# Preemptive Switch Memory Usage to Accelerate Training Jobs with Shared In-Network Aggregation

Hao Wang[1], Yuxuan Qin[1], ChonLam Lao[2], Yanfang Le[3], Wenfei Wu[4], Kai Chen[1]
[1]iSING Lab, Hong Kong University of Science and Technology
[2]Harvard University, [3]Intel, [4]Peking University

*Abstract*—**Recent works introduce In-Network Aggregation (INA) for distributed training (DT), which moves the gradient summation into network programmable switches. INA can reduce the traffic volume and accelerate communication in DT jobs. However, switch memory is a scarce resource, unable to support massive DT jobs in data centers, and existing INA solutions have not utilized switch memory to the best extent.**

**We propose DSA, an Efficient Data-Plane switch memory Scheduler for in-network Aggregation. DSA introduces preemption to the switch memory management for INA jobs. In the data plane, DSA allows gradient tensors with high priority to preempt the switch aggregators (basic computation unit in INA) from tensors with low priority, which avoids an aggregator wasting time in idle. In the control plane, DSA devises a priority policy which assigns high priority to gradient tensors that benefit overall job efficiency more, e.g., communication-intensive jobs. We prototype DSA and experiments show that DSA can improve the average JCT by up to $1.35\times$ compared with baseline solutions.**

## I. Introduction

To alleviate the communication bottleneck in large-scale distributed training (DT), e.g., large language model [16], researchers have recently proposed a new computation and communication primitive — *In-Network Aggregation (INA)* to accelerate the gradient aggregation in DT jobs. In INA, the switch performs an *AllReduce* operation to the gradient streams from multiple workers, and multicasts the aggregation results back to workers. By reducing the total traffic volume and freeing up network bandwidth, DSA could effectively accelerate DT jobs.

In INA, a job's gradient packets with the same *packet sequence number (PSN)* across workers are addressed to the same basic aggregation unit — *aggregator* — in the switch memory, where the aggregator sums the gradient packets up and returns the result back to workers along with the ACK packets. The switch memory (i.e., the aggregators) can be assigned to DT jobs either in an isolated mode or in a shared mode (Section II-A).

However, a closer look at the switch memory usage mode in existing solutions reveals that the switch memory is not utilized to the best extent, and there exists potential to further improve its utilization as well as the overall job efficiency. First, the aggregators could be used in a finer granularity, and perform more aggregation operations. Existing solutions make an aggregator occupied by *all* packets with the same PSN across workers; however, workers may progress differently in gradient sending, and (possible) straggler workers would make the aggregator wait for their packets without actually performing aggregation.

Second, existing memory allocation policies (i.e., isolated memory or shared memory with First-Come-First-Serve) do not give preference to "important" tensors, i.e., multidimensional arrays of numerical data, which could benefit the overall job efficiency more if given more INA resources. For example, a job with a shorter remaining time or a higher communication-to-computation ratio should be given priority over other jobs to finish first.

To this end, we propose a system with Data-plane Switch memory scheduling for in-network Aggregation, namely DSA, which introduces *preemption* to switch memory allocation. DSA can improve switch memory utilization as well as multi-job efficiency in INA. In the data plane, DSA's *preemption mechanism* allows an aggregator to process a job's partial packets and yield to another job with a higher priority, which enables finer-grained aggregator usage. In the control plane, DSA's *preemption policy* assigns a tensor priority according to its contribution to overall job efficiency, which helps "important" tensors to win the switch aggregator preemption.

DSA overcomes three challenges. First, most existing INA solutions have exhausted switch pipeline resources (e.g., stages, ALUs), and it is challenging to add extra preemption logic to the packet processing without impacting existing logic and the consequent goodput. DSA adds an auxiliary server, called PS, into the architecture to assist the switch in aggregation and handle various failure cases, without affecting the existing gradient aggregation on the switch pipeline (Section III).

Second, preemption complicates the correctness guarantee, causing difficulty in protocol design. Preemption could lead to one aggregation split on two spots (the switch and the PS), together with various packet loss possibilities, the protocol could easily meet with issues of protocol suspension, duplicate computation, and/or memory leak. DSA devises a *reminder* mechanism on the PS, which cooperates with workers' reliability mechanism and the switch deduplication to achieve *all-case* correctness (Section IV).

Third, preemption should be carefully tuned. Low-frequency preemption would cause aggregators to spend more time idle, and high-frequency preemption would cause aggregators to spend time thrashing instead of aggregating. To avoid both cases, DSA designs a *priority downgrading* method for the preemption policy (Section V).

We prototype DSA on a Tofino programmable switch and in the end-host network stack. We evaluate DSA on a testbed with one Wedge100 programmable switch and 10 NVIDIA V100 GPUs, and an NS3 [26] simulator with a 64-node topology. Experiments show that DSA outperforms state-of-the-art solutions — SwitchML [46] and ATP [34] — by up to $1.89\times$ and $1.35\times$, respectively. We also validate and confirm the effectiveness of the two design intuitions, i.e., preemptive mechanism and preemption policy, individually.

## II. Background

### A. Analysis of Memory Usage Modes in INA Solutions

**INA Preliminaries.** A model training algorithm takes an initial model and a dataset as input, and iteratively computes a gradient based on the model and the dataset and updates the model until the model converges. In distributed training where the model and the dataset can be partitioned or replicated on multiple workers, an important operation is to sum up tensors (model or gradient) among workers and return the result back to the workers, which is called *AllReduce*. In distributed training, the communication overhead could bottleneck the whole system [39], [46], and INA is an effective approach to accelerate AllReduce as well as the entire job [34], [20].

INA performs AllReduce on recent popular programmable switches. The programmable switch has a piece of on-chip SRAM which can store durable cross-packet states, and can load a user-specified program to operate on the states and packets. Thus, it provides an opportunity to aggregate multiple packets as one, and return the result back to the network.

In INA solutions, the switch memory is organized as an array of *aggregators*. All workers would perform an AllReduce operation on their gradients. Each worker chunks its gradient into a sequence of packets, and assigns each packet a *packet sequence number (PSN)*. Workers send their gradient packets to the switch, and packets with the same PSN are *addressed* to the same aggregator, and the aggregator aggregates the gradient packets to an aggregation result. On completion of aggregating all workers' gradient packets, the switch sends an ACK packet to downstream devices (workers or the PS), where the ACK packet piggybacks the aggregation result.

With the switch aggregating gradient packet streams as one, the network carries the aggregation results instead of the raw gradient packets. The total traffic volume is reduced, and thus, the AllReduce operation, as well as the whole job, is accelerated. Experiment results also show that INA can achieve up to $1.8\times$ speedup, compared to the current best practice, Ring All-reduce [46].

**Synchronous and Statistical INA.** We classify existing INA solutions into two classes — synchronous INA and statistical INA. In synchronous INA, the switch aggregator array is divided into isolated regions, and each DT job reserves a region for its lifetime. Within the region, a packet with $PSN$ is addressed to an aggregator with the modulo operation:

$$aggregator.index \leftarrow PSN\%Size + Offset.$$

SwitchML, PANAMA, and NetReduce[46], [18], [38] are in this class.

