A Generic Service to Provide In-Network Aggregation for Key-Value Streams

Yongchao He
IIIS, Tsinghua University
Beijing, China

Wenfei Wu*
Peking University
Beijing, China

Yanfang Le
Intel, Barefoot Switch Division
Santa Clara, CA, USA

Ming Liu
University of Wisconsin-Madison
Madison, WI, USA

ChonLam Lao
Harvard University
Cambridge, MA, USA

ABSTRACT

Key-value stream aggregation is a common operation in distributed systems, which requires intensive computation and network resources. We propose a generic in-network aggregation service for key-value streams, *ASK*, to accelerate the aggregation operations in diverse distributed applications. ASK is a switch-host co-designed system, where the programmable switch provides a best-effort aggregation service, and the host runs a daemon to interact with applications. ASK makes in-depth optimization tailored to traffic characteristics, hardware restrictions, and network unreliable natures: it vectorizes multiple key-value tuples' aggregation of one packet in one switch pipeline pass, which improves the per-host's goodput; it develops a lightweight reliability mechanism for key-value stream's asynchronous aggregation, which guarantees computation correctness; it designs a hot-key agnostic prioritization for key-skewed workloads, which improves the switch memory utilization. We prototype ASK and use it to support Spark and BytePS. The evaluation shows that ASK could accelerate pure key-value aggregation tasks by up to 155 times and big data jobs by 3-5 times, and be backward compatible with existing INA-empowered distributed training solutions with the same speedup.

CCS CONCEPTS

• **Networks → In-network processing**.

KEYWORDS

In-Network Aggregation, P4, Key-Value, Big Data.

ACM Reference Format:
Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. 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 '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575708

*Wenfei Wu is the corresponding author.

1 INTRODUCTION

Aggregating multiple key-value streams is an operation widely existing in various distributed systems, e.g., *reduce()* in Big Data [7, 29, 68], *AllReduce()* in Distributed Training [32, 47, 61, 67], *MPI_Reduce()* in High-Performance Computing (HPC) [12, 46], *SUM()* in Database [17, 48, 64], etc. The aggregation operation may require intensive resources on computation, disk IO, and network [56, 69], and could dominate numerous workloads' overall performance. For example, in distributed training, the gradient aggregation can take up to 79% of the training time [61], and in the typical MapReduce job, such as WordCount, the *ReduceByKey()* operation takes 94.67% of the time [37]. In addition, Reduce-related collective functions are the most significantly used and time-consuming operators in hundreds of open-source HPC applications [46].

Among the many acceleration solutions for aggregation, a recent communication and computation primitive — *In-Network Aggregation (INA)* [47, 61] — has gained wide attention. It uses a programmable switch[1] to aggregate multiple traversing streams into one, which reduces the network traffic volume and consequently accelerates the entire aggregation task. One class of INA solutions has demonstrated the success in scenarios such as distributed training [47, 52, 61, 66, 67] and HPC [33]. In addition to these end-to-end systems, another class of preliminary showcases [25, 60], as well as our strawman solution (§2.2), demonstrates the switch's capability to perform key-value stream aggregation much faster than hosts.

However, distributed training-oriented INA solutions [32, 33, 47, 52, 61, 66] are not generally applicable to the *key-value stream aggregation* scenarios (§2.1.1). By comparative analysis (§2.1), we reveal that these solutions target a traffic pattern of *value stream aggregation* (§2.1.2), which is a special case of key-value stream aggregation. Value stream aggregation is *synchronous aggregation*, whose design is simplified by its traffic pattern. In contrast, the key-value aggregation has to be *asynchronous aggregation* (§2.1.3), which fails all existing reliability mechanisms [32, 34, 38, 47, 49]. Alternatively, the class of key-value aggregation showcases [25, 60] lacks system-wide considerations such as application interfacing, correctness guarantee, and performance maximization, and can hardly be practical to support numerous distributed applications. With state of the art insofar, building an *end-to-end system* to provide in-network key-value aggregation for distributed applications, as well as advancing the end-to-end performance, have not received attention.

[1]We use "switch" to denote programmable switch in the following text.

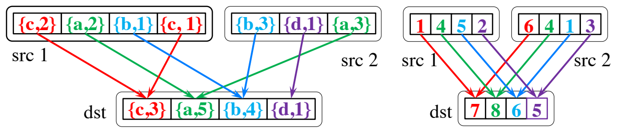
In this paper, we propose a solution named *ASK* to provide correct and performant *Aggregation Service for Key-value streams* in distributed systems. ASK is a general-purpose aggregation service decoupled from specific applications, allowing multiple applications (instances) to multiplex it. ASK co-designs the switch and hosts, where the host runs a dedicated service to exchange key-value data with applications through inter-process communication, and the switch performs a best-effort aggregation service for traversing key-value streams between hosts (§3.1). To maximize the performance gain without compromising the correctness, ASK makes in-depth customization and optimization, tailored to the switch hardware restrictions, traffic characteristics, and the network unreliable natures. In details, ASK overcomes three challenges (§2.3).

First, the system needs to vectorize one packet's multi-tuple aggregation in one switch pipeline pass to promote the system goodput. However, the switch programmability and memory access mode provided by Protocol Independent Switch Architecture (PISA) [23] restrict the vectorization. ASK co-designs the switch memory layout and host packetization to achieve the vectorization and support variable-length keys in real-world workloads (§3.2). Second, the system needs a reliability mechanism specifically for asynchronous aggregation; but none of existing solutions could function correctly, and vectorized packet aggregation further complicates the switch states and logic. ASK carefully crafts the host sliding-window scheme and the switch deduplication logic to achieve reliability and correctness and additionally improves the system scalability by reusing persistent connections in the host service (§3.3). Third, the switch has to address keys to switch memory in runtime. Still, the practical key-distribution-skewed workload could lead to low utilization of switch memory when cold keys first reserve the switch memory. ASK devises a shadow copy mechanism to fetch the intermediate results from the switch periodically and reset the switch memory, allowing hot keys a second chance to reserve the switch memory (§3.4).

We prototype ASK and integrate it with Spark [68] and BytePS [39, 58]. Experiments on microbenchmark show that ASK can (1) improve key-value aggregation throughput by up to 155 times with the same CPU usage, (2) saturate the high-speed network at a line rate of near 100Gbps, and (3) scale the total aggregation throughput linearly with the number of the servers, up to 92.61Gbps×8 for eight servers. Consider computation and communication together, ASK can perform key-value aggregation at a higher speed than host-only systems, e.g., speeding up big data jobs by up to 4.56 times while reducing the CPU usage by 88.7%, and achieve the same acceleration as INA-based training systems [47, 61] in distributed training.

In summary, the contributions of this paper are as follows:

- We build a general-purpose end-to-end system ASK to provide in-network key-value aggregation as a service to diverse distributed applications.
- We vectorize the multi-key packet aggregation under switch hardware restrictions to improve the network goodput, which greatly boosts the overall performance of applications.
- We build a lightweight reliability mechanism specifically for asynchronous aggregation, which guarantees the correctness of aggregation computation.



(a) Key-value stream aggregation (b) Value stream aggregation

Figure 1: Example of Aggregation Patterns.

- We agnostically prioritize hot-key aggregation, which improves the switch memory utilization for key distribution skewed workloads.
- We prototype ASK and make an extensive evaluation to show that ASK supports diverse distributed applications and accelerates system performance significantly.

2 BACKGROUND AND MOTIVATION

Key-value stream aggregation is asynchronous. Programmable switches have the potential to accelerate the process, but the end-to-end system design still faces several challenges.

2.1 Aggregation Patterns

2.1.1 Key-Value Stream Aggregation. Formally, a key-value stream $f^{(m)}$ is denoted as a sequence of key-value tuples,

$$f^{(m)} =< (k_1^{(m)}, v_1^{(m)}), (k_2^{(m)}, v_2^{(m)}), \cdots, (k_{K_m}^{(m)}, v_{K_m}^{(m)}) >, \quad (1)$$

where K_m denotes the number of key-value tuples in the m^{th} stream. In multiple key-value stream aggregation ($1 \le m \le M$), a key k''s value in the final result is denoted as

$$v' \leftarrow \sum_{m=1}^{M} \sum_{i=1}^{K_m} v_i^{(m)} \; I(k_i^{(m)} = k'), \quad (2)$$

where $I(k_i^{(m)} = k')$ is the identity function returning 1 if two keys are equal and 0 otherwise. Figure 1(a) illustrates this aggregation pattern. Many workloads, e.g., MapReduce [30], in Big Data [62, 68] and Streaming Processing [15, 24, 45] follow this pattern.

2.1.2 Value Stream Aggregation. The value stream aggregation is actually vector aggregation and can be viewed as a special case of the key-value aggregation. Each value stream is denoted as an ordered sequence of K values

$$f^{(m)} =< v_1^{(m)}, v_2^{(m)}, \cdots, v_K^{(m)} >, \quad (3)$$

where the value index can be viewed as the key. After aggregating M value streams, the value at index i is

$$v_i \leftarrow \sum_{m=1}^{M} v_i^{(m)}. \quad (4)$$

Figure 1(b) illustrates the aggregation pattern, in which the *dst* generates a new value from multiple values by value's index in the stream. The gradient [35] tensor aggregation in the distributed training systems [47, 50, 61] is an typical example of value stream aggregation. And the collective operations, e.g., *AllReduce(), Reduce()*, in HPC [12, 31, 52] also take this aggregation pattern.

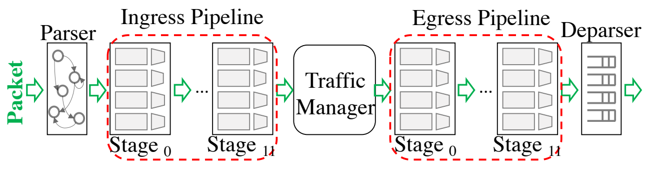


Figure 2: Protocol Independent Switch Architecture (PISA).

