acceptable CPU utilization. The gradual increase in CPU usage as the number of senders grows can be attributed to the increased workload associated with handling aggregation operations from multiple senders. However, the rate of increase in CPU usage is sublinear, indicating that AggBox's performance scales well with the number of senders.

The efficient CPU utilization can be attributed to several factors. First, AggBox's use of eBPF and kernel-space aggregation allows for fast and efficient processing of incoming aggregation packets. By offloading the aggregation operations to the kernel space, AggBox reduces the overhead associated with user-kernel crossings and context switches.

6 Discussion and Future Work

AggBox's design, with its runtime library, in-kernel broadcasting and aggregation components, and retransmission mechanism, provides a comprehensive solution for optimizing communication in distributed systems. By leveraging the power of eBPF and kernel-space processing, AggBox achieves significant performance improvements while maintaining compatibility with existing systems.

While AggBox can accelerate distributed communication by offloading broadcasting and aggregation operations to the kernel space using eBPF, there are several limitations and areas for future improvement.

large-scale data center deployment. AggBox is designed to improve the performance of distributed systems by reducing the overhead of user-kernel crossings and kernel networking stack traversals. However, the scalability of the system in terms of the number of senders and receivers may be limited by the capacity of the underlying network infrastructure and the processing power of the participating nodes. In extremely large-scale environments with millions of senders and receivers, the performance gains achieved by AggBox may be less pronounced, as the network becomes the primary bottleneck.

To further enhance scalability, future work could explore the integration of AggBox with advanced networking technologies, such as RDMA (Remote Direct Memory Access) or programmable switches, which can offload certain network processing tasks to dedicated hardware. Additionally, investigating techniques for efficient load balancing and partitioning of workloads across multiple nodes could help distribute the processing overhead and improve overall system scalability.

Applicability in HPC Clusters. High-performance computing (HPC) clusters often rely on specialized networking technologies, such as InfiniBand (IB) or RDMA over Converged Ethernet (RoCE), which provide high-bandwidth, low-latency communication capabilities. These environments typically employ hardware-based multicast mechanisms for efficient data distribution.

In its current form, AggBox may not be directly applicable to HPC clusters that heavily rely on these specialized networking technologies. The system is primarily designed to optimize communication in commodity Ethernet-based networks, where the overhead of user-kernel crossings and kernel networking stack traversals is more significant.

However, the concepts and techniques used in AggBox, such as offloading communication operations to the kernel space and leveraging eBPF for efficient packet processing, could potentially be adapted to work with IB or RoCE networks. Future research could explore the integration of AggBox with these technologies, taking into account their unique characteristics and hardware-based multicast capabilities.

Scope in GPU-Centric Deployments. Similar considerations apply to GPU-based ML training, which dominates modern distributed learning workloads. In typical GPU training clusters, communication occurs at two layers: GPU-to-GPU transfers via NVLink or GPU-Direct RDMA for intra-node communication, and host-to-host transfers via Ethernet for inter-node gradient synchronization. AggBox targets the latter layer and remains relevant even in GPU-centric environments.

AggBox is well-suited for parameter server architectures where gradient aggregation occurs on CPU-based servers even when workers use GPUs, as demonstrated in our evaluation with

ResNet-34, GPT-2, and Wav2Vec. It also benefits commodity cloud deployments where GPU-Direct is unavailable and gradients must traverse the host network stack, as well as edge and federated learning scenarios where heterogeneous devices participate in training [33]. For deployments with full GPU-Direct RDMA support such as NVIDIA DGX clusters, hardware-based solutions provide optimal performance and AggBox offers limited additional benefit. However, many organizations use commodity cloud instances where AggBox's kernel-level optimizations can significantly reduce communication overhead without requiring hardware modifications.

Impact of Approximation in Scientific Computing. AggBox's quantization approach for floating-point operations may introduce approximations that, while acceptable for ML/AI workloads, could be problematic for scientific computing where small errors accumulate over time. Scientific simulations requiring strict precision should carefully evaluate AggBox's accuracy trade-offs. Future work could explore alternative approaches such as fixed-point arithmetic or hardware-assisted floating-point processing, and provide user controls for precision levels to better support scientific computing applications.

