Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Canary: Congestion-aware in-network allreduce using dynamic trees

Daniele De Sensi [a,c,*], Edgar Costa Molero [b], Salvatore Di Girolamo [a], Laurent Vanbever [b], Torsten Hoefler [a]

[a] *Department of Computer Science, ETH Zurich, Rämistrasse 101, Zürich, 8092, Switzerland*
[b] *Department of Information Technology and Electrical Engineering, ETH Zurich, Rämistrasse 101, Zürich, 8092, Switzerland*
[c] *Department of Computer Science, Sapienza University of Rome, Piazzale Aldo Moro 5, Rome, 00185, Italy*

ARTICLE INFO

Keywords:
In-network compute
Allreduce
Load balancing

ABSTRACT

The allreduce operation is an essential building block for many distributed applications, ranging from the training of deep learning models to scientific computing. In an allreduce operation, data from multiple hosts is aggregated together and then broadcasted to each host participating in the operation. Allreduce performance can be improved by a factor of two by aggregating the data directly in the network. Switches aggregate data coming from multiple ports before forwarding the partially aggregated result to the next hop. In all existing solutions, each switch needs to know the ports from which it will receive the data to aggregate. However, this forces packets to traverse a predefined set of switches, making these solutions prone to congestion. For this reason, we design Canary, the first congestion-aware in-network allreduce algorithm. Canary uses load balancing algorithms to forward packets on the least congested paths. Because switches do not know from which ports they will receive the data to aggregate, they use timeouts to aggregate the data in a best-effort way. We develop a P4 Canary prototype and evaluate it on a Tofino switch. We then validate Canary through simulations on large networks, showing performance improvements up to 40% compared to the state-of-the-art.

1. Introduction

As the parallelism in computing systems steadily increases, the performance scalability of applications running on data centers becomes more dependent on communication performance. The allreduce operation is a widely-used communication primitive, both for the training of machine learning models [1], but also in scientific computing in general [2,3]. In an allreduce, each host has a vector of data elements. All the vectors must be *reduced* (i.e., aggregated) together element-wise using a commutative and associative operator. Then, after aggregation, data is distributed back to the hosts.

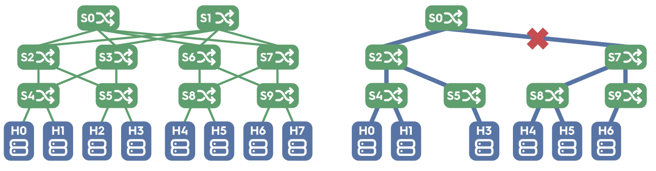
Allreduce accounts for a significant fraction of the training time of deep learning models, with estimates ranging from 50% for 10 Gbps networks [4], to 20%–30% for 100 Gbps networks [5]. Moreover, improvements in computation speed significantly outpace network bandwidth improvements. For example, we observed a 10x increase in GPU floating-point performance in 2.5 years [4,6]. In contrast, it took ten years for network bandwidth to increase by 10x [4,7]. Thus, we can expect application performance to be even more dependent on communication performance in the future.

For these reasons, several allreduce optimization techniques have been proposed [1], including (but not limited to) data quantization [8], sparsification [9], non-blocking collectives [10,11], solo allreduce [12], hierarchical synchronization [13,14], and in-network reductions [4,5,15,16]. This work focuses on in-network reductions, i.e., solutions where the network switches aggregate data. Several works showed that in-network allreduce transmits half of the data volume transmitted by the host-based bandwidth-optimal allreduce algorithm [17] (e.g., *ring* allreduce) [4,5,15]. Thus, if the network aggregates the data at line rate, this potentially halves the time required to complete the reduction.

All existing in-network allreduce algorithms adopt a similar approach [4,5,15,16,18,19], which we describe through an example. Let us consider the network depicted in Fig. 1(a), where the hosts *H0*, *H1*, and *H3–H6* want to perform an in-network allreduce. First, they set up a *reduction tree*, where the leaves are the hosts participating in the reduction, and the intermediate nodes are a subset of the switches in the network, as shown in Fig. 1(b). This setup step mostly involves installing forwarding rules in the switches. By doing so, for example, *S4* knows that it must aggregate the data coming from *H0* and *H1*, and forward the aggregated result to *S2*. Similarly, *S5* does not wait for data coming from *H2*, and forwards the data coming from *H3* to *S2* immediately after receiving it.