2.1.3 Synchronous v.s. Asynchronous INA.. INA [28, 47, 61, 66] has shown the promising performance gain due to the recent advances in programmable switches [4, 22, 23]. To perform INA, the switch memory is organized as a pool of *aggregators*, which is the computation and storage unit. When key-value streams traverse the switch, the switch assigns each key-value tuple to an aggregator by the key using an addressing scheme, e.g., runtime random hashing [47] or static linear allocation [61]. The aggregator performs the aggregation and consumes the packets. Upon aggregation completion, the switch writes the aggregation result to a packet and sends it to the destination host.

In value stream aggregation, all streams' keys (indices) are linear, contiguous, and aligned, and all senders are synchronized to send streams at the same pace. Thus, for each key, all its appearances (across streams) at the switch are synchronized. The switch can immediately know the aggregation completion, send the result to downstream, and release and reuse the aggregator; large streams can circularly use the limited aggregators. We refer to this aggregation pattern as *synchronous aggregation.*

In key-value stream aggregation, keys are unordered and unforeseeable (especially for real-time data streaming [15, 24, 45]), and there is no synchronization among senders. Keys have to be dynamically addressed to aggregators in runtime, and the switch has no idea about a key's last appearance as well as the key's aggregation completion. Thus, the switch cannot immediately send the result to downstream, and release and reuse the aggregator; the excessive keys in large streams have to fall back to hosts for processing. We refer to this aggregation pattern as *asynchronous aggregation.* Notably, we can forcibly adapt value streams to asynchronous aggregation, but cannot adapt key-value streams to synchronous aggregation.

2.2 Promise of In-Network Key-Value Aggregation

2.2.1 Potentials and Constraints of Programmable Switches. Programmable switches [5, 8, 18, 65] follow a PISA [23] architecture (Figure 2). Compared with the traditional switch, the programmable switch has *ingress/egress pipelines* to achieve the programmability on packets. One pipeline consists of a sequence of match-action stages, and each stage has circuits to run switch programs and memory (SRAM) to store states. The switch programs are user-defined ones written in domain-specific languages such as P4 [22], which can match packets on header fields and perform actions (e.g., arithmetics) on packets and the stage states. In existing INA solutions, the switch program writes values in packets to the switch memory and performs the aggregation operation.

Programmable switches can run various switch programs at line rate without affecting network functions (e.g., forwarding), typically much faster than the network I/O speed on hosts. For example, the total processing capacity of Intel^R Tofino3^{TM} ASIC [4] can be up to 25.6$Tbps$ (400Gbps×64 ports).

Programmable switches also have several constraints. (1) A pipeline has very limited memory resources (~15MB SRAM), which brings huge challenges for processing large streams. (2) The programming model is constrained: the memory on stages is isolated, and the program cannot use it as a uniform address space; a packet can only traverse all stages of a pipeline sequentially in the runtime, called one pass; memory can be declared as *register arrays*[2] in the program, but each register array can only perform one read and one write in one pass. The limited programmability further causes challenges to write correct and performant switch programs.

2.2.2 Strawman Solution. We present a strawman solution demonstrating the performance gain of offloading key-value stream aggregation to the switch. Since there is no end-to-end system designed yet, we make three assumptions to simplify the design of the strawman solution.

(1) Each packet carries one key-value tuple. In value streams, one packet can carry multiple values because the first value's index (key) can denote multiple contiguous values' indices. But in key-value streams, neighboring tuples cannot be represented by one key. And a switch memory register array cannot process multiple tuples. Also, the key size is set to 4 bytes in concert with the switch memory register size.
(2) The network is reliable, and no packet loss occurs in the experiment. Because asynchronous aggregation is a new pattern not supported by existing systems, its specific reliability mechanism is missing.
(3) All keys could fit into the switch memory. If not, the system needs an addressing scheme to assign keys to aggregators, which is still missing for asynchronous aggregation.

We set up the vanilla Spark [14] and the strawman solution on a single machine to run WordCount [29] and measure the aggregation throughput, respectively. In the strawman solution, the host sends each key-value tuple individually in a packet to the switch, the switch addresses each key to an aggregator and merges tuples, and the host finally fetches the result back. Other experiment settings (e.g., key size) are in §5.2. Figure 3(a) and 3(b) show that the in-network key-value aggregation outperforms the on-host aggregation. With the same number of CPU cores (16 cores), the maximum gain is up to 5 times; INA achieves line rate of 100Gbps with 16 cores, but the vanilla Spark achieves the peak throughput with 56 ones; even with all cores involved, the strawman solution's peak throughput is 3.4 times of the vanilla Spark. This experiment demonstrates the promising prospect of the in-network key-value aggregation, i.e., freeing up valuable CPU resources for complex computations while gaining higher performance.

2.3 Challenges

The strawman solution demonstrates the promise of in-network key-value aggregation, but the assumptions are not practical for

[2]A register acts as an aggregator in ASK, we use *register* and *aggregator* interchangeably in the following text.

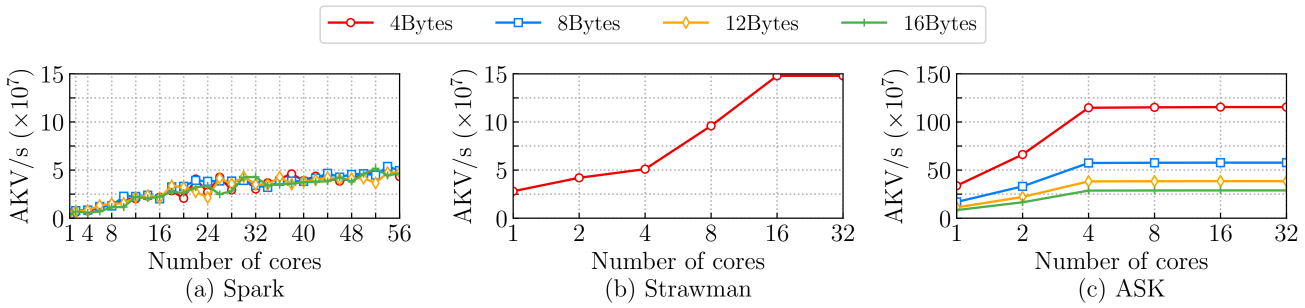


Figure 3: Aggregated key-value tuples per-second (AKV/s) on a single machine.

real-world tasks. Assumption (1) does not exploit the network bandwidth, assumption (2) could be violated in unreliable networks, and assumption (3) may not hold for real-world traces. We relax these assumptions and overcome three challenges to build a correct and performant end-to-end system ASK. By exploiting the hardware potentials (mainly the multi-key packet vectorization below) and taking advantage of the traffic characteristics in practical workload, ASK eventually achieves a performance boosting up to **155 times** compared with Spark (Figure 3(c)).

Vectorize multi-key packet aggregation with restricted switch memory access mode. A key-value tuple could be small in size, and a single-key packet would limit the network goodput. For streams whose packets only have one key-value tuple in the payload, even if the throughput reaches the line rate of 100Gbps, the goodput would only be 9.76Gbps[3]. To improve the goodput, a packet must carry multiple key-value tuples, called a *multi-key* packet. The multi-key packet further requires the switch to *vectorize* multiple tuples' aggregation within the packet's one pass in the switch pipeline. However, in a switch program, *the hardware restricts the register (aggregator) array to be read and written only once in one packet's pass*, contradicting the need for vectorization.

ASK co-designs the switch memory layout, i.e., two-dimensional aggregator arrays, and the host packet construction, i.e., flow space partition, to achieve efficient multi-key packet vectorization. In addition, ASK also devises coalesced key placement for variable-length keys in real-world workloads. (§3.2)

Devise a reliability mechanism specifically for asynchronous aggregation in unreliable networks. Applications expect the computation results to be *correct*, i.e., each key-value tuple aggregated exactly once. However, packet retransmission, common in data center [20, 21], could cause a packet to be falsely aggregated more than once. For asynchronous aggregation specifically, none of the existing reliability mechanisms (TCP and existing INA) could function correctly. In synchronous aggregation, e.g., ATP [47], SwitchML [61], etc. [32, 52], each aggregator spares a 1-bit state to record the appearance of a packet for deduplication. In asynchronous aggregation, however, this method cannot be applied, because a key's last appearance in key-value streams is unforeseeable, causing the state unbounded. To complicate matters further, a vectorized multi-key packet can *diverge* in all tuples' aggregation, i.e., some aggregated but some not, and these *partially-aggregated* packets require more complicated data structure and deduplication logic in the switch.

ASK designs a fine-grained state to record "per-tuple" appearance and co-designs the host sliding-window scheme and the switch reliability mechanism with deduplication. ASK also leverages the persistent connections in the host service to bound the state in the switch, avoiding state explosion. (§3.3)

Agnostically prioritize hot keys in asynchronous aggregation. In asynchronous aggregation, the switch addresses keys to aggregators in a First-Come-First-Serve (FCFS) scheme in the runtime. But the key distribution in a real-world workload could be skewed; an early *cold key* (less frequent) in the stream could occupy an aggregator for the entire lifetime of its aggregation task, wasting the aggregator's opportunity to serve *hot keys* (more frequent). Keys are unforeseeable, without providing a chance to pre-allocate aggregators for hot keys.