In statistical INA, the switch aggregator array is a shared pool among all jobs. All jobs reserve aggregators in a decentralized manner. A packet with $PSN$ is addressed to

$$aggregator.index \leftarrow Hash(JobID, PSN).$$

From the perspective of an aggregator, it serves job packets with First-Come-First-Serve (FCFS). The aggregator would detect addressing collisions (due to hash collision), and pass the late-arrived packet to a *fallback server*. The fallback server complements the aggregation for addressing failed packets. ATP [34] is in this class, and INAlloc [59] is in a hybrid mode of the two classes.

**Comparison.** Statistical INA could make better use of the switch resource. Like the debate of constructing computer networks in circuit switching or packet switching in the 1970s, the latter's statistical time-division multiplexing improves the link bandwidth utilization. In statistical INA, the gradient packet reserves the aggregator, and the ACK packet releases the aggregator; an aggregator is occupied transiently within one RTT. On the contrary, synchronous INA reserves an aggregator region for a job's lifetime; as the training algorithm iteratively computes and transmits gradients, the switch aggregators are idle when the job is in the computation phase. When multiple jobs share the switch aggregator, statistical INA allows the aggregators to be multiplexed in a fine time granularity, and thus, acquire a higher resource utilization and a shorter average job completion time (JCT).

### B. Problem Statement

**Switch memory is not sufficient to support massive jobs.** Switch memory is a scarce resource. The on-chip SRAM has to be designed of a small size to maintain high packet processing speed, e.g., ~$10\,\mathrm{MB}$ for Tofino [1]. In addition, production networks spare a portion of the switch memory for network functions such as forwarding table [46], load balancing [42], firewall [8], etc. In contrast, typical ML models have a size of hundreds of megabytes, which far exceeds the limitation of the switch memory.

Present programmable switch memory cannot satisfy the requirement of massive jobs [54] in production networks. Theoretically, to support aggregating gradient packets at a line rate, the switch memory needed for a job equals the bandwidth-delay product [46], e.g., each job needs ~$1\,\mathrm{MB}$ switch memory under $100\,\mathrm{Gbps}$ bandwidth. Therefore INA can support at most ten jobs, which is insufficient for an industrial production environment. For example, the trace of a two-month workload from a GPU cluster with hundreds of machines in Microsoft [29] shows that there can be a total of 96260 jobs, i.e., about a thousand jobs every day, and half of them last more than hours.

**Existing solutions have not used the switch memory in the finest time granularity.** By the analysis above, statistical INA uses the switch memory transiently, from the first packet reserving the aggregator to the ACK packet releasing the

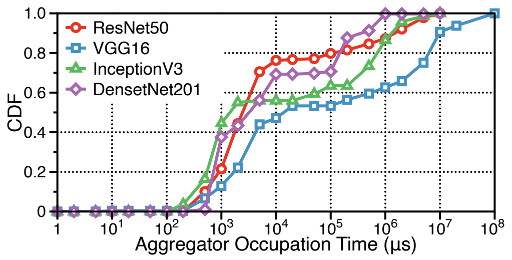


Figure 1: The aggregator occupation time, training four models in the cluster in §VII-A.

aggregator. However, switch aggregators still could experience idle time — the synchronization latency and the round-trip time between the switch and the PS.

A DT job involves multiple workers, and they may proceed with different progresses in training and gradient sending. First, the heterogeneous environments on workers may lead to their gradient generation not at the same time. Second, system overheads such as system calls and memory copy on different servers could fluctuate the gradient sending time and rate. Third, each worker could obtain different bandwidth in the network in a multi-tenant cloud environment.

Statistical INA has a fallback PS to handle addressing collisions, and packets need to experience a round trip between the switch and the PS. In failure cases, the fallback server is necessary for correctness guarantee; but in the normal case, the round trip for successfully aggregated packets unnecessarily contributes to extra aggregator occupation time.

Figure 1 shows the CDF of the aggregator occupation time when training four models. The median occupation time is 3-20 ms. In data centers, the typical RTT is less than 50 μs [58], which is 2-3 orders of magnitude smaller. Thus, the aggregator is severely underutilized. And the long occupation time is mainly from the worker synchronization delay, because the round trip between the switch and the PS is sub-RTT.

**Existing solutions have not assigned switch aggregators to more beneficial tensors.** Existing statistical INAs make the aggregators serve gradient packets in FCFS, implying that all workloads are of the same importance. Like TCP, FCFS together with the AIMD (additive-increase-multiplicative-decrease) congestion control on the host, all jobs would converge to resource sharing with max-min fairness.

However, in a multi-tenant scenario, certain workloads could benefit the overall multi-job efficiency more than others. First, within a model, sending and aggregating its front layers could benefit more to JCT, because the next-epoch training (forward propagation) could start immediately on the completion of the front layer transmission. Second, diverse models have different communication-to-computation ratios, and allocating the INA resource to communication-dominant models could reduce the JCT more significantly. Third, inspired by existing studies on job scheduling [22], [9], [17], prioritizing jobs with the shortest remaining time reduces the average JCT.

### C. *Intuition and Challenges*

**Intuition.** We propose to introduce *preemption* for switch memory management to improve its overall utilization. Each

| Stage | 0 | 1 | 2 | 3 | 4~10 | 11 |
|---|---|---|---|---|---|---|
| **Map RAM** | 13/48 | 22/48 | 0/48 | 11/48 | 44/48 | 33/48 |
| **Meter ALU** | 2/4 | 2/4 | 0/4 | 1/4 | 4/4 | 3/4 |
| **SRAM** | 16/80 | 22/80 | 0/80 | 18/80 | 48/80 | 37/80 |

Figure 2: Resource usage of ATP on Tofino [2], generated by P4i, a visualization tool for P4 programming [24]. This figure shows 3 mostly used types of resource (overall 20 types) on each of the 12 stages.

tensor is assigned a *priority* for preemption. In the data plane, enforcing a *preemption mechanism* could reduce aggregator idle time, making an aggregator perform more aggregation operations. In the control plane, applying a *preemption policy* could prioritize tensors that benefit more to the overall job efficiency. We build a system named DSA— Data-plane Switch memory scheduler with in-network Aggregation — with these intuitions.

**Challenges.** DSA overcomes three challenges. First, the programmable switch has limited programmability and computation resources, which causes difficulty in adding the new preemption function as well as its consequent failure handling. Existing solutions make in-depth optimization (e.g., resubmit and recirculate) to pack the switch functions into limited pipeline stages. Figure 2 shows that ATP almost uses up all meter-ALU resources of 7 stages (12 stages in all) on the switch pipeline. Adding DSA's new preemption function means to spare pipeline stages and reduce packet payload and the system goodput; such overhead could cancel out DSA's performance gain. DSA makes architectural design: placing the simple core aggregation function on the switch to maximumly use its line-rate processing capability, and attaching an auxiliary server, named PS, to handle all preemption and failure cases for correctness (Section III).

Second, preemption makes one aggregation operation possibly processed at two spots (the switch and the PS), which causes complex possibilities of protocol suspension, memory leak, duplicate computation, etc. DSA makes a synthetic design combining worker reliability, switch deduplication, and a new PS reminder mechanism, which cooperates to guarantee correctness. We also make an *all-case* correctness analysis (Section IV).

Third, the protocol needs to decide on a proper preemption frequency. When a preemption occurs, the switch sends the partial aggregation result in the PS. Too frequent preemptions will increase in-network traffic volume and impact the performance of INA; conservative preemption will cause aggregators with partial results to wait for straggler packets in idle time, hurting the aggregator utilization. DSA devises a preemption policy and priority downgrading method to tune the preemption frequency (Section V).