[*] Corresponding author at: Department of Computer Science, Sapienza University of Rome, Piazzale Aldo Moro 5, Rome, 00185, Italy.
E-mail addresses: desensi@di.uniroma1.it (D. De Sensi), cedgar@ethz.ch (E. Costa Molero), salvatore.digirolamo@inf.ethz.ch (S. Di Girolamo), lvanbever@ethz.ch (L. Vanbever), torsten.hoefler@inf.ethz.ch (T. Hoefler).

https://doi.org/10.1016/j.future.2023.10.010
Received 18 May 2023; Received in revised form 4 September 2023; Accepted 20 October 2023
Available online 29 October 2023



(a) A network with 8 hosts. (b) A reduction tree.

Fig. 1. In-network allreduce example.

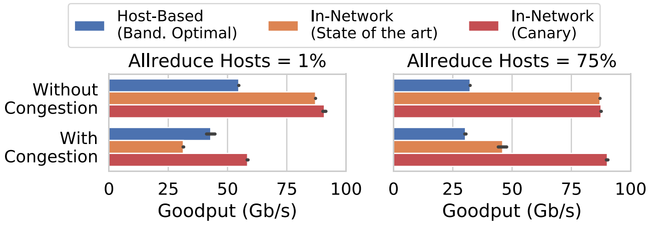


Fig. 2. Goodput of the bandwidth-optimal host-based allreduce, of the state-of-the-art in-network allreduces, and of Canary, when running on 1% and 75% of the hosts in a 1024 hosts network.

Albeit the described algorithm is simple and effective, it is also prone to congestion, because each switch in the reduction tree needs to know a priori its children and its parent in the reduction tree. This forces packets to be always routed on the same paths, regardless of their congestion. Network congestion can significantly slow down applications [20–24], and this is particularly relevant for in-network reductions. For example, let us assume that the link between *S7* and *S0* in Fig. 1 is congested. Even if *S0* already received the data from all its other children, it still needs to wait for the data coming from *S7* before starting the *broadcast* phase.

Thus, it is enough to have congestion on just one of the links composing the reduction tree to slow down the entire operation. A straightforward solution would consist in running the allreduce traffic in a separate traffic class. However, as we also show in Section 5.2.4, concurrent allreduces issued by different applications (e.g., different training jobs) would still interfere unless they are mapped to different classes. Because the number of concurrent allreduces can be higher than the available traffic classes [15,22,25,26], this is not a viable solution.

For these reasons, in this work we design and evaluate Canary (*Congestion-Aware In-Network Allreduce Using Dynamic Trees*), the first congestion-aware in-network allreduce algorithm. Canary relies on traffic load balancing algorithms to send packets on the least congested paths, dynamically building the reduction tree and adapting it throughout the execution based on congestion.

To illustrate the impact of congestion on in-network allreduce, we simulate a 2-level fat tree network [27] connecting 1024 hosts (we provide more details on the simulation infrastructure in Section 5.2). We execute an allreduce first on 1% and then on 75% of the hosts in the network. We observe in Fig. 2 that, when there is no congestion, both state-of-the-art in-network allreduce algorithms (using a static reduction tree) and Canary provide a 2x bandwidth improvement over the bandwidth-optimal host-based allreduce.

Then, we introduce congestion by concurrently executing a random uniform injection traffic pattern on the remaining hosts (99% and 25%, respectively). We observe that congestion causes a drop in the performance of the state-of-the-art in-network allreduce, which can even perform worst than the host-based allreduce.

Instead, Canary is less affected by congestion and provides a performance advantage over both the bandwidth-optimal host-based allreduce and state-of-the-art allreduces. As we describe in detail in Section 3, this is possible because Canary dynamically builds and adapts the reduction tree to the network conditions by relying on existing congestion-aware traffic load balancing techniques.

Table 1
Canary design principles, and comparison with the state-of-the-art. ✓: considered, 🔍: partially considered, ✗: not considered, ?: unknown. **CA**: Congestion-Awareness, **DRM**: Dynamic Resource Management, **DFT**: Dynamic Fault Tolerance.

Algorithm/Network	Year	CA	DRM	DFT
PERCS [29]	2010	✗	?	?
Aries [30]	2012	✗	?	?
Tofu [31]	2018	✗	?	?
SHARP [16,19]	2020	✗	✗	✗
Klenk et al.[5]	2020	✗	✓	✗
PANAMA [18]	2020	🔍	✓	✗
ATP [15]	2021	✗	✓	✗
SwitchML [4]	2021	✗	✗	✗
OmniReduce [32]	2021	✗	✗	✗
PIUMA [33]	2021	✗	?	✗
Flare [34]	2021	✗	✗	✗
Canary	2022	✓	✓	✓

In this work, we introduce the following contributions:

- We identify the impact of congestion on in-network allreduce, and we design an algorithm that relies on dynamic in-network reduction trees (Section 3).
- We improve the management of the switch resources and the fault tolerance compared to the state-of-the-art because the switches only store a soft state (Section 3).
- We implement a P4 Canary prototype on a Tofino switch [28], to assess the feasibility and limitations of our algorithm (Sections 4 and 6).
- We perform a detailed analysis through large-scale simulations, calibrated on our P4 implementation (Section 5) showing that, on congested networks, Canary is up to 40% faster than state-of-the-art in-network allreduce algorithms.