ASK devises a shadow copy mechanism to agnostically prioritize hot keys. The receiver periodically swaps the copy for aggregation, guiding traffic to the new copy, and fetching and resetting the old copy. Even if cold keys could occasionally preempt an aggregator in one period, hot keys still have the chance (and are more likely) to reseize aggregators back in the periodical swapping. (§3.4)

3 DESIGN

We design ASK to provide a correct and performant key-value aggregation service for the application. On hosts, ASK runs a daemon to exchange key-value data with applications through inter-process communication and prepare packets; on switches, ASK aggregates key-value tuples by keys in a best-effort manner (§3.1). The host daemon packs multiple key-value tuples into a packet with careful key addressing and placement to vectorize multi-tuple aggregation (§3.2). A lightweight, reliable transmission mechanism can ensure that ASK can always provide correct aggregation results even under unreliable network conditions (§3.3). ASK also provides a key agnostic prioritization mechanism to prevent the cold key from occupying the aggregator for the lifetime of the task, thereby improving the aggregator utilization (§3.4). The description below uses one switch as an example, but all the designs can be applied to multiple switches.

[3]A packet has a 24-byte framing overhead [9], 54-byte Ethernet/IP/INA header [47, 52, 61], and a 4-byte key and 4-byte value payload.

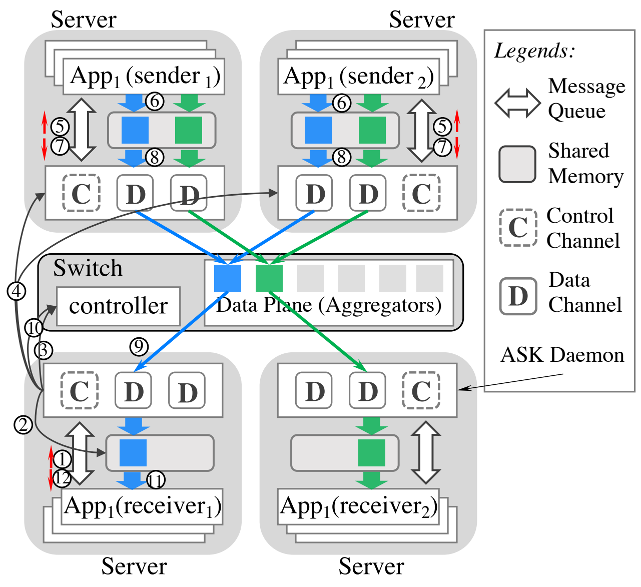


Figure 4: Overview of ASK. Receiver$_1$ and Receiver$_2$ start two concurrent aggregation tasks.

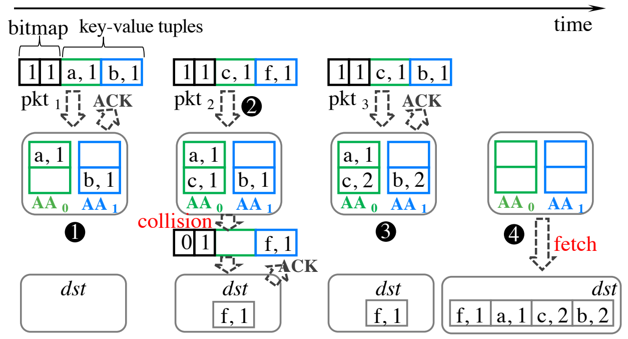


Figure 5: An aggregation example in ASK. ASK packet format is a bitmap followed by a list of key-value tuples. AA_0 and AA_1 are two aggregator arrays. *dst* is ASK daemon running on receiver host.

3.1 Architecture and Workflow

As shown in Figure 4, upon service booting, the switch initializes a set of aggregators in its data plane. Due to the need for vectorization (§3.2), the aggregator pool in ASK is organized as two-dimensional *aggregator array (AA)*, i.e., an array of AAs. The first dimension accesses an AA, and the second one accesses an aggregator. All AAs are of the same size.

ASK also sets up a daemon process on each server to interact with the applications. Each daemon would initialize a *control channel* and several *data channels* for aggregation tasks. These channels persistently run in the whole lifetime of the ASK service, and would serve multiple aggregation tasks. The data channel is between the host and the switch, and works in a duplex transmission mode: it can send key-value streams and receive the aggregation results. The workflow of executing an aggregation task is depicted in Figure 4, comprised of the following steps.

Task Setup. Applications submit aggregation tasks to ASK daemons. An aggregation task has multiple senders and one receiver on end-hosts and is initiated from the receiver (if senders decide to start a task, they notify the receiver, which would initialize the task, similarly to the receiver initiating it).

The receiver submits an aggregation task to its local ASK daemon with a task `ID` (①). The receiver-side daemon first allocates a piece of shared memory on the host for the task (the shared memory reduces memory footprint to copy data between the ASK daemon and the application) (②), and then applies for a switch memory region (range on AAs) from the switch controller (③). The receiver-side daemon notifies all sender-side daemons about the aggregation task by the control channel (④), including the task ID, the switch memory region, and the application-related context. Each sender-side daemon passes the notification to the corresponding application via a local message queue (⑤). The sender application allocates a piece of the shared memory, writes the key-value data into the shared memory (⑥), and then notifies its local daemon that the sending task is ready by a message of task ID and the shared memory region (⑦).

The sender-side daemon assigns each sending task to one of its data channels with load balancing, i.e., `hash(ID)` to a data channel. Each sending task is enqueued to a data channel, and a data channel serves multiple sending tasks in FIFO. The senders' and receiver's data channels would temporarily form an aggregation hierarchy.

The sender streams the packets to the receiver with the task `ID` and the destination IP address in the packet (⑧). The ASK switch uses the task `ID` to identify the aggregator memory region and the destination IP address to route packets to the aggregation task receiver. It then extracts the key-value tuples from the packet and aggregates each key-value tuple individually. For a key-value tuple, if the aggregator is available, the switch aggregates it and marks on the packet that it has been aggregated. If the switch aggregates all the key-value tuples within a packet, the switch replies an acknowledgment packet (ACK) to the sender; otherwise, it forwards this packet to the receiver. Upon receiving a data packet, the receiver-side data channel aggregates the remaining key-value tuples in the packet to the `ID` associated shared memory (allocated in the task beginning), and replies with an ACK.

Task Teardown. When a sender's key-value data are sent and acknowledged, the sender-side data channel sends a `FIN` packet to the receiver-side data channel. Upon receiving the `FIN`s of all senders, the receiver-side data channel fetches the results from the switch AA regions, merges them with its local results (⑨), and notifies the receiver application about the aggregation task completion with the shared memory address (⑫). The receiver application reads the aggregated results from the shared memory (⑪). Finally, the receiver-side daemon notifies the switch controller to deallocate the switch memory region for other future aggregation tasks to reuse ⑩.

Example. Figure 5 shows an example of the aggregation procedure where the switch and the receiver host receive three consecutive packets. There are two AAs in the switch in the example. Each packet has a two-bit bitmap and carries two key-value tuples, where the i^{th} bit in the bitmap indicates whether the i^{th} key-value tuple in the packet exists. pkt_1, pkt_2, and pkt_3 carry two key-value tuples; thus, the bits in each packet's bitmap are set. ❶ The first packet pkt_1's two key-value tuples (with key a and b) are mapped independently in the two AAs. Note that the i^{th} tuple in a packet will be dynamically hashed to an aggregator in the i^{th} AA in the switch. As both aggregators are available to pkt_1, all the tuples in the packet are aggregated in the switch and the switch replies an ACK to the sender. The ACK packet carries the same sequence number as pkt_1. ❷ The second packet pkt_2 is "partially aggregated": its key-value tuple $(c, 1)$ reserves a new aggregator, but $(f, 1)$ collides with $(b, 1)$ at the aggregator in AA_1. When the i^{th} tuple is consumed by the switch, the switch unsets the i^{th} bit in *bitmap* at the packet header. The packet, e.g., pkt_2, is forwarded with the new bitmap to the destination host. The receiver host uses the bitmap in the packet to find the remaining tuple, e.g., $(f, 1)$ in pkt_2, and aggregates it at the destination node. Finally, it replies with an ACK as the whole packet gets consumed there. ❸ The third packet pkt_3's two key-value tuples (with key c and b) are absorbed by the two AAs and replied ACK from the switch. Note that key b appears twice (in pkt_1 and pkt_3), it always belongs to the second key subspace (§3.2.2), and is encoded to the second tuple slot in packet payload and processed by the second AA (AA_1). ❹ Finally, the destination node fetches aggregated results from the switch, merges them with its local results, and clears the switch aggregators.

3.2 Multi-key Addressing and Placement

ASK co-designs the switch memory layout and the packet construction to vectorize the multi-key packet aggregation, improve aggregator utilization, and coalesce aggregators to support variable-length keys. We carefully divide the logic across host and switch to avoid the single-key multiple-spot and partial matching effects , and maximize the switch aggregator occupancy percentage.

3.2.1 *Vectorize Multi-key Packet Aggregation.*

In the switch, a packet would sequentially traverse the multiple stages of the packet processing pipeline [23], each stage with isolated and scarce SRAM (1280KB/stage × 16 stage/pipeline × 4 pipelines in Tofino3 [4]). SRAM are declared as register arrays in the switch program. Due to the hardware limitation, a register (aggregator) array can only be read/written once (§2.2.1) in one packet pass, but each stage allows 4 register arrays to be declared. Thus, ASK declares multiple register arrays to vectorize multi-tuple aggregation. The register arrays form a two-dimensional *aggregator array (AA)*. All AAs are of the same size, with each AA processing one tuple in the packet. Figure 6 shows the AA allocation on the switch memory. The first dimension accesses an AA, and the second one accesses an aggregator.

Each aggregator has a fixed size, denoted as $2n$ bits, e.g., 16/32/64bits. When storing a key-value tuple `{key, val}`, ASK uses bits 0 to $n-1$ (*vPart*) and bits n to $2n-1$ (*kPart*) to store `val` and `key`, respectively. If a key is less than n bits, ASK pads it n bits.

An ASK packet contains the ASK header after the IP header and the ASK payload. The payload has multiple slots with the same number as the AAs in the switch, and each slot can carry a key-value tuple to an AA in the switch.