## III. DSA Design Overview

### A. *Architecture*

Figure 3 illustrates the architecture of DSA and an example. DSA is a host-switch codesigned transport layer protocol to support the AllReduce operation among a job's multiple workers. Each worker sends its gradient to the network and

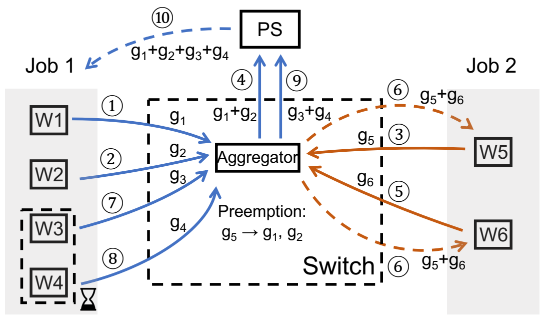


Figure 3: DSA architecture and an example of preemption. The solid arrows represent gradient packets, and the dotted arrows represent the ACK packets.

expects the sum of all workers' gradients from the network. On each worker, DSA deploys its own *network stack* to exchange gradient with the training application. In the network, DSA consists of *a switch and an auxiliary server* to perform the aggregation operation on workers' gradient streams. In the following text, the auxiliary server is called "*PS*".

The worker (its DSA network stack) is in charge of exchanging gradients and aggregation results with the training applications, computing the preemption priority, packetizing tensors and (de)packetizing. A worker chunks its gradient tensors into a sequence of packets, each with a *packet sequence number (PSN)*. All workers are initialized with the same PSN so that packets with the same PSN could be assigned to the same aggregator at the switch for aggregation. The worker would receive ACK packets from the switch or the PS, which not only acknowledges the receipt of the gradient packets but also piggybacks the aggregation result. The worker assembles the aggregation results in the ACK packets, and gives it back to the training application for model update.

The switch is in charge of aggregating gradient packets with the best effort and preemption. DSA organizes the switch memory as an *aggregator array*, where each aggregator has the format of `<JobID_PSN, bitmap, priority, tensor_values>`. Each aggregator can store the same length of tensor values as a gradient packet's payload. The aggregator array is shared among all jobs, and a gradient packet is addressed to an aggregator by hashing as in statistical INA.

The PS is in charge of handling failure cases, including aggregating fallback gradient packets and initiating active failure correction (i.e., the reminder mechanism in Section IV). The PS maintains a dictionary to record the intermediate results. The dictionary stores the mapping of

`JobID_PSN: <tensor_value, bitmap, time>`,

which handles packets from the switch.

## *B. Workflow*

**Initialization.** In system initialization, DSA allocates the aggregator array in the switch. When a DT job is started, it is assigned a global unique `JobID`. The job iteratively trains the model on multiple workers; in each iteration (a.k.a. an epoch), a worker computes a gradient, all workers perform an AllReduce operation to sum up their gradients and get the results, and each worker updates its model and proceeds to the next iteration. DSA involves in the AllReduce process.

**Worker sending gradient packets.** In an AllReduce, a worker chunks its gradient into a sequence of packets, each with its payload carrying the gradient values and a packet sequence number `PSN`. All workers are synchronized with the same initial PSN. The worker maintains a *sliding window* to stream the packets to the network; once it receives ACK packets, it advances the sliding window to send new packets. In DSA, the ACK packets piggyback the aggregation results from the switch or the PS.

Each worker in a job is assigned with a *Rank ID* to distinguish them, ranging from 1 to the number of workers $N$. The rank ID is also encoded into the packets: the DSA network stack computes a $N$-bit bitmap with one bit whose index equals the rank ID set 1 to indicate the worker.

Each job performs a *policy priority* calculation locally (detailed in Section V), and encodes the *priority* into the packet header. DSA would enforce aggregator enforcement among jobs in the switch according to their priority.

**Switch and PS aggregating packets "with preemption".** As a packet arrives at the switch, DSA addresses it to an aggregator by $Hash(JobID, PSN)$ in the array. If the aggregator is empty, the packet `tensor_values`, `JobID_PSN`, `bitmap` and the `priority` are written into the aggregator. If the aggregator is reserved and its `JobID_PSN` is the same with the packet, their bitmaps are further compared (i.e., `agtr.bitmap` ANDing `pkt.bitmap`): if the comparison returns 0, indicating that the packet arrives at the aggregator for the first time, the packet is accumulated to the aggregator; otherwise, the packet has arrived before, and it should not participate in the aggregation again. We name the process to check and avoid duplicate computation *deduplication*.

If a packet's `JobID_PSN` differs from its aggregator's, DSA applies preemption to improve switch memory utilization. The `priority` fields of the packet and the aggregator are compared, and the one with a higher priority is kept in the switch. If the aggregator wins, the packet is forwarded to the PS; if the packet wins, the packet and the aggregator's contents are swapped, and then the packet is forwarded to the PS.

The PS maintains a dictionary for each job. The dictionary records the fallback packets from the switch, including the packets that fail in preemption, and packets that were evicted. The dictionary uses `PSN` as the key, and `bitmap` and `tensor_values` as the value. The PS creates an entry in the dictionary for each new `PSN` and updates the value with the fallback packets.

On both the switch and PS, as they proceed with the packet aggregation, if the `bitmap` is full, indicating the aggregation is complete, an ACK packet is constructed and returned to workers. The ACK packet piggybacks the aggregation result; as it passes the switch, it is duplicated and multicast to all workers. On the switch, when the ACK packet is constructed, the aggregator is deallocated for future use; on the PS, the dictionary entry is also deleted after constructing the ACK packet.

The preemption mechanism could cause "partial aggregated" results in the aggregator, together with packet loss in an unreliable network, the whole transmission protocol could experience data missing, data duplication, and protocol suspension. We make a detailed analysis and design to handle these cases in Section IV.

**Worker receiving ACK packets.** The worker would receive ACK packets which carry the aggregation results. The ACK packets move the sliding window forward, and the worker continues to send new gradient packets in the window. The sending is terminated when all gradient packets are sent and acknowledged. As the worker receives ACK packets, it assembles the aggregation results as the aggregated gradient. At the end of the gradient sending, the assembled aggregated gradient is returned to the training application for the model update.

### C. Example

Figure 3 exemplifies the preemptive aggregator allocation. Job 1 has four workers W1, W2, W3, and W4, and Job 2 has two W5 and W6. $g_i$ denotes a packet from W$i$; $g_1$~$g_4$ have the same PSN, and $g_5$ and $g_6$ have the same PSN. Suppose W3 and W4 are the stragglers in the current epoch. Firstly, W1 and W2 send the gradient packets to switch (①②); the aggregator is empty and is reserved by Job 1, waiting for the gradient packets from W3 and W4. At this time, Job 2's workers W5 and W6 send gradients (suppose $g_5$ and $g_6$ have a higher priority than $g_1$, $g_2$, $g_3$, and $g_4$). When $g_5$ arrives (③), the switch performs preemption: the switch sends the partial aggregation result of W1 and W2, i.e., $g_1 + g_2$, to the PS and replaces the aggregator by $g_5$ (④). When $g_6$ arrives (⑤), the aggregation of Job 2 is completed, so the switch multicasts the ACK back to W5 and W6, carrying $g_5 + g_6$, and releases the aggregator (⑥). When $g_3$ and $g_4$ arrive (⑦⑧), the they reserve the aggregator again. A timeout at the PS would trigger the *reminder* mechanism to fetch the aggregator result to the PS (⑨). Finally, the PS adds up the two partial aggregation results of Job 1 ($g_1 + g_2 + g_3 + g_4$) and multicasts the ACK carrying the complete result to W1, W2, W3, and W4 (⑩).