2. State of the art

We now discuss Canary's fundamental design principles (summarized in Table 1), that distinguish it from most existing in-network allreduce algorithms.

2.1. Congestion awareness

Existing interconnection networks have a large path diversity [22,35] and, to avoid congestion, load balance the traffic using algorithms like ECMP [36], flowlet switching [37,38], Valiant routing [39], and others. Some of these algorithms, like ECMP, distribute packets over the available paths by selecting the destination port based on the result of a hash function computed on some packet header fields. However, although many networks use ECMP [40], it has been shown that traffic often experiences congestion, even in the presence of alternative non-congested paths [37,41,42]. For this reason, some load balancing algorithms try to select the least congested path among those available. Such algorithms are also offered by some of the largest cloud providers when deploying high-performance virtualized clusters [43,44].

As discussed, however, all state-of-the-art in-network reduction algorithms always send the packets on the same paths regardless of congestion (Table 1, ✗). Moreover, to the best of our knowledge, only one in-network allreduce algorithm distributes the traffic over multiple reduction trees [18], showing advantages compared to a single reduction tree (Table 1, 🔍). Nevertheless, because the algorithm statically selects trees in a round-robin way, it is still congestion oblivious, even if it balances traffic over multiple paths.

Differently from all existing solutions, Canary dynamically builds, packet by packet, the optimal reduction tree based on the current network status, by routing packets on the least congested paths, and by aggregating the packets in a best-effort fashion (Table 1, ✓ – Section 3.1). It is worth remarking that simply adding load balancing capabilities to existing in-network allreduce algorithms would not work and that we need to design new ways to aggregate the data, as we discuss in Section 3.

2.2. Dynamic resource management

Each switch aggregates in a memory buffer, packet by packet, the data it receives from its children. Switches, however, have limited memory, and existing in-network allreduce algorithms adopt different approaches to deal with this. Most algorithms reserve some buffer space before starting an allreduce and, if there is no buffer space available, they fall back to a host-based algorithm [16,19,31,45]. Because this reservation process requires interacting with the control plane and might introduce some latency (up to *"a few seconds"* [18]), resources are usually reserved when the application starts and deallocated when the application terminates [4,16,19].

However, by doing so, long-lived applications would reserve resources for their entire execution, even if they would only sporadically use in-network reductions, potentially excluding other applications from using them. This decreases the number of concurrent in-network reductions that can be executed, which might be a relevant problem on multi-tenant datacenters [15] (Table 1, ✖). Other approaches, instead, dynamically partition the available resources across the currently active reductions [5,15,18], as we also do in Canary (Table 1, ✓ – Section 3.2). Specifically, in Canary the switches allocate and deallocate the memory in an on-demand fashion and strictly for the time required to complete the reduction.

2.3. Dynamic fault tolerance

Another critical point to discuss is how to deal with links or switch failures [46,47]. Because existing algorithms statically determine the reduction trees, if a link or a switch on the tree fails, the in-network reduction cannot progress. In most cases, the network controller detects switch failures [16,18] and builds another reduction tree. However, hosts might be in an inconsistent state after a failure (e.g., some hosts might have successfully received all the reduced data, whereas others might just have received part of the data). Recovering from such a state might imply re-issuing the entire reduction operation [16,19] from scratch on a different reduction tree or falling back to host-based reductions (Table 1, ✖). Other solutions [4] delegate the task of detecting and recovering from failure to the upper layer.

Canary, on the other hand is self-contained and can autonomously detect and recover from switch failures without re-starting the entire reduction from scratch. Indeed, Canary builds reduction trees dynamically and keeps only a soft state in the switches, and treats switches and links failures in the same way as a packet loss. In both cases, Canary requires only the retransmission of the small fraction of data that was stored in the switch when it failed (Table 1, ✓ – Section 3.3). Because managing both packet losses and switches faults adds complexity to the algorithm, Canary partitions its functionalities between switches and a *leader host* (Section 3.1.4).

3. Canary design

In general, we can consider in-network allreduce algorithms composed of two phases: a *reduce* phase (where data flows from the hosts to the root of the reduction tree) and a *broadcast* phase (where aggregated data flows from the root to the hosts). In Canary, hosts send packets to the same root switch (predetermined before starting the application), but packets are forwarded on the least congested paths towards the root to bypass congestion. Canary is orthogonal to the load balancing algorithm, and switches can use any existing algorithm to select the next-hop (either on a per-packet [41] or a per-flowlet granularity [37]).