Despite these limitations, AggBox demonstrates the potential of using eBPF and kernel-space optimizations to accelerate distributed communication in many common scenarios. By addressing the identified challenges and exploring new optimizations, AggBox can be further enhanced to support a broader range of use cases and provide even greater performance benefits.

7 Related Work

eBPF Applications. Enhanced Berkeley Packet Filter (eBPF) facilitates advanced networking and systems operations. Electrode [39] accelerates distributed protocols using safe in-kernel packet operation. XAgg [37] deploys an XDP-based aggregation to enhance gradient aggregation in distributed machine learning. BMC [8] uses eBPF for a kernel-level cache, boosting UDP Memcached GET request throughput. SPRIGHT [24] enhances performance in serverless architectures via eBPF-driven fast packet forwarding. Syrup [12] employs eBPF maps for seamless data exchange across system layers, supporting custom scheduling policies. XRP [38] offloads kernel storage operations like B-tree lookups to eBPF, reducing overhead. In contrast, AggBox focuses on leveraging eBPF and TC/XDP hooks to optimize broadcasting and aggregation operations in distributed systems, aim to enabl efficient and scalable communication for a wide range of applications.

Distributed in-network computing. Previous research efforts mainly concentrated on distributing programs across devices to support specific paths or symmetric topologies [4, 5, 19]. ClickINC [34] supported multi-path deployment in data centers, assuming network symmetry. Flightplan [28] demonstrated P4 program disaggregation for rack-scale deployments, introducing routing changes to achieve processing objectives. ATP [17] proposed a dynamic, best-effort in-network aggregation service for multi-tenant deep learning, facilitating efficient switch resource sharing across jobs. Instead of focusing on specific application domains or network topologies, AggBox provides a generic framework for distributing in-network aggregation and broadcast communication without relying on specific network characteristics.

8 Conclusion

We introduced AggBox, a system utilizing eBPF and kernel-level packet processing to optimize broadcasting and aggregation in distributed systems. By offloading these operations to the kernel and designing protocols specifically for eBPF constraints, AggBox reduces latency and overhead while maintaining compatibility with standard Ethernet deployments. Our evaluation shows AggBox consistently outperforms optimized baseline implementations, achieving up to 84.5% reduction in broadcast latency and 56.1% reduction in aggregation time.

AggBox addresses unique challenges in eBPF-based collective operations through innovative design choices including group-based retransmission, hierarchical aggregation support, and quantization for floating-point handling. While not a replacement for specialized hardware solutions like RDMA, AggBox fills an important gap for commodity cloud deployments requiring better communication performance without hardware modifications.

References

1 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: `https://www.tensorflow.org/`.

2 Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations, 2020. `arXiv:2006.11477`.

3 K. A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016. URL: `https://arxiv.org/abs/1611.04482`.

4 Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. Speed: Resource-efficient and high-performance deployment for data plane programs. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–12, 2020. `doi:10.1109/ICNP49622.2020.9259384`.

5 Xiang Chen, Hongyan Liu, Qingjiang Xiao, Kaiwei Guo, Tingxin Sun, Xiang Ling, Xuan Liu, Qun Huang, Dong Zhang, Haifeng Zhou, Fan Zhang, and Chunming Wu. Toward low-overhead inter-switch coordination in network-wide data plane program deployment. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 370–380, 2022. `doi:10.1109/ICDCS54860.2022.00043`.

6 Hong-Ning Dai, Raymond Chi-Wing Wong, Hao Wang, Zibin Zheng, and Athanasios V Vasilakos. Big data analytics for large-scale wireless networks: Challenges and opportunities. *ACM Computing Surveys (CSUR)*, 52(5):1–36, 2019.

7 Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network aggregation for shared machine learning clusters. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 829–844, 2021.

8 Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021. URL: `https://www.usenix.org/conference/nsdi21/presentation/ghigoff`.

9 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. `arXiv:1512.03385`.