### D. Discussion

**Reducing traffic volume.** Preemption is able to reduce the network traffic volume. Each aggregation operation in the switch reduces one packet in the network. Therefore, to reduce the traffic volume, we should increase the number of aggregation operations per unit time. Recall that under non-preemptive allocation, the aggregator will be idle when it is waiting for the straggler workers' gradient packets. During this period, other gradients colliding at the aggregator will fall back to the PS, which brings lots of network traffic. With preemption, we can reduce the times of fallback to the PS by introducing only several partial aggregation packets.

**Congenstion control and handling out-of-order packets.** When multiple jobs share the cluster, they may contend for network resources, including link capacity and switch memory. An ideal resource sharing should allow jobs to saturate the network. Like ATP [34], DSA applies on-switch ECN as the congestion signal, and Additive-Increase-Multiplicative-Decrease (AIMD) as the congestion control algorithm. Considering the $100\,\mathrm{Gbps}$ link capacity, DSA chooses $60\,\mathrm{KB}$ as the initial window size.

Unlike existing solutions, where the ACK packets are generated from a single spot (e.g., switch [46] or PS [34]), DSA's ACK packets can be generated from two spots — the switch or the PS. Thus, ACK packets may arrive at the worker out of order. In classical TCP, out-of-order packets would lead to duplicate ACK; the sender would regard duplicate ACK as a packet loss and link congestion, and thus, the sender would decrease its sending window (a.k.a., fast retransmission). In DSA, however, the out-of-order packets and duplicate ACK are normal, and is not a signal for congestion. Thus, DSA chooses to take no actions to change the sliding window size when observing out-of-order or duplicate ACKs.

## IV. Handling Failure Cases

### A. Failure Cases

The interaction between the three endpoints, as well as the non-ideal network conditions, could cause complex failure cases in packet aggregation. We name the aggregation operation on packets with the same PSN in a job an *aggregation task.*

**Packet Loss.** The network could be unreliable. If a gradient packet is lost before arriving at the switch/PS, the switch/PS could not complete the aggregation tasks.

**Duplicate Computation.** If one worker's gradient is lost, all workers cannot receive the ACK packet, and may retransmit their gradient packet. The switch would observe deduplicate appearance of a packet; the switch must not aggregate the packet again; otherwise, the aggregation result is incorrect.

**Protocol Suspension.** Assume the first few packets of task A are aggregated in a switch aggregator. Then another task B with a higher priority sends packets to the same aggregator, and evicts A's partial results. After task B completes and deallocates the aggregator, task A's remaining packets arrive and reserve the aggregator again. For task A, there are two partial aggregation results at the PS and the switch, waiting for each other; workers cannot receive ACK and advance the window, thus, the protocol suspends.

**Memory Leak.** If a worker retransmits a packet, and receives its (delayed) ACK before the retransmitted packet arrives at the switch. Then the retransmitted packet reserves an aggregator. If all workers receive the ACK and proceed to send new packets. The aggregator would be occupied without being released.

### B. Mechanisms

Except for the memory leak problem is solved by the "priority downgrading" method in Section V, all other issues are solved by a combination of the retransmission mechanism on workers, deduplication mechanism on the switch, and a *reminder* mechanism on the PS to handle all-case failures and ensure correctness. The essential intuition is that the PS can work as a data-plane coordinator to fetch data from all parties (the switch and workers) and correct the error cases.

DSA applies a *retransmission* mechanism on workers. After a gradient packet is sent out, as long as its ACK does not return for a *timeout*, the gradient packet is retransmitted. After three retransmissions, the worker sends a special *reminder* notification to the PS, which triggers the reminder mechanism on the PS.

DSA reuses the `bitmap` in each aggregator for deduplication. Recall that an aggregator maintains a bitmap to record the participation of each worker. When one worker's gradient packet appears for the first time, its bit in the bitmap is set 1; when the same packet appears again later, the switch recognizes the duplicate data, and would not accumulate the packet payload to the aggregator values. Thus, DSA can deduplicate repeated gradient packets.

DSA devises a *reminder* mechanism on the PS. In the mechanism, the PS sends requests of a PSN to the switch and all workers, fetching the (possibly partial) aggregation result on the switch, and the original gradient packets to the PS; the PS further computes the aggregation result and sends it as an ACK packet back to workers. The reminder mechanism can be triggered in two ways: if an entry in the PS dictionary experiences a timeout without completion, the PS actively runs the reminder mechanism for the entry's PSN; if a worker is stuck without receiving an ACK, the worker sends a reminder notification to the PS, which triggers the PS's reminder mechanism.

**Discussion.** Compared with the classical TCP, the reminder mechanism at the PS (receiver) is a new feature specific to INA. Unlike TCP where the switch just forwards packets without withholding them, INA protocols have to temporarily store the intermediate value in the switch "until all packets are aggregated", and such a waiting process would cause the protocol to suspend in progress. Making the PS (receiver) able to actively fetch intermediate results and the gradient packets helps the protocol to get rid of the suspension state and make progress.

## C. All-case Analysis

We consider three kinds of aggregation tasks: **type-I** tasks do not experience preemption, **type-II** tasks experience and win the preemption, and **type-III** ones experience but lose the preemption. We analyze the correctness in all three cases with the above mechanisms.

Type-I and type-II tasks are similar as they occupy the switch aggregator in their whole life. In the ideal case, all gradient packets are aggregated at the switch, and the ACK is generated and multicast to all workers. If one worker's gradient packet is lost, the worker (and other workers) would retransmit the gradient packet; the switch deduplicates packets and completes the aggregation, and sends the ACK back to workers.

If all workers' ACK packets are lost, all workers retransmit the gradient packets, redoing the aggregation and ACK reply.

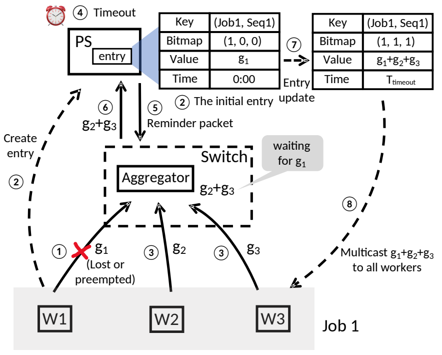


Figure 4: An example of the reminder mechanism. The solid arrows represent data flow, and the dotted arrows represent events.

If some workers' ACK packets are lost but some are not, the workers missing ACK packets would retransmit their gradient packets. Unfortunately, these retransmitted packets can never complete the aggregation (bitmap would never be full); the workers would experience a timeout and send a reminder notification to the PS, where the PS's reminder mechanism fetches all workers' gradient packets, completes the aggregation, and sends the ACK to all workers (including the failed ones).

Type-III tasks have the most complex possibilities. In the ideal case without packet loss, the task has a partial result at the PS (either the first few packets lose preemption or are evicted to the PS). If the later packets do not win the preemption, they are forwarded to the PS, and the aggregation is completed at the PS and the ACK is replied. If the later packets find that the aggregator is empty (deallocated by the collided tasks), these packets leave a second partial result in the switch; in this case, the task entry in the PS would experience a timeout and trigger the reminder mechanism to fetch the on-switch result to the PS, and the PS completes the aggregation and replies the ACK.