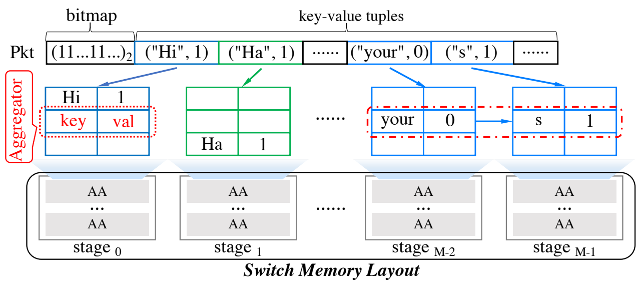


Figure 6: Switch memory layout with the aggregators. (AAs located in the same stage work in parallel.)

The overall aggregation process is similar to prior INA works [47, 61] for tuples whose keys fit in n bits. To support multi-key vectorization, ASK adds three functions. First, the host attaches an N-bit `bitmap` (the number of keys) to the packet header, where the i^{th} $(i = 0, \cdots, N-1)$ bit indicates the existence of the i^{th} key-value tuple in the payload. Second, when the switch performs the aggregation for an incoming packet, it feeds the i^{th} key-value tuple to the i^{th} AA. The example in Figure 6 shows that the tuple `("Ha", 1)`, which is placed in the second slot within the packet, is indexed to the second AA. ASK calculates the aggregator index within the AA, i.e., $hash(key)$, reads the corresponding aggregator's kPart key', and compares it with the key. The switch performs aggregation only if key' is blank or $key' = key$; otherwise, the tuple's aggregation fails, and it is forwarded to the destination host along with the packet for further processing. Third, upon a successful aggregation, ASK unsets the corresponding bit in `bitmap` to 0. If all key-value tuples in a packet are aggregated, the switch drops the packet and acknowledges the sender an ACK with the same sequence number as the data packet; otherwise, the switch sends the packet with remaining (with bit 1 in `bitmap`) key-value tuples to the receiver host. This procedure also indicates that a valid key-value tuple will be aggregated at either the host or the switch. Note that ASK is a best-effort service, but we can guarantee aggregation correctness (discussed in § 3.3).

3.2.2 *Sender-Assisted Addressing.*

If a key's multiple tuples are placed at different slots in packets, that key will occupy multiple aggregators in different AAs, which wastes aggregators. To avoid the *single-key-multiple-spot* problem, ASK further devises the packet construction at the sender. One crucial feature in key-value aggregation is that the operation is commutative, allowing us to arbitrarily change the key aggregation order. Further, the key stream is unforeseeable and could be unbounded, but the switch memory is scarce. Thus, stateful addressing schemes within the switch would be impractical. Hence, we develop an *ordered key-space partition* mechanism at the sender to classify each key to a dedicated AA and apply runtime addressing within the AA. So that one key will always be mapped to a single dedicated AA in the switch.

Assuming the keyspace is $\mathbb{K}$, and there are N *AAs* on the switch, ASK partitions $\mathbb{K}$ into N non-overlapping subspaces $\mathbb{K}_i$ $(i = 0, \cdots, N-1)$, where $\mathbb{K} = \bigcup_{i=0}^{N-1} \mathbb{K}_i$ and $\mathbb{K}_i \bigcap \mathbb{K}_j = \emptyset, i \neq j$. A key-value tuple `{key, val}` then falls into one subspace $\mathbb{K}_i$ with a hash function $\mathbb{F}$, i.e., $i = \mathbb{F}(key)\%N$.

When constructing a packet, the sender packs key-value tuples following the key subspaces – sequentially picking a key-value tuple from $\mathbb{K}_i$ and placing it in the i^{th} slot in the payload. If no key-value tuple is in $\mathbb{K}_i$, ASK will leave the i^{th} slot blank. The same key across different packets is always placed at the same slot in the payload and processed by the same AA on the switch. Note that the hash function $\mathbb{F}(\cdot)\%N$ should be uniform so that keys can be evenly distributed across subspaces.

3.2.3 *Coalesced Placement for Variable-Length Keys.*

Practical workloads could contain keys whose length is beyond an aggregator's kPart. ASK uses multiple aggregators to store a key-value tuple, where the key size could be variable. A naïve approach to aggregate 4∼15-byte keys under 32-bit aggregators is dividing each key into four segments, placing them independently in their AAs based on the hash function, performing four lookups sequentially during the aggregation phase, and aggregating the value only if all segments are matched. Unfortunately, this design could lead to aggregation errors. When two long keys X_1X_2 and Y_1Y_2 reserve four aggregators in two AAs independently, a third key X_1Y_2 would be falsely recognized as an existing key if the switch validates each of its segments independently.

The fundamental problem of the naïve design is that the segments in one long key have an association instead of independence. Thus, we advocate a design that coalesces multiple AAs in physically adjacent stages to store the whole long key-value tuple and addresses the key with a "unified" index (i.e., hashing the entire long key) in all AAs (Figure 6). After dividing a key-value tuple (key, val) into k parts, val is only stored in the last aggregator while others are left blank, i.e., $(key, val) = \{(key_1, 0), \cdots, (key_k, val)\}$.

As an example in Figure 6, a key-value tuple `("yours", 1)` is divided into two parts `{("your", 0), ("s", 1)}`, and fed to AA_{M-2} and AA_{M-1}, where both AAs use the unified array index (i.e., `hash("yours")`). Whereas in another key `"yourself"`, the `"your"` part would reserve a different aggregator (hashing `"yourself"`) other than that in `"yours"`.

ASK dedicates k groups of AAs for variable-length keys, each group with m AAs on physical adjacent stages (AA_i to $AA_{i+m-1}, \cdots, AA_{i+(k-1)m}$ to AA_{i+km-1}). Each group could handle keys with the length in the range $[n, nm)$ (n is the length of the aggregator kPart). We name these keys *medium keys.* Medium Keys are padded to nm. And each packet could carry k medium keys (for the k groups). Note that the dedicated AAs would not process short keys, because that would cause aggregation errors, e.g., a short key `"your"` could not be aggregated by at the aggregator reserved by `"yourself"`.

Together, the whole key space is first divided into short, medium, and long keys. The short and medium key subspace is further divided into subspaces as in §3.2.2. Long keys would be collected and sent to the receiver separately to the host receiver for processing, bypassing the switch. The choice of m should adapt to the key size distribution: a small m would cause more long keys without INA, but a large m would possibly cause packet payload and AAs to be wasted if medium keys cannot fill in the key-value tuple slots in the packet payload. In the current implementation, we empirically choose m to be 2 and k to be 8, and this value is suitable for most real-world datasets that we studied [1, 2, 16, 19].

3.3 Reliability and Correctness

Unreliable network conditions could lead to packet loss and retransmission. Duplicated packet appearance should not lead to values being aggregated again. In traditional TCP, the end-to-end reliability mechanism would remove the duplicated packet at the receiver, but ASK is more complicated: as a flow has three endpoints: the sender, the switch receiver, and the host receiver, if a "partially aggregated" packet is retransmitted, duplicated key-value tuples in the packet should be eliminated separately at the switch and the host receiver.

For example, a packet with two key-value tuples $[(a, 1), (b, 1)]$ is partially aggregated at the switch — $(a, 1)$ aggregated but $(b, 1)$ not, and then lost before arriving at the host receiver. The sender retransmits the packet to the switch. If the switch aggregates this packet directly, $(a, 1)$ would be aggregated twice. Still, if the switch forwards this packet directly, $(a, 1)$ would be aggregated by the host receiver and eventually aggregated twice when both receivers' results are merged. Either case is incorrect. The correct behavior should be "dropping" $(a, 1)$ and carrying $(b, 1)$ to the host receiver.

A straightforward way to avoid repeated aggregation is to implement a reliability mechanism at the switch and break the end-to-end flow into two separate reliable flows, where the switch serves as the receiver endpoint of the sender host, and the sender endpoint of the receiver host. Since switch memory is too scarce to record an unbounded key-value stream, we implemented a lightweight reliability mechanism, where the switch only maintains the per-tuple states for a window of packets in each flow. Furthermore, the switch only serves as the receiver endpoint, thus, the sender side functions, e.g., retransmission and timeout, are still on the host; and the ACK packets sent from the switch do not require any states maintained in the switch.

Host Sender. The sender maintains a *sliding window* whose maximum size is W packets. The sender always sends packets in the window, and ACKs would move the window forward and trigger sending new packets. A packet is retransmitted if its ACK does not arrive for a timeout. ASK does not use out-of-order ACKs to trigger retransmission, because both the switch and the host receiver could reply ACKs, causing out-of-order packets, which could be misinterpreted by the sender as packet loss; instead, ASK chooses a fined-grained timeout (100us v.s. Linux default 200ms). When all packets of one aggregation task are sent and acknowledged, the sender sends a `FIN` packet to the aggregation receiver, which fetches the aggregation results from the switch.

Switch Receiver. The switch maintains a *receive window* for a sender (data channel), which is a $2W$-bit array named `seen`. `seen` is circularly used to record each packet's appearance in the unbound key-value flow. There would be at most W packets in flight, and each is indicated by one bit in `seen`.

$$\begin{aligned} idx &\leftarrow pkt.seq\%(2W), \\ observed &\leftarrow switch.seen[idx]. \end{aligned} \tag{5}$$

The switch uses the packet sequence number to find the bit index in the `seen` bitmap and obtain the state from the `seen` bitmap. If a packet appears for the first time, i.e., its bit is unset, it is recorded and further participates in the aggregation procedure in §3.2.1; otherwise, it is a retransmitted packet, which would skip the switch aggregation. In both cases, the packet's indication bit in seen is set.

$$switch.seen[idx] \leftarrow 1. \tag{6}$$

As the array is circularly used, each packet would also clear a bit one window away for a future packet to use (at $idx + W$).

$$switch.seen[(idx + W)\%(2W)] \leftarrow 0. \tag{7}$$

There is a corner case where a very stale packet earlier than the current sliding window arrives at the switch (due to some long-time network delay), and it falsely overwrites the bit in seen. For example, the switch currently maintains a window with sequence number from $2W$ to $3W$. A packet with a sequence number of W arrives at the switch, which could falsely overwrite the state of the packet sequence $3W$ in seen. To resolve this issue, ASK additionally records the current window boundary and drops packets out of the boundary. ASK always records the maximum sequence number observed: $max\_seq = max(max\_seq, pkt.seq)$ for each packet. The current window range is $(max\_seq - W, max\_seq]$. If a packet has a sequence number smaller than or equal to $max\_seq - W$, it is a stale packet (earlier than the current window) and is dropped.