10 Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A generic service to provide in-network aggregation for key-value streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 33–47, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3575693.3575708`.

11 Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.

12 Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3477132.3483548`.

13 Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

14 Guangchen Lan, Dong-Jun Han, Abolfazl Hashemi, Vaneet Aggarwal, and Christopher Brinton. Asynchronous federated reinforcement learning with policy gradient updates: Algorithm design and convergence analysis. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

15 Guangchen Lan, Huseyin A Inan, Sahar Abdelnabi, Janardhan Kulkarni, Lukas Wutschitz, Reza Shokri, Christopher G Brinton, and Robert Sim. Contextual integrity in LLMs via reasoning and reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2025.

16 Guangchen Lan, Sipeng Zhang, Tianle Wang, Yuwei Zhang, Daoan Zhang, Xinpeng Wei, Xiaoman Pan, Hongming Zhang, Dong-Jun Han, and Christopher G Brinton. Mappo: Maximum a posteriori preference optimization with prior knowledge. *arXiv preprint arXiv:2507.21183*, 2025.

17 ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021. URL: `https://www.usenix.org/conference/nsdi21/presentation/lao`.

18 Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. A review of applications in federated learning. *Computers & Industrial Engineering*, 149:106854, 2020.

19 Yiran Li, Kevin Gao, Xin Jin, and Wei Xu. Concerto: cooperative network-wide telemetry with controllable error rate. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 114–121, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3409963.3410499`.

20 Priyanka Mary Mammen. Federated learning: Opportunities and challenges. *arXiv preprint arXiv:2101.05428*, 2021.

21 Navneet Kumar Pandey, Kaiwen Zhang, Stéphane Weiss, Hans-Arno Jacobsen, and Roman Vitenberg. Distributed event aggregation for content-based publish/subscribe systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, page 95–106, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2611286.2611302`.

22 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas

Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. arXiv:1912.01703.

23 pcap(3PCAP). pcap(3PCAP) man page | TCPDUMP &amp; LIBPCAP — tcpdump.org. https://www.tcpdump.org/manpages/pcap.3pcap.html. [Accessed 15-06-2024].

24 Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 780–794, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544259.

25 Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

26 Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/sapio.

27 Jianchang Su, Yifan Zhang, Linpu Huang, and Wei Zhang. Boad: Optimizing distributed communication with in-kernel broadcast and aggregation. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 51–57, 2024.

28 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 *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 571–592. USENIX Association, April 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/sultana.

29 Zeyi Tao and Qun Li. eSGD: Communication efficient distributed deep learning on the edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/hotedge18/presentation/tao.

30 Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

31 Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *ACM Comput. Surv.*, 53(2), mar 2020. doi:10.1145/3377454.

32 Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

33 Zhi Wang, Tao Gong, Shu Hui Huang, and Bo Yi. Graph neural network with soft actor-critic and attention based large model for intelligent edge routing in consumer internet of things. *IEEE Transactions on Consumer Electronics*, 2025.

34 Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, and Bin Liu. Clickinc: In-network computing as a service in heterogeneous programmable data-center networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 798–815, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3603269.3604835.

35 Sepehr Abbasi Zadeh, Ali Munir, Mahmoud Mohamed Bahnasy, Shiva Ketabi, and Yashar Ganjali. On augmenting tcp/ip stack via ebpf. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 15–20, 2023.

36 Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016. doi:10.1145/2934664.

37 Qianyu Zhang, Gongming Zhao, Hongli Xu, and Peng Yang. Xagg: Accelerating heterogeneous distributed training through xdp-based gradient aggregation. *IEEE/ACM Transactions on Networking*, pages 1–15, 2023. `doi:10.1109/TNET.2023.3339524`.

38 Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association. URL: `https://www.usenix.org/conference/osdi22/presentation/zhong`.

39 Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association. URL: `https://www.usenix.org/conference/nsdi23/presentation/zhou`.

40 Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel distributed transactions with eBPF. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 401–417, Santa Clara, CA, April 2024. USENIX Association. URL: `https://www.usenix.org/conference/nsdi24/presentation/zhou-yang`.