Packet loss could happen on any hops of the round-trip path: worker-switch, switch-PS, and PS-worker. Any packet loss would cause two possibilities: the PS has a partial result, and/or some workers do not receive ACK. In the former case, the PS initiates the reminder mechanism; in the latter case, the worker notifies the PS to start the reminder mechanism; with the reminder mechanism, the PS fetches the partial result from the switch and/or the gradient packets from workers, and complete the aggregation and replies the ACK.

## D. Example

Figure 4 illustrates the reminder mechanism procedures. We note the first gradient fragment of worker $i$ as $g_i$. $g_1$ arrives at the switch first; however, due to reasons like address collision and preemption failure, $g_1$ loses possession of the aggregator (①), and arrives at the PS. Then the PS will create an entry for this gradient packet (②). When $g_2$ and $g_3$ arrive at the switch later (③), they may occupy the aggregator and wait for $g_1$. However, $g_1$ is already at the PS, so the switch cannot complete the aggregation and release the aggregator by itself.

Table I: Definitions and notations

| | |
|---|---|
| $P_j(l)$ | the priority of the gradients in layer $l$ for Job $j$ |
| $l$ | layer of the gradients |
| $T_j$ | remaining time of the current epoch for Job $j$ |
| $L_j$ | number of layers in the model of Job $j$ |
| $Comm_j$ | communication time of one epoch of Job $j$ |
| $Comp_j$ | computation time of one epoch of Job $j$ |

By the reminder mechanism, the PS would detect a timeout for an entry (setting in §VI)(④), and send a reminder packet to the switch, which contains the job ID and PSN (⑤). The reminder packet fetches the aggregated result, i.e., $g_2 + g_3$, from the switch to the PS (⑥). Then the PS updates the corresponding entry (⑦). Finally, if the entry bitmap is full (all "1"s), the PS multicasts this aggregated result to all workers in an ACK packet (⑧).

## V. Preemption Policy

**Priority setting.** DSA's preemption policy tries to assign a high priority to tensors which could contribute more to the overall job efficiency and help them to win the preemption in the switch. A tensor's priority is assigned based on three heuristics[1].

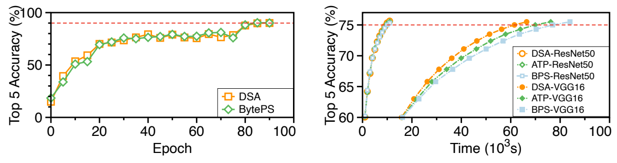
First, within one model, in the gradient computation (i.e., backward propagation), the gradient's front layers are generated lastly, but they would be used first in the next epoch. Thus, prioritizing the transmission of model front layers helps them to complete more quickly, and start the next epoch earlier. Second, different models have different communication-to-computation skewness in training, and prioritizing communication-intensive models could make transmission time reduction more significant. Third, based on classical scheduling policy studies [22], [9], [17], prioritizing jobs whose current epoch has the shortest remaining time could reduce the average JCT.

Using the notations in Table I, a tensor's priority can be computed as follows

$$P_j(l) = \frac{1}{T_j} \times \frac{L_j}{l} \times \frac{Comm_j}{Comp_j}. \tag{1}$$

The three factors are multiplied instead of linearly combined, since they have different units and ranges. In the runtime, $T_j$, $Comm_j$, and $Comp_j$ are measured with the model training.

**Priority downgrading.** To avoid an aggregator being occupied for a long time, DSA applies a priority downgrading method[2] at the switch. If an addressing collision happens and the packet fails to preempt, the aggregator's priority is halved, i.e., shifting one bit to the right. When a high-priority task reserves an aggregator for a long time, it has more chance to meet with collisions; each collision causes the task priority to downgrade, and eventually, the task yields the aggregator to a newer task.



(a) ETA (b) TTA

Figure 5: End-to-end DNN training quality. Batch size is 32.

## VI. Implementation

DSA host network stack is built on the Raw Ethernet Programming of Mellanox [3] with 4000+ lines of code (LoC) in C++. And the DSA switch logic has 3000+ LoC in P4 on an Intel/Barefoot Tofino switch. We elaborate on two mechanisms in the implementation.

**Packet swapping for aggregator replacement.** When a packet preempts the aggregator, it evicts the aggregator value to the PS. This process is implemented by a *packet swapping* method. On the Tofino switch, the aggregator array is declared as a switch register array, but the Tofino P4 language only allows one-time memory read and write by a single instruction `read_and_modify`. The instruction can swap the packet payload with the aggregator `tensor_values`. But the metadata fields, i.e., `priority` and `JobID_PSN`, need to be read first, checked to decide preemption, and then written, which does not fit in the instruction. DSA performs *resubmit* operation on the packet, putting it back to the beginning of the ingress pipeline, and thus these fields can be written in the second pass. Although resubmit consumes switch internal bandwidth, it only occurs in preemption.

**Settings of the reminder mechanism.** Both the worker and the PS are configured with timeout for failure handling. The timeout is updated by an Exponentially Weighted Moving Average (EWMA) in the runtime based on measurement (like that in TCP [31]). On the worker side, the timeout is based on the RTT measurement (the time between a gradient sending and its ACK arriving). On the PS, the timeout is based on the measurement of an entry setup and its aggregation completion. To avoid spurious reminders, DSA forces the minimum timeout to be $1\,\mathrm{ms}$.

## VII. Evaluation

We evaluate DSA on the testbed and by simulation. The key findings are as follows:

- DSA's acceleration does not hurt model accuracy in testbed experiments, and the convergence time is improved by $1.27\times/1.15\times$ compared to BytePS/ATP.
- DSA outperforms INA solutions (SwitchML and ATP) and vanilla BytePS in training speed in all cases on the testbed, including large model training such as BERT.
- DSA improves JCT by up to $1.89\times/1.35\times$ compared to SwitchML/ATP in large-scale simulation.
- Both the preemption mechanism and the policy take effect in accelerating INA; the former improves switch memory utilization by up to $2.27\times$, and the latter improves JCT by up to $1.16\times$ compared with the baselines.

[1]These heuristics provide a fundamental basis for priority setting in tensor transmission. However, they represent an initial attempt to address this complex problem. Future work could explore more sophisticated methods.

[2]Ideally, We can add a timer to each aggregator and set a timeout to avoid long occupations. However, the limited programming resource of P4 switches prevents it, instead, we apply the priority downgrading.

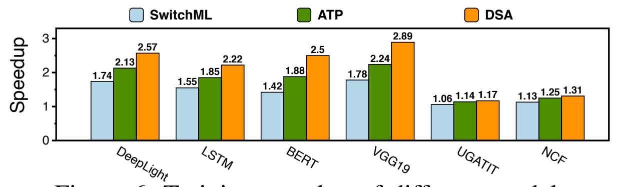
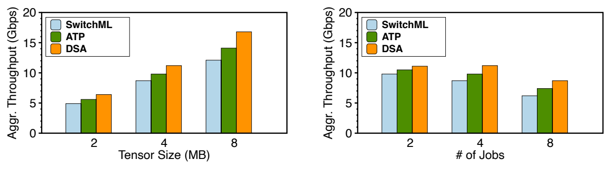


Figure 6: Training speedup of different models.