We note that (1) the array size should be at least $2W$ to guarantee that the record/clearance operation is correct. Because when observing the i^{th} packet, all packets in the range $[i - W + 1, i + W - 1]$ are possibly in the current window, and the cleared bit should be out of this range (whose size is $2W - 1$). (2) The receive window abstraction has a memory-compact design using the switch's atomic "test-and-set" instructions `set_bit(b)`[4] and `clr_bitc(b)`[5]. Its array size is W, saving 50% memory for seen. The design is as follows.

A Compact seen. The array seen is designed with W bits. The packet sequence $0 \cdots, S - 1$ is divided into segments of size W, i.e., for a packet with a sequence number s, it is in the segment of $q = \lfloor s/W \rfloor$ and its offset within the segment is $r = s\%W$.

According to $q\%2$, the segment can be an even segment or an odd one. The switch would iteratively observe packets from even and odd segments. In this design, seen uses 1/0 to denote the appearance of a packet in an even/odd segment. The operation for a packet is as follows.

$$observed \leftarrow \begin{cases} set\_bit(seen[r]) & \text{if } q \bmod 2 = 0, \\ clr\_bitc(seec[r]) & \text{if } q \bmod 2 = 1. \end{cases} \tag{8}$$

There are four cases when a packet arrives at the switch, and all cases correctly record the appearance and return the observation state.

- **Case 1:** An even-segment packet arrives, and its bit is 0. The operation would return 0, and set the bit.
- **Case 2:** An even-segment packet arrives, and its bit is 1. The operation would return 1, and set the bit.
- **Case 3:** An odd-segment packet arrives, and its bit is 1. The operation would return 0, and unset the bit.
- **Case 4:** An odd-segment packet arrives, and its bit is 0. The operation would return 1, and unset the bit.

[4]An atomic instruction that sets the bit b and returns the previous bit value.
[5]An atomic instruction that unsets the bit b and returns the complement of the previous bit value.

A single `set_bit()/clr_bitc()` instruction undertakes the three functions in the original design: recording the observation, returning previous record (flipped for odd-segment packets), and initializing the bit state one-window away. In returning previous record, for the odd segment's packets, 0 in seen means observed and returning its complement flips it to 1, matching the semantic of observed. In initializing the future bit, `set_bit` in an even segment would set the bit (to 1), making it prepared for the next odd segment, and `clr_bitc` in an odd segment would unset the bit (to 0), making it prepared for the next even segment.

There are two cases if a packet is identified as a retransmitted packet. If the packet was fully aggregated, it is dropped, and the switch replies its ACK. If the packet was "partially aggregated", the switch should "drop" the aggregated key-value tuples, then forward the packet with the remaining key-value tuples to the destination node.

To handle the partially-aggregated packets, we record packets' aggregation states, i.e., their bitmaps, at the end of the switch pipeline. The states are stored in a circular array of the same size as the window, called `PktState`, each array unit storing a bitmap. Each bit in a `PktState` unit indicates whether a tuple in one packet has been aggregated in the switch. When a packet is first observed ($observed = 0$), the packet's aggregation result is recorded by copying the packet's bitmap to the `PktState` as shown in Equation (9).

$$switch.PktState[idx\%W] \leftarrow pkt.bitmap. \tag{9}$$

When a packet is observed again ($observed = 1$) at the switch, the aggregation state is written back to the packet as shown in Equation (10).

$$pkt.bitmap \leftarrow switch.PktState[idx\%W]. \tag{10}$$

Thus, retransmitted partial-aggregated packets only carry valid key-value tuples (with bit 1 in `bitmap`) to the host receiver for further aggregation.

Host Receiver. The receiver host similarly maintains a receive window to record the packet's appearance. On the first appearance, a packet will be processed, i.e., an unaggregated key-value tuple in the packet is aggregated locally; on the later appearances, the packet is dropped; in both cases, the receiver replies with an ACK to the sender.

Bounding Switch States. The reliability mechanism requires the switch to maintain a per-flow state, which could affect the system's scalability. Since all streams on the same server multiplex the ASK data channels, the per-flow state (*seen* and *PktState*) can be associated with each data channel. In the current implementation, the max sliding window size is set to be 256, thus $256 + 256 \times 32$ bits ($1056B$, for seen and `PktState`) are needed for one data channel on the switch. A top-of-rack (TOR) switch can spare 264KB SRAM (out of ~15MB) to sufficiently support 64 servers.

3.4 Hot-Key Agnostic Prioritization

Key-value streams arrive at ASK online, and keys are unforeseeable. That is, each key's multiple appearances arrive asynchronously. In asynchronous aggregation, key-value tuples are addressed to aggregation in runtime in a First-Come-First-Serve (FCFS) manner; a reversed aggregator would be held by its key in the entire lifetime

Algorithm 1: Shadow Copy

```
1  Switch():
2    copy_indicator ← not(copy_indicator)
3  Read(key):
4    read_part ← 1 - copy_indicator
5    index ← hash(key) % N + read_part * N
6    return AA[index].val
7  Write(key, val):
8    write_part ← copy_indicator
9    index ← hash(key) % N + write_part * N
10   key′ ← AA[index].key
11   if key′ == key or is_blank(key′) then
12       Aggregate {key, val} into AA[index]
13       return true /* aggregation success */
14   return false /* conflict */
```

of the aggregation task, because the key's last appearance is unknown. However, real-world key-value streams could exhibit *key distribution skewness*. For example, according to Zipf's law [44], in all languages, the frequency of a word is inversely proportional to its index (index starts from 1) if all words are sorted in descending order by their frequency. If a low-frequency key, a.k.a. cold key, reserves an aggregator during the entire aggregation task, the late-arrived high-frequency keys, a.k.a., hot keys, could not preempt the aggregator. As a result, the aggregators would not be utilized to the best extent.

ASK makes a key-distribution agnostic design. It builds a shadow copy [61] for each AA, and periodically swaps between copies in runtime. When switching to a new copy, key-value tuples would get a new chance to reserve the empty aggregators. Statistically (for many rounds), hot keys would have more opportunity than (collided) cold keys to reserve the aggregator, and the overall aggregation efficiency could be improved.

For an AA with 2N aggregators, we divide it into two copies, referring to the first N aggregators and the last N aggregators. When the switch performs an aggregation operation on one of the copies, the host receiver can read the intermediate results on the other copy. The switch is modified with a *copy indicator* (one bit) to direct packets to one of the two copies. As shown in Algorithm 1, when the number of arrived packets at the host receiver reaches a tunable threshold, the host receiver sends a swapping notification to the switch; the switch flips the copy indicator (`Switch()` in line 1), which directs packets to the new copy; the receiver further fetches the results in the old copy and cleans up the old copy (`Read()` in line 3-6). At the same time, the switch will use the new copy to perform the aggregation operation (`Write()` in line 7-14).

Since PISA [23] restricts that each stage can only process one data packet at a time, when the switch pipeline is processing a copy-switching notification packet, there must be no other packets reading the *copy indicator*, thus ensuring the `Switch()` operation to be atomic. Moreover, in the runtime, `Read()` and `Write()` operate on two physically disjoint areas, avoiding the problem of read-write conflicts and ensuring the correctness of the final result.

4 IMPLEMENTATION

ASK consists of the aggregation function on the switch, and the network stack and service framework on hosts. The ASK switch aggregation function is implemented in P4 [22] with ~5000 lines of code, and the ASK network stack and service framework on hosts are implemented in DPDK [3] with ~4500 lines of C code. There are 32 AAs per pipeline, and each AA has 32768 aggregators. The switch's multiple pipelines can be used independently or chained together to form a longer pipeline. Thus, one packet can pack 32 8-byte key-value tuples using one pipeline or up to 128 8-byte key-value tuples if chaining pipelines. On the host, ASK daemon is implemented as a DPDK process with a thread pool. ASK uses one thread as the control channel and binds each data channel to one remaining thread in the pool.

The application interacts with ASK through a plugin. This plugin can convert data formats between the application and ASK. We build plugins for Spark and BytePS. The Spark [14] plugin has ~1800 lines of JAVA code, and the BytePS [6] plugin has ~500 lines of C++ code.

5 EVALUATION

In this section, we show ASK's good properties in supporting key-value stream aggregation.

- ASK effectively supports real-world and artificial key-value stream aggregation, and the performance gain is from both traffic reduction and computation offload (§5.2).
- The design choices of multi-key vectorization and hot-key prioritization effectively improve the system performance (§5.3 and §5.4).
- ASK accelerates the big data system (§5.5) and is backward compatible with value stream aggregation systems like distributed training (§5.6).

5.1 Experiment Settings

Cluster Setup. We conduct the experiment using one 32-port Tofino [5] switch and nine servers. Each server runs Ubuntu 18.04 (kernel 4.15.0-20) and has 56 Xeon^R Gold 5120T cores, 192GB RAM, 19TB disk, and one NVIDIA GeForce RTX 2080Ti GPU with driver version 430.34 and CUDA 10.0, and is connected to one of the switch port with a 100Gbps ConnectX-5 NIC [10].

Baselines. We evaluate ASK[6] in benchmarks, a big data system, and distributed training. (1) In benchmarking, we compare ASK with a host-only aggregation solution (PreAggr) [7] to demonstrate ASK can reduce CPU overhead while speeding up the key-value stream aggregation. (2) In big data system, the baseline is the vanilla Spark [68], Spark with RDMA for network IO acceleration (SparkRDMA [11]), and Spark with shared memory (SparkSHM[8]) which writes intermediate data on shared memory to exclude disk IO overhead. (3) In distributed training, we compare ASK with ATP [47] and SwitchML [61] to show that ASK can seamlessly support value stream aggregation, and have similar performance with

[6]By default, 4 ASK Data Channels are configured on each host.

[7]PreAggr: Instead of aggregating all key-value tuples at the receiver, each sender will aggregate key-value tuples by sorting them by key first and then merging neighboring tuples with the same key [14] (aka pre-aggregation).

[8]SparkSHM only use ASK for data transmission but does not perform INA, which excludes the influence of ASK's engineering optimization.

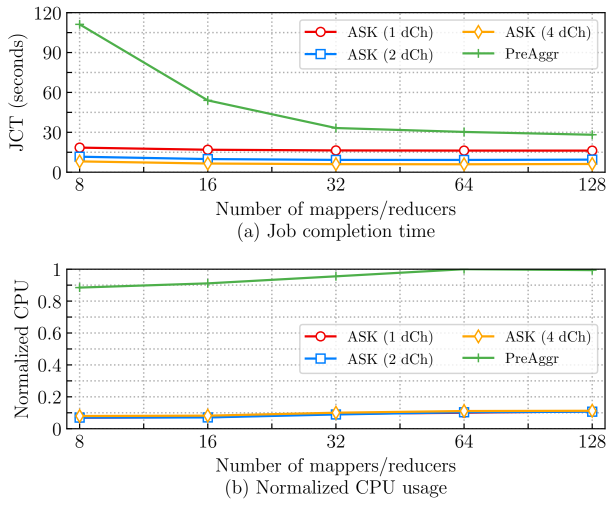


Figure 7: Comparison of ASK (1/2/4 Data Channels (dCh)) and end host based solution.

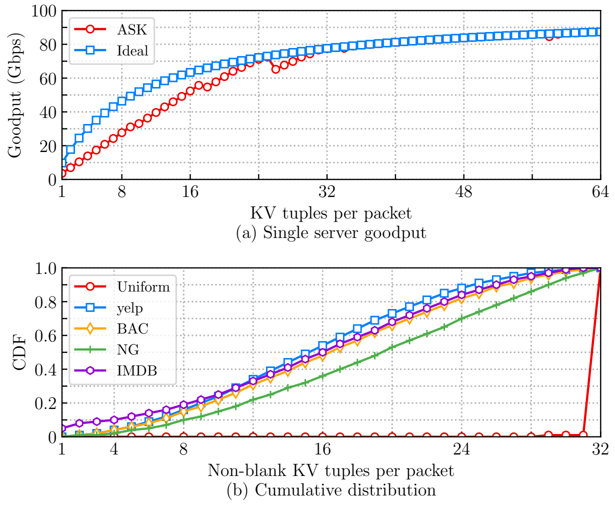


Figure 8: Impact of multi-key design on single server's goodput and Non-blank key-value tuples per packet.

single-key INA systems. (4) Finally, we compare ASK with pure network transmission (denoted as NoAggr) to study the system overhead, scalability, and tradeoff, explore how small packet size impacts the aggregation throughput, and give an analysis of ASK's scalability.

Datasets. When benchmarking the big data system, we use traces from production, including yelp [19], NG [2], BAC [16], and LMDB [55]. We also generate artificial traces such as uniform distribution and Zipf distribution [44] to understand the effectiveness of hot-key agnostic prioritization. In distributed training, we use popular models (ResNet50/101/152 and VGG11/16/19) with ImageNet [36, 63].

Metrics. We measure following metrics to compare different solutions' performance and overhead: (1) Job Completion time (JCT), a job's (multiple aggregation tasks) total execution time; (2) throughput/goodput of each host; (3) the training throughput (image/second) of image classification tasks in distributed training, and (4) CPU utilization.

5.2 In-Network Aggregation Benchmark

5.2.1 Computation Offload. Like other INA solutions, ASK offloads computation from hosts to the switch, which can reduce CPU overhead significantly while speeding up the performance. We show the computation offload in a MapReduce [30] job by comparing ASK with the host-only solution "PreAggr". We use only one sending host whose bandwidth equals the receiver's, excluding the network bottleneck's impact. In this experiment, we start the same number of *map threads (mapper)* and *reduce threads (reducer)* on the sending host and receiving host, respectively. Among them, the map thread is used to generate key-value streams, and the reduce thread is used to aggregate key-value tuples. In all experiments, the total data volume (key-value tuples) is fixed and follows a uniform distribution. The number of mapper/reducer threads is tunable.

Figure 7 shows that ASK consistently outperforms PreAggr in terms of JCT but consumes much fewer CPU cycles. In PreAggr, mappers' local aggregation reduces data volume significantly, from 51.2GB raw data to 256MB intermediate results, and the network transmission time is negligible. ASK achieves a JCT of about 16 seconds with 1 data channel, and a minimum JCT of about 6 seconds with 4 data channels; PreAggr spends 111.20s/33.22s with 8/32 threads. Because ASK consumes CPU only for packet IO (1.78%/3.57%/7.14% CPU for 1/2/4 data channels) but PreAggr consumes CPU for both computation and IO (14.3% for 8 threads, and 100% at the peak for 56 threads).

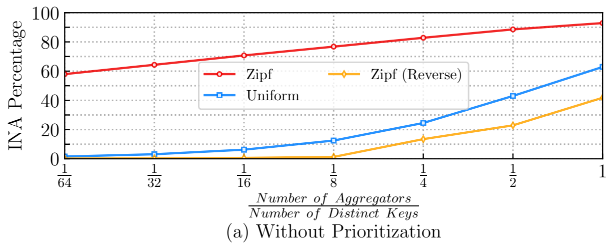
Table 1: Traffic reduction on different datasets. The traffic reduction is defined as $\frac{aggregated\ tuples}{incoming\ tuples}$ (first line) and $\frac{aggregated\ packets}{total\ packets}$ (second line), respectively.

Dataset	yelp	NG	BAC	LMDB
Aggregated key-value tuples (%)	92.18	85.73	94.32	91.49
Switch ACKed Packets (%)	72.01	84.35	90.36	88.59

5.2.2 Traffic Reduction in Real-World Traces. In data-intensive scenarios such as big data or distributed training, a large amount of traffic will put a huge burden on the network and affect the performance of other tasks. Reducing network traffic is crucial to alleviating network congestion and improving application performance. ASK can significantly reduce network traffic by aggregating traffic on TOR and actively discards the aggregated packets to prevent them from entering the network further and causing congestion. We repeat the experiment above with production datasets and count the ratio of key-value tuples/packets aggregated by the switch. As shown in Table 1, the switch can aggregate 85.73% ∼ 94.32% key-value tuples and absorb 72.01% ∼ 90.36% network traffic.

5.3 Effectiveness of Multi-key Vectorization

The multi-key design can effectively improve the goodput. Assuming one packet contains x 8-byte key-value tuples and the overhead



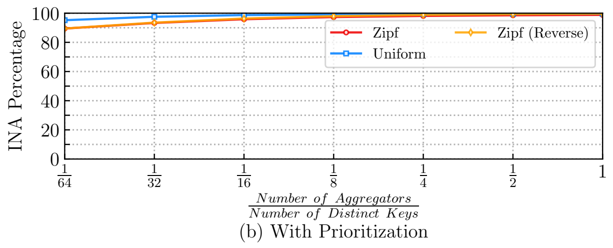


Figure 9: Key-value tuples aggregated by the switch with/without agnostic prioritization, varying with the ratio of the total number of aggregators to the number of distinct keys in the aggregation task.

of sending a packet is 78 bytes[9]. In the 100Gbps network, the ideal goodput will be $\frac{8x}{8x+78} \times 100Gbps$. We conduct data transfer experiments between two servers and vary the number of *key-value tuples per packet* from 1 to 64, then measure the actual goodput. Figure 8(a) compares the results of ASK with the theoretical ideal goodput. When the key-value tuples per packet do not exceed 32, the goodput increases almost linearly with the packet size. In this range, ASK's throughput is bounded by the PPS on the host. The small glitches (at 18 and 26 on the X-axis) out of the linearity are caused by the overhead of transferring a packet from the memory to the NIC via PCIe[10]. When the tuples per packet exceed 32, the experiment result matches the theoretical value.

ASK's key space partition to construct multi-key packets could cause some tuple slots in the packet to be blank when packing keys in a key-skewed dataset. Figure 8(b) measures the cumulative distribution of the number of non-blank (valid) key-value tuples contained in packets constructed from different datasets. Ideally, when the key distribution is uniform (line *Uniform*), there is no blank tuple in almost every packet. Real-world traces show a bit worse efficiency, but the worst traces (yelp [19]) still contains average 16.91 valid key-value tuples per packet, better than previous works [41, 47, 64] which only support one key per packet.

5.4 Effectiveness of Key Agnostic Prioritization

We show that the key agnostic prioritization in ASK can improve aggregator utilization, i.e., aggregating more hot keys with fewer aggregators. We generate two datasets from uniform distribution and Zipf distribution (§3.4) [44], respectively. The Zipf distribution has a skewed key distribution, which holds for all natural languages [57] and even artificial systems [59]. In the experiment, the *Zipf dataset* means that hot keys appear in the front and the cold keys appear in the rear in the key-value stream; *Zipf (reverse) dataset* reverses the key appearance order, making cold keys in the front and hot keys in the rear; in *Uniform dataset*, all keys have the same frequency (no hot and cold keys). We fix the number of distinct keys to 2^{16} (each dataset contains about 10^8 keys), and vary the number of aggregators from 2^4 to 2^{16}.