(a) Varying tensor size (b) Varying number of jobs

Figure 7: Aggregation throughout in AllReduce.

## A. *Testbed Experiments*

We integrate DSA into BytePS [30], a DNN training framework, and evaluate its end-to-end performance on a small-scale testbed. We also use microbenchmarks to evaluate the communication speedup.

*1) Settings:* **Testbed.** Our testbed consists of 5 physical GPU servers, each with 2 V100 GPUs, 40 CPU cores (Intel Xeon Gold 5115), 128 GB memory, 2 Mellanox ConnectX5 100 Gbps NICs, and one Edgecore Wedge100BF-32X switch. Each server has two 100 Gbps links connecting to the switch. To scale up our testbed, we further divide one physical server into two separated docker containers, each with 1 GPU, 20 CPU cores, 64 GB memory, and a 100 Gbps virtual NIC.

**Models and Workloads.** We train representative DNN models, including ResNet50 [25], and VGG16 [50] (tested in ATP [34], using Cifar100 dataset [33]), and DeepLight, LSTM, BERT, VGG19, UGATIT, and NCF (tested in SwitchML [46]). We also run an AllReduce benchmark to measure the aggregation throughput, where each worker repeatedly sends fixed-size tensors and receives the result.

**Baselines and Metrics.** We compare DSA against ATP, SwitchML, and vanilla BytePS (NtoN RDMA mode). To make a fair comparison, all solutions are integrated into BytePS [30], and all systems enable the NIC feature of TSO and MP-QP. The packet size of ATP and DSA is 306 B, and that of SwitchML is 180 B. When running multiple jobs with SwitchML, we isolate the switch memory evenly among jobs.

We use epoch-to-accuracy (ETA) and time-to-accuracy (TTA) to measure the training quality, and the aggregation throughput, i.e., worker receiving throughput, to measure the training speed.

*2) Performance:* **Training Quality.** In Figure 5(a), each experiment runs a single job with 8 workers training ResNet50. Considering the small scale of our testbed, we limit the switch memory (aggregators) to 1 MB. We observe that the curve of DSA is similar to BytePS's, and they converge to the same accuracy. Thus, DSA's acceleration does not affect the training accuracy.

In Figure 5(b), each experiment runs 2 DNN models, ResNet50 and VGG16, each with 4 workers. And DSA and ATP have one extra PS for each job. DSA reaches 75% top-5 accuracy $1.15\times$ and $1.27\times$ faster than the ATP and BytePS in VGG16 training. DSA only slightly outperforms the baselines ($< 1.01\times$) when training ResNet50, because ResNet50 is computation-intensive, which is consistent with the observations in ATP [34].

**Training Speed.** In Figure 6, each experiment runs two identical jobs concurrently, each with 5 workers. DSA outperforms SwitchML and ATP in all cases; specifically, when training BERT, a widely-used large model, DSA achieves a training speedup of 76% and 33% compared to SwitchML and ATP, respectively.

**Throughput Gain.** In AllReduce benchmarking, we configure two settings: the first is to fix the number of jobs to 4 and vary the tensor size, and the second is to fix the tensor size to 4 MB and vary the number of jobs. We use the first 8 containers $\{w_1 \sim w_8\}$ to be the workers and the remaining 2 to be the PSes $\{p_1, p_2\}$. All jobs with the odd ID are located on $\{w_1 \sim w_4,\ p_1\}$, and other jobs are on $\{w_5 \sim w_8,\ p_2\}$. Figure 7 shows the results. DSA outperforms SwitchML and ATP by up to $1.33\times$ and $1.76\times$, respectively. We also observe that all INA solutions' speedup is more significant with larger tensor sizes and fewer concurrent jobs.

## B. *Simulations on Large-scale Jobs and Networks*

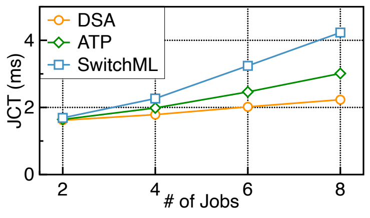
Considering the limited scale of our testbed, here we evaluate DSA at a large scale with NS3 [26] simulation as a supplement.

*1) Settings:* **Topology.** We set up a star topology, where one switch connects to 64 servers, each with a 100 Gbps link. The base RTT is 10 µs. The switch spares 5 MB memory for INA. We use the same packet size (306Byte) as the setting of ATP [34].

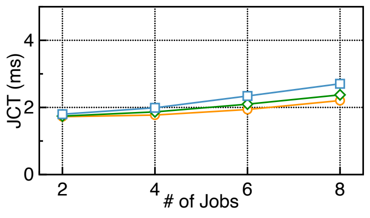
**Training Behavior in Simulation.** Our simulation considers the communication and computation overlapping [56] of DNN training. For simplicity, we assume that all DNNs have only two layers with the same size, and each layer is evenly divided into two tensor partitions [44]. Since the backward propagation (BP) starts to compute the gradients from the back layer, we assume that the first tensor partition of the second layer is transmitted first, followed by the first layer, and finally, the second tensor partition of the second layer. When a worker receives all aggregation results of the first layer, it can directly start the computation of the first layer in the next epoch. In contrast, the computation of the second layer must wait for the computation completion of the first layer and the arrival of the second layer's aggregation results.

**Workloads.** We assume two types of DNNs; the first one (A) is communication-intensive with a tensor partition size of 4 MB and a computation time of 0.32 ms per layer (theoretical communication time: computation time = 2:1). The second one (B) is computation-intensive with a tensor partition size of 2MB and a computation time of 0.64 ms per layer (theoretical communication time: computation time = 1:2).

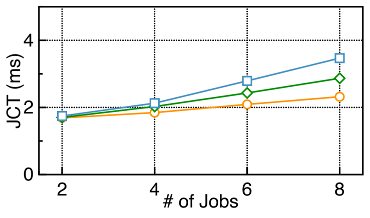
**Baselines and Metrics.** We still use ATP and SwitchML as the baselines. We measure job completion time (JCT) to evaluate



(a) DNN A

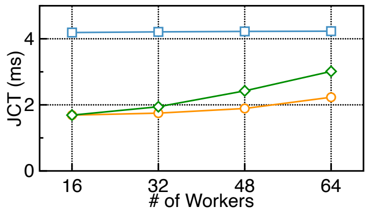


(b) DNN B

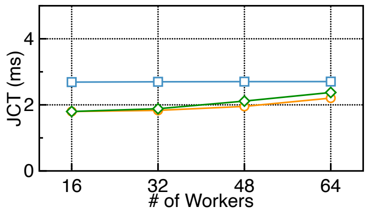


(c) A:B = 1:1

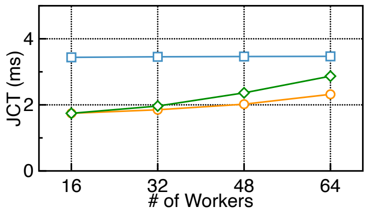
Figure 8: Average JCT with different numbers of jobs.



(a) DNN A



(b) DNN B



(c) A:B = 1:1

Figure 9: Average JCT with different numbers of workers.

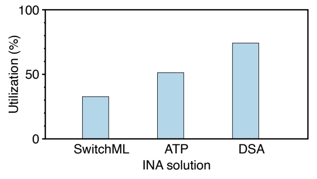
the system efficiency, where a job is defined as a one-epoch training — computing a gradient, AllReducing gradients, and updating the model.