Figure 9(a) shows that switch aggregators are underutilized without key-agnostic prioritization. Because an aggregator could be occupied by a cold key (never appearing in the future), it will not be released until the end of the aggregation task. Increasing the number of aggregators, allowing more keys to be held in the switch, could increase the switch aggregation ratio. Making the hot keys appear early and occupy the aggregator, could also increase that ratio — ASK performs better on *Zipf* than *Zipf (reverse)*. Both methods do not always apply — for the former, the switch momory could be scarce and limited; for the latter, key-value streams could be unforeseeable without being sortable by frequency ahead of sending.

Figure 9(b) shows that key-agnostic prioritization significantly improves the aggregator utilization, avoiding a cold key occupying the aggregator for the entire task. We can use much fewer aggregators than distinct keys to complete the aggregation of almost all key-value tuples, e.g., the aggregator-to-distinct-key ratio of 1/16, achieving 95.85% on-switch aggregation.

[9] 78 = 12 (Inter-Packet Gap) + 7 (Preamble) + 1 (Start Frame Delimiter) + 14 (Ethernet Header) + 20 (IP Header) + 20 (ASK Header) + 4 (CRC).

[10] The Transaction Layer Packet (TLP [13]) transferred from the memory to NIC needs to start from PCIe lane$_0$ (16 lanes in total) and at an even cycle of the CPU clock, and each TLP has at least 24 bytes overhead on the PCIe.

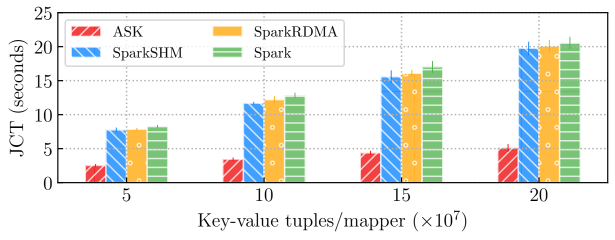
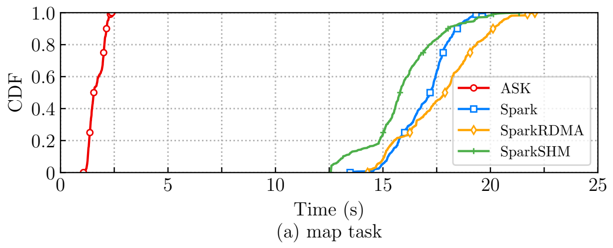


Figure 10: A comparison of ASK and Spark in terms of job completion time.

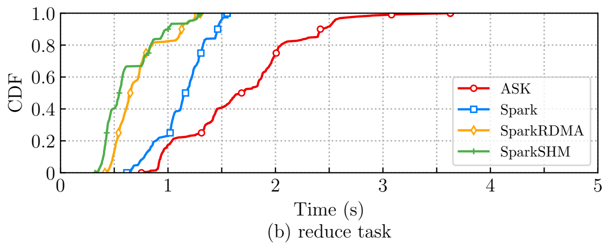
5.5 Effectiveness in Data Analytic Systems

We measure the ASK's synthetic performance acceleration to the Big Data system. We run WordCount in HiBench's SparkBench [7]. In the experiments, we set up 3 machines, each with 32 mappers (a map task in Spark [68]) and 32 reducers (a reduce task in Spark); each mapper has 2^{18} distinct keys. We randomly generate 5×10^7, 10×10^7, 15×10^7 and 20×10^7 key-value tuples per mapper. The baselines are Spark, SparkSHM, and SparkRDMA (§5.1 baselines). Figure 10 shows the results, and we get the following observations.

First, SparkRDMA and SparkSHM do not provide significant performance gain to Spark. Because after pre-aggregation in mappers, the intermediate results' volume is very small. Thus, improving the network throughput and disk I/O cannot obviously improve the overall JCT for aggregation jobs.



(a) map task



(b) reduce task

Figure 11: A comparison of ASK and Spark in terms of task completion time.

Second, ASK outperforms all other baselines in terms of JCT. Its JCT can be reduced by 67.3% to 75.1% compared with other baselines in all settings. The performance gain is from the computation offload. The aggregation is performed on the switch at the line rate instead of the CPU. Figure 11 shows the task completion time (TCT) of mappers and reducers, further validating the reason for the performance gain. In Spark with ASK, the mappers' TCT is significantly shorter than other baselines (mean 1.67s v.s. 15.89s-17.67s in the other three), because ASK's mappers do not use CPU for aggregation. ASK reducers have a longer TCT because some mappers are co-located with the reducer on the same machine, and these mappers' data needs to be aggregated by the local reducers. The mapper TCT decrement is more significant than the reducer TCT increment, so the overall JCT is reduced.

5.6 Extend to Deep Learning Systems

ASK can also cover the special case of value stream aggregation and be compatible with distributed training. We implement a parameter server system for distributed training by integrating ASK with BytePS [39]. We compare ASK with existing INA-based distributed training frameworks ATP [47] and SwitchML [61] on model training, and measure the training speed (image/second).

Figure 12 shows that ASK, ATP, and SwitchML have similar performance because they all use the switch to accelerate the gradient aggregation process. ASK and ATP slightly outperform SwitchML on some models because SwitchML's small packet size cannot fully utilize the network bandwidth.

5.7 Overhead and Scalability

We compare ASK with pure network transmission to study its bandwidth overhead and analyze the tradeoff between overhead, efficiency, and scalability. Compared with pure network transmission, ASK packets introduce overhead to bandwidth efficiency, and we

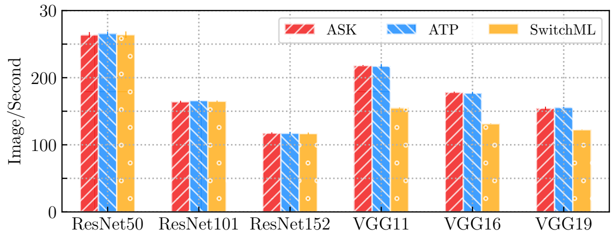


Figure 12: Single job throughput in distributed training.

argue that the overhead is acceptable compared with the significant computation acceleration and excellent scalability.

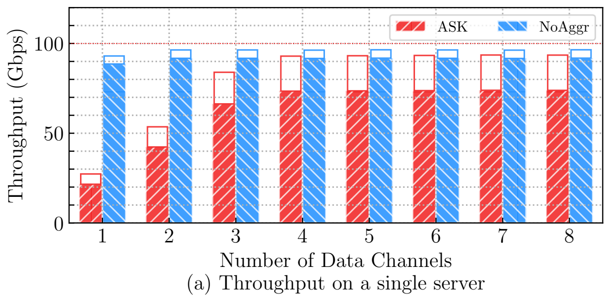
5.7.1 Bandwidth Overhead. With one pipeline, the hardware limitation restricts the number of AAs to be 32 and the packet payload to be 256Bytes. Figure 13(a) shows the aggregation throughput when there are only one sending host and one receiving host. "NoAggr" transmits packets with DPDK and 1500 bytes MTU. We tune the number of data channels. Overall, both ASK and NoAggr can saturate the NIC bandwidth, but the goodput of NoAggr and ASK is 91.75Gbps v.s. 73.96Gbps; and NoAggr saturates the bandwidth with 2 cores while ASK with 4 ones.

5.7.2 Scalability. ASK's processing speed could linearly scale with the number of senders, which significantly outperforms host-only solutions. We use one host as the receiver, tune the number of sending hosts, and show the average sender throughput in Figure 13(b). ASK's average throughput stays constant even with more servers because most of the traffic is directly aggregated and acknowledged by the switch, eliminating the bottleneck at the receiving host. But the average throughput in NoAggr is inversely proportional to the number of sending hosts (e.g., 11.88Gbps for 8 servers), where the receiving host's link becomes the bottleneck.

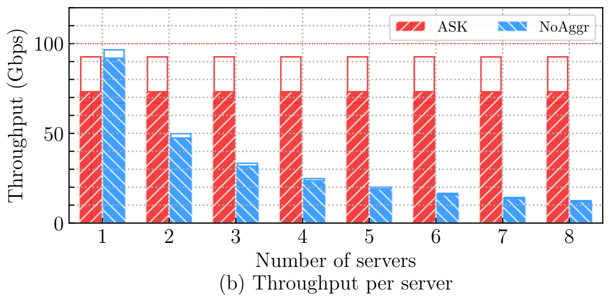
We argue that ASK's bandwidth overhead is acceptable considering its benefits. (1) The CPU cycles saved by computation offload are much larger than the ones cost in sending small packets (see Figure 3); (2) ASK shows excellent scalability, which is critical for distributed systems. (3) If the switch can spare more port bandwidth to chain pipelines and recirculate packets, the goodput can be further promoted (e.g., 4 pipelines achieving ~90Gbps/host).

6 RELATED WORK

INA has been deeply explored in distributed machine learning. Under some circumstances, the network would be the bottleneck in communication-intensive models [54]. ATP [47], SwitchML [61], SHARP [33], NetReduce [52], iSwitch [51], NVIDIA's accelerator centric network [43], and PANAMA [32] propose to apply INA to accelerate the gradient aggregation in distributed training. Flare [28] proposes a RISC-V-based switch module to aggregate vectors. The INA solutions above target value stream aggregation. Some other works deploy middleboxes [56] or high-performance dedicated servers [27, 50] other than switches to achieve in-network aggregation in specific scenarios, such as wireless communication and MapReduce. ASK is the first *on-switch, generic, vectorized, reliable,*



(a) Throughput on a single server



(b) Throughput per server

Figure 13: Aggregation throughput. The filled bar represents the goodput, and the empty bar represents bandwidth overhead consumed by the packet header, crc, etc.

and hot-key prioritized key-value aggregation service for diverse applications.

Key-value stream aggregation can also be accelerated by speeding up the network transmission, e.g., using the high-speed network (SparkRDMA [11]), or compressing traffic (OmniReduce [31]). ASK is complementary with these methods. Programmable switches can also accelerate operations other than aggregation, e.g., storage (NetCache [41] and DistCache [53]), replication (NetChain [40], HyperLoop [42] and Harmonia [70]), load balancing (AppSwitch [26]), and filter (Cheetah [64], FPISA [67], and NetAccel [48]), and ASK can work together with these operators in system building.

Trio [66] is a new type of programmable switch that adopts the run-to-completion architecture instead of the pipeline. Trio increases the memory available to the data plane of the programmable switch from $O(10MB)$ to $O(1GB)$ while reducing restrictions on memory access and increasing programmability at the cost of processing speed. The design of ASK can be very well adapted to this architecture. With Trio, the shadow copy mechanism and variable-length key processing of ASK can be further improved to support more jobs.

7 DISCUSSION

Deployment in Mutli-rack networks. When ASK is extended to the hierarchical aggregation, the senders are leaf nodes, the receiver is the root, and switches are the intermediate nodes. However, each switch must maintain the states for all data channels of its leaf nodes, where states could explode. To avoid state explosion, ASK could be deployed on TOR switches, providing a best-effort service only to hosts within one rack. And cross-rack traffic would bypass the receiver TOR switch and proceed to the receiver host for eventual aggregation.

Congestion Control. When multiple jobs coexist in the cluster and contend for bandwidth, a congestion control mechanism is needed for the jobs to share and saturate the bandwidth. ASK is compatible with existing ECN-based and loss-based INA congestion control mechanisms, e.g., ATP [47] and PANAMA [32]. When applying a congestion control mechanism, the congestion window should not exceed the maximum window defined in the reliability mechanism (§3.3), protecting the switch receive window from malfunctioning.

Multi-Tenancy. ASK supports multi-tenancy. When there are aggregation tasks from multiple tenants, these tasks need to encode the tenant ID into the task ID. Then the ASK daemon would isolate these tasks on the host, and ASK switch controller would isolate these tasks' memory regions in the switch.

Whether there is an alternative design of the Shadow Copy Mechanism. The shadow copy mechanism aims to process more hot keys in the limited switch memory. A seemingly obvious approach is to manage the AAs as set associative with a replacement policy such as LRU. However, this approach cannot be simply implemented on the programmable switch. In an "unreliable" network, the action of "evicting cold items to the receiver" (making space for hot items) requires the switch to make Active Repeat Request (ARQ) until the eviction succeeds (identified by a receiver-to-switch acknowledgement), but the switch programming language does not natively support repeat requests, and it is not practical to suspend packet processing for the trial-and-error eviction operation.

Actually, the two copies in the ASK shadow copy mechanism form an AA set as mentioned in the approach above. ASK makes the receiver periodically initiate the "eviction and replacement", which is triggered by the statistics on the receiver. And implementing ARQ (i.e., reliable `Read()` in §3.4) for eviction is more feasible on the receiver host than the switch.

8 CONCLUSION

In-network computing provides a novel architecture for improving the performance of distributed systems. Although in-network computing has demonstrated its potential in distributed training, there is still a lack of sound system design to support a broader range of aggregation jobs. ASK is the first switch-host co-designed system that provides key-value stream aggregation service to diverse applications simultaneously, which can accelerate applications' performance by reducing traffic volume and offloading computation. ASK overcomes challenges of vectorizing multi-key by key addressing and placement, correctness guarantee by a lightweight reliability mechanism, and utilizing switch memory to a better extent by hot-key agnostic prioritization. The evaluation shows that ASK could significantly accelerate key-value stream aggregation and applications such as big data and distributed training.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the valuable comments. Wenfei Wu is supported by Peking University Start-up funding. Ming Liu is supported in part by NSF grants CNS-2106199 and CNS-2212192.

REFERENCES

[1] 2011. Large Movie Review Dataset. https://ai.stanford.edu/~amaas/data/sentiment/.

[2] 2020. 20 Newsgroups. http://qwone.com/~jason/20Newsgroups/.

[3] 2020. DPDK (Data Plane Development Kit). http://dpdk.org.

[4] 2021. Intel$^{@}$ Tofino^{TM} 3. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[5] 2021. Intel$^{@}$ Tofino^{TM} Series of P4-Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[6] 2022. BytePS. https://github.com/bytedance/byteps.

[7] 2022. HiBench. https://github.com/Intel-bigdata/HiBench.git.

[8] 2022. Intel FlexPipe. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[9] 2022. IP Packet Overhead. https://infohub.delltechnologies.com/l/powerscale-network-design-considerations/ip-packet-overhead.

[10] 2022. Mellanox ConnectX-5. http://www.mellanox.com/related-docs/user_manuals/ConnectX-5_VPI_Card.pdf.

[11] 2022. Mellanox SparkRDMA. https://github.com/Mellanox/SparkRDMA.git.

[12] 2022. MPI Forum. https://www.mpi-forum.org.

[13] 2022. PCI Express. https://en.wikipedia.org/wiki/PCI_Express.

[14] 2022. Spark. https://spark.apache.org.

[15] 2022. Spark Stream. https://spark.apache.org/streaming/.

[16] 2022. The Blog Authorship Corpus. http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm.

[17] 2022. TPC-H Benchmark. http://www.tpc.org/tpch/.

[18] 2022. XPliant Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[19] 2022. Yelp Open Dataset. https://www.yelp.com/dataset.

[20] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM* (New Delhi, India).

[21] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling {ECN} in multi-service multi-queue data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 537–549.

[22] Pat Bosshart et al. 2014. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014).

[23] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[24] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[25] Ge Chen, Gaoxiong Zeng, and Li Chen. 2021. P4COM: In-Network Computation with Programmable Switches. *arXiv preprint arXiv:2107.13694* (2021).

[26] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. 2017. AppSwitch: Application-Layer Load Balancing within a Software Switch. In *Proceedings of the First Asia-Pacific Workshop on Networking* (Hong Kong, China) *(APNet'17)*. Association for Computing Machinery, New York, NY, USA, 64–70. https://doi.org/10.1145/3106989.3106998

[27] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. 2012. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 29–42. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/costa

[28] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: Flexible In-Network Allreduce. *arXiv preprint arXiv:2106.15565* (2021).

[29] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 10.

[30] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[31] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 676–691.

[32] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. 2021. In-network Aggregation for Shared Machine Learning Clusters. *Proceedings of Machine Learning and Systems* 3 (2021), 829–844.

[33] Richard L Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. 2020. Scalable hierarchical aggregation and reduction protocol (sharp) tm streaming-aggregation hardware design and evaluation. In *International Conference on High Performance Computing*. Springer, 41–59.

[34] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *SIGCOMM*. ACM, New York, NY, USA. https://doi.org/10.1145/2934872.2934908

[35] Moritz Hardt, Ben Recht, and Yoram Singer. 2016. Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning*. PMLR, 1225–1234.

[36] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[37] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*. IEEE, 41–51.

[38] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association.

[39] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. 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)*. USENIX Association.

[40] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[42] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 297–312. https://doi.org/10.1145/3230543.3230572

[43] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 996–1009.

[44] Clyde Kluckhohn. 1950. Human behavior and the principle of least effort.

[45] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[46] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. 2019. A large-scale study of MPI usage in open-source HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[47] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 741–761. https://www.usenix.org/conference/nsdi21/presentation/lao

[48] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. 2019. The Case for Network Accelerated Query Processing. In *CIDR*.

[49] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. SocksDirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*. 90–103.

[50] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[51] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating distributed reinforcement learning with in-switch computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 279–291. https://ieeexplore.ieee.org/abstract/document/8980345.

[52] Shuo Liu, Qiaoling Wang, Junyi Zhang, Qinliang Lin, Yao Liu, Meng Xu, Ray CC Chueng, and Jianfei He. 2020. NetReduce: RDMA-Compatible In-Network Reduction for Distributed DNN Training Acceleration. *arXiv preprint arXiv:2009.09736* (2020).

[53] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA,

143–157. https://www.usenix.org/conference/fast19/presentation/liu

[54] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. Parameter Hub: A Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3267809.3267840

[55] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. http://www.aclweb.org/anthology/P11-1015

[56] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. 2014. NetAgg: Using Middleboxes for Application-Specific On-Path Aggregation in Data Centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 249–262. https://doi.org/10.1145/2674005.2674996

[57] Bill Z Manaris, Luca Pellicoro, George Pothering, and Harland Hodges. 2006. Investigating Esperanto's Statistical Proportions Relative to other Languages using Neural Networks and Zipf's Law.. In *Artificial Intelligence and Applications*. 102–108.

[58] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3341301.3359642

[59] Steven T Piantadosi. 2014. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic bulletin & review* 21, 5 (2014), 1112–1130.

[60] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (Palo Alto, CA, USA) *(HotNets-XVI)*. Association for Computing Machinery, New York, NY, USA, 150–156. https://doi.org/10.1145/3152434.3152461

[61] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[62] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[63] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[64] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2407–2422.

[65] Sven Ulland. 2011. Kernel panic/crash, bnx2 flow control flooding and network outages. Linux-PowerEdge – Linux on Dell PowerEdge Servers discussion http://lists.us.dell.com/pipermail/linux-poweredge/2011-October/045485.html.

[66] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 633–648.

[67] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. 2022. Unlocking the Power of Inline Floating-Point Operations on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/nsdi22/presentation/yuan

[68] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[69] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. 2018. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[70] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 376–389. https://doi.org/10.14778/3368289.3368301

Received 2022-07-07; accepted 2022-09-22