**Parameter Settings.** 1) Job placement: we assume no overlapping usage of servers among different jobs, and each server contains at most one worker. 2) Job start time: to reflect the real situation, we avoid all DNN jobs starting at exactly the same time; here, we use a random variable $t$ as the job start time, where $t \sim U(0, 1\,\text{ms})$. 3) Computation speed variance: Considering the different computation speeds of different workers, we add a jitter $j$ on the sending side, where $j \sim U(0, 300\,\mu\text{s})$, which is approximate to the computation time of a tensor partition. 4) Parameter server: For ATP and DSA, a parameter server is required for each job. Without loss of generality, we add 8 extra servers as the PSes for the 8-job and 64-worker case. 5) Priority setting: we calculate the priority according to the formula in §V. Since our DNNs have two layers, all $L_j$ is 2. The $l$ can be 1 or 2 according to which layer. For DNN A, the $\frac{Comm_j}{Comp_j} = 2$, for DNN B, the $\frac{Comm_j}{Comp_j} = 0.5$. We use the remaining time of the current epoch to estimate $T_j$.

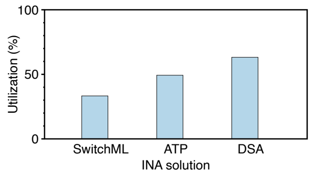
*2) Performance:* We run two groups of simulations. In the first group, we fix the number of workers in each job to be 8, and change the number of jobs. In the second group, we fix the number of jobs to be 8, and vary the number of workers in each job (each job has the same number of workers). We conduct three simulations in each group: all jobs are DNN A; all jobs are B; the ratio of A to B is 1:1.

**Speedup with different numbers of jobs.** Figure 8 shows the JCT with different numbers of jobs. DSA outperforms SwitchML and ATP by up to 1.89× and 1.35×. We also observe that the speedup of DSA becomes more significant with more jobs. This result is consistent with our testbed.

**Speedup with different numbers of workers.** Figure 9 shows the JCT for different INA solutions with different numbers of workers. DSA outperforms SwitchML and ATP in all

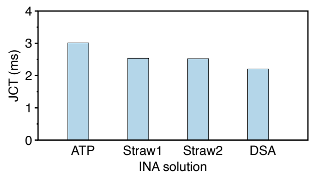


(a) DNN A

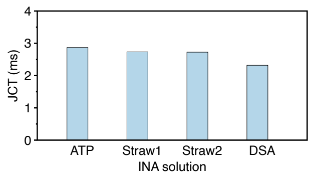


(b) DNN B

Figure 10: [Simulation] Switch memory utilization.



(a) DNN A



(b) A:B = 1:1

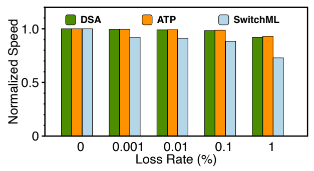
Figure 11: [Simulation] Speedup of preemption policy.

cases. DSA obtains more improvement over ATP with more workers. This result matches the expectation because with more workers, the synchronization cost of aggregation will increase. Thus preemption can obtain more performance gain.

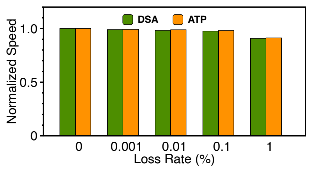
## *C. Validating Individual Methods*

**Preemption Mechanism.** We simulate 8 jobs, each with 8 workers, and other settings are the same with §VII-B. We define the switch memory utilization as the aggregation throughput over the link capacity $100\,\text{Gbps}$. In the multi-job scenario, we calculate the average utilization of all jobs. Figure 10 shows switch memory utilization measured on two types of DNNs. DSA outperforms SwitchML and ATP by 2.27× and 1.45× for DNN A. The numbers for DNN B are 1.9× and 1.28×. We observe more improvement on communication-intensive models, i.e., DNN A, which is consistent with the testbed experiment.

**Preemption Policy.** We make two strawman preemption INA solutions: the first one always does preemption upon hash collision; the second one has a 50-50 chance to perform preemption. We compare DSA with these two solutions and



(a) Loss at one worker



(b) Loss at PS

Figure 12: [Testbed] Training speed under different packet loss rates.

ATP by simulation. There are 8 jobs, each with 8 workers. Other settings are in §VII-B. Figure 11 shows the average JCT under two settings: 1) all jobs are DNN A; 2) DNN A and DNN B each have 4 jobs. DSA, Straw1, and Straw2 outperform ATP by 1.35×, 1.19×, and 1.19× for DNN A. For the mixed-models setting, the numbers are 1.22×, 1.05×, and 1.05×. DSA performs better than the strawman solutions and shows more improvement with a mixed-model setting. The results prove that DSA benefits from the preemption policy.

**Performance in Lossy Networks** We evaluate the loss recovery mechanism of DSA by artificially dropping packets. We choose ATP [34] and a modified SwitchML [46] (using the same timeout and congestion control mechanism of ATP) as the baselines. To reflect the combined effect of preemption and packet loss, we run two concurrent jobs, each with four workers. We train VGG16 on Cifar100. Figure 12(a) shows the impact of packet drop on one worker, and Figure 12(b) shows that on the PS (note that SwitchML does not use a PS, so we only compare DSA with ATP). We observe that DSA is resilient to packet loss and provides a training speed close to ATP. We also measure the ratio of each packet loss type, i.e, corresponding to which aggregation tasks (§IV-C). For packet loss at one work, the ratio is around 8:1:1, for the packet loss at the PS, it is 5:1:1. The observations are 1) The ratio of each aggregation task is independent of loss rate. 2) The ratios of **type-II** and **type-III** tasks are almost the same. 3) Packet loss at the PS brings more impact to the preemption tasks.

## VIII. Related Work

**Synchronous INA solutions.** Synchronous INA isolates switch memory for individual jobs. Since model training iteratively computes and transmits gradients, synchronous INA causes switch memory to be idle when the host is computing. DSA is in the scope of statistical INA, which avoids this disadvantage. SHARP [20], [19] supports layer-4 aggregation with ASIC in InfiniBand switches. SwitchML [46] builds single-job INA on the Tofino switch. DAIET [45] implements a proof-of-concept on the Tofino switch to show INA's performance gain instead of an end-to-end system. Camdoop [11], NetAgg [41], and MLFabric [51] propose to add a high-performance server as the midpoint to aggregate data streams. Parameter Hub [39] proposes to use a cluster of servers to replace the endpoint parameter server. iSwitch [36], PANAMA[18], and NetReduce [38] propose INA acceleration for DT jobs on FPGA. Flare [35] gives a RISC-V-specific INA design. Klenk et al. simulate an INA design on switches for NVIDIA intra-GPU NVLink network [32]. Faraj et al. mentioned AllReduce is performed by "fast math units" on the network in IBM Blue Gene/P supercomputer [14]. INA is also proposed in wireless networks [7], [48], but the solutions target the single-job scenario.

**Statistical INA solutions.** Statistical INA makes all jobs share the switch memory. Switch aggregators are reserved and released in a finer time granularity, which improves the switch memory utilization. DSA proposes a preemptive switch memory usage mode, which could further refine the aggregator usage granularity and improve the utilization. ATP [34] makes jobs share a pool of aggregators. INAlloc [59] allows a hybrid memory usage mode, which jointly considers job deadline guarantee and overall efficiency. GRID [13] considers the switch processing capacity limitation and distributes the aggregation workloads on the network-wide switches.

**Other DT acceleration solutions.** The following solutions also accelerate DT jobs. They either reduce the transmission time or allocate resources to jobs, and they are complementary with INA and DSA. General network acceleration [4], [55], [60], [6], [5] can speedup DNN communication. TicTac [23], Poseidon [56], P3 [28], and ByteScheduler [44] overlap the communication time with computation time in model training. OmniReduce [15], AdaGrad [47], and DGC [37] reduce traffic volume by methods like quantization and compression. GossipGraD [12] and Sip-ML [49] improve transmission rate by InfiniBand or optical networks. BlueConnect [10], Blink [52], AFS [27], and PLink [40] make topology-aware job management and allocate network bandwidth to jobs. Gandiva [53], Tiresias [21], SLAQ [57] schedule GPU resources to jobs. Optimus [43] considers job placement to balance GPU and bandwidth usage among jobs.

## IX. Conclusion

In this paper, we proposed DSA to accelerate DT jobs with shared INA in a cluster. DSA introduces preemption into the memory management, and improves the switch memory utilization as well as the overall job efficiency. In the data plane, DSA applies a preemption mechanism to use the switch aggregators in a finer time granularity, avoiding aggregator idle time. In the control plane, DSA devises a preemption policy to assign high priority to gradient tensors that benefit more to the overall job efficiency. In addition, DSA designs a reminder mechanism that handles various failure cases in the protocol. DSA prototype and experiment results show that DSA can outperform the state-of-the-art INAs (ATP and SwitchML) by up to 1.89× and 1.35× respectively in terms of average JCT.

## Acknowledgement

We thank the anonymous ICNP reviewers and our shepherd Prof. Jiaqi Zheng for their feedback and suggestions. This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20R, the GRF 16213621, and the ACCESS project. Wenfei Wu and Kai Chen are the corresponding authors.

## References

[1] Second-generation p4-programmable ethernet switch asic: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, 2021.

[2] The source code of atp: https://github.com/in-ATP/ATP, 2021.

[3] Raw ethernet programming: https://docs.nvidia.com/networking/display/MLNXOFEDv461000/Programming, 2023.

[4] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering azure storage with rdma. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[5] Wei Bai, Kai Chen, Shuihai Hu, Kun Tan, and Yongqiang Xiong. Congestion control for high-speed extremely shallow-buffered datacenter networks. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 29–35, 2017.

[6] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, 2015.

[7] Raghav Bhaskar, Ragesh Jaiswal, and Sidharth Telang. Congestion lower bounds for secure in-network aggregation. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 197–204, 2012.

[8] Jiamin Cao, Jun Bi, Yu Zhou, and Cheng Zhang. Cofilter: A high-performance switch-assisted stateful packet filter. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 9–11, 2018.

[9] Nathanaël Cheriere, Pierre Donat-Bouillud, Shadi Ibrahim, and Matthieu Simonin. On the usability of shortest remaining time first policy in shared hadoop clusters. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 426–431, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Minsik Cho, Ulrich Finkler, and David Kung. Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In *Proceedings of the 2nd SysML Conference*, 2019.

[11] Paolo Costa, Austin Donnelly, Antony IT Rowstron, and Greg O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *NSDI*, volume 12, pages 3–3, 2012.

[12] Jeff Daily, Abhinav Vishnu, Charles Siegel, Thomas Warfel, and Vinay Amatya. Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent. *arXiv preprint arXiv:1803.05880*, 2018.

[13] Jin Fang, Gongming Zhao, Hongli Xu, Changbo Wu, and Zhuolong Yu. Grid: Gradient routing with in-network aggregation for distributed training. *IEEE/ACM Transactions on Networking*, 2023.

[14] Ahmad Faraj, Sameer Kumar, Brian Smith, Amith Mamidala, John Gunnels, and Philip Heidelberger. Mpi collective communications on the blue gene/p supercomputer: Algorithms and optimizations. In *Proceedings of the 23rd international conference on Supercomputing*, pages 489–490, 2009.

[15] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 676–691, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694, 2020.

[17] Chengxi Gao, Victor C.S. Lee, and Keqin Li. D-srtf: Distributed shortest remaining time first scheduling for data center networks. *IEEE Transactions on Cloud Computing*, 9(2):562–575, 2021.

[18] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network aggregation for shared machine learning clusters. *Proceedings of Machine Learning and Systems*, 3:829–844, 2021.

[19] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10. IEEE, 2016.

[20] Richard L Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. Scalable hierarchical aggregation and reduction protocol (sharp) tm streaming-aggregation hardware design and evaluation. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*, pages 41–59. Springer, 2020.

[21] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[22] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2):207–233, may 2003.

[23] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems*, 1:418–430, 2019.

[24] Frederik Hauser, Marco Haberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *arXiv preprint arXiv:2101.10632*, 2021.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[26] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527, 2008.

[27] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739, 2021.

[28] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *Proceedings of Machine Learning and Systems*, 1:132–145, 2019.

[29] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[30] 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.

[31] Alex Kesselman and Yishay Mansour. Optimizing tcp retransmission timeout. In *International Conference on Networking*, pages 133–140. Springer, 2005.

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

[33] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

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

[35] Xiang Li, Fabing Li, and Mingyu Gao. Flare: A fast, secure, and memory-efficient distributed analytics framework. *Proceedings of the VLDB Endowment*, 16(6):1439–1452, 2023.

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

[37] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[38] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 376–391, 2023.

[39] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for

distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[40] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. *Proceedings of Machine Learning and Systems*, 2:82–97, 2020.

[41] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. 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*, pages 249–262, 2014.

[42] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[43] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In $Eurosys$ *18*.

[44] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[45] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156, 2017.

[46] 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.

[47] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth annual conference of the international speech communication association*, 2014.

[48] Mohamed A Sharaf, Jonathan Beaver, Alexandros Labrinidis, and Panos K Chrysanthis. Tina: A scheme for temporal coherency-aware in-network aggregation. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 69–76, 2003.

[49] Shaohuai Shi, Xiaowen Chu, and Bo Li. Mg-wfbp: Efficient data communication for distributed synchronous sgd algorithms. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 172–180. IEEE, 2019.

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

[51] Raajay Viswanathan, Arjun Balasubramanian, and Aditya Akella. Network-accelerated distributed machine learning for multi-tenant settings. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 447–461, 2020.

[52] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems*, 2:172–186, 2020.

[53] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[54] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[55] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, Yibo Zhu, and Lei Cui. Congestion control for cross-datacenter networks. *IEEE/ACM Transactions on Networking*, 30(5):2074–2089, 2022.

[56] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, 2017.

[57] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404, 2017.

[58] Junxue Zhang, Wei Bai, and Kai Chen. Enabling ecn for datacenter networks with rtt variations. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 233–245, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Bohan Zhao, Chang Liu, Jianbo Dong, Zheng Cao, Wei Nie, and Wenfei Wu. Enabling switch memory management for distributed training with in-network aggregation. In *2023 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2023.

[60] Yangming Zhao, Yifan Huang, Kai Chen, Minlan Yu, Sheng Wang, and DongSheng Li. Joint vm placement and topology optimization for traffic scalability in dynamic datacenter networks. *Computer Networks*, 80:109–123, 2015.