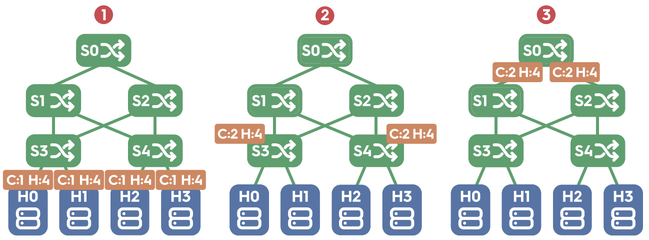


Fig. 3. Aggregation counter update. *C:* aggregation counter, *H:* number of hosts.

Canary aggregates packets that traverse the same switch in the same time window. Each switch allocates memory on-demand when receiving packets in the *reduce* phase and deallocates it in the *broadcast* phase, after sending the aggregated data to its children. To simplify the description of the algorithm, we first describe in Section 3.1 a scenario where switches have infinite memory, where a fully reliable network is used (i.e., packets are never dropped and switches/links never fail), and where there is at most one application at a time using Canary. We then remove these assumptions in Section 3.2, Section 3.3, and Section 3.4 respectively.

3.1. General design

Before describing the details of the algorithm, we analyze the challenges of using non-predetermined paths in in-network reductions. For the moment, we assume that each host can fit all the data it needs to reduce in a single network packet. We then discuss in Section 3.1.3 how to deal with larger data.

If we consider the network shown in Fig. 3, we can see that if we select *S0* as a root and we let packets follow different paths from the hosts to the root, *S1* and *S2* would not know how many packets they will receive from their children. According to the network conditions, sometimes both *S3* and *S4* might decide to forward their packets to *S1*, sometimes they might decide to forward their packets to *S2*, and sometimes *S3* might forward packets to *S1* and *S4* to *S2* (or vice-versa). As a consequence, *S1* might receive 0, 1 or 2 packets to aggregate (and the same for *S2*). This ambiguity is not present in existing in-network reduction approaches because packets always follow predetermined paths, and each switch knows exactly how many packets to wait for and aggregate. For this reason, existing in-network allreduce algorithms cannot simply be extended by using congestion-aware traffic load balancing, and a different approach must be adopted.

3.1.1. Reduce

Because the switch does not know how many packets to wait for, in the *reduce* phase Canary aggregates all packets received in a given time window. The first time a switch receives a reduction packet, it creates a *descriptor* and stores it in its memory. The descriptor contains a data accumulator (where the switch stores the data carried by the packet) and the root address (also carried by the packet). The descriptor also contains a timer that the switch starts when the first reduction packet is received. After storing the information carried by the packet in the descriptor, the switch drops the packet. The switch stores in the descriptor also the list of ports from which it received the allreduce packets (it will use it to reach the children in the *broadcast* phase). For all subsequent packets, the switch aggregates the data carried in the packets with that stored in the accumulator and then drops the packets. When the timeout expires, the switch retrieves the accumulated data from the descriptor, stores it in a new packet, and sends it to the next hop towards the root. The switch selects the next hop using any available congestion-aware load balancing algorithm, thus dynamically building the reduction tree depending on the current network conditions.

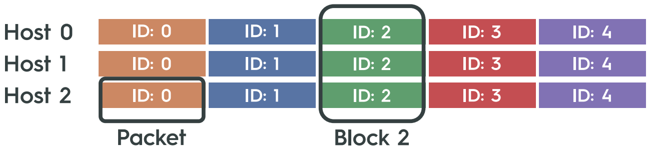


Fig. 4. Packets and reduction blocks.

Eventually, all packets reach the root and the *reduce* phase is concluded. Each packet sent by the hosts carries a counter indicating the number of already reduced packets, together with the number of hosts participating to that reduction (Fig. 3 (❶)). Counters coming from multiple packets are summed by the switches and stored in the descriptor. For example, if *S3* reduces the data coming from *H0* and *H1*, it will send a packet to *S1* with a counter equal to 2 (❷). At some point the accumulated counter will be equal to the number of hosts (❸), meaning that all data coming from the hosts has been reduced and that the root can start the *broadcast* phase.

Intermediate switches might receive some packets after the timeout expiration if the timeout is too short. In that case, the packet is identified as a straggler and immediately forwarded to the next hop. In turn, the following switch considers that packet as a straggler or not, depending on its timeout. The switch can determine if a packet is a straggler because it does not deallocate the descriptor until the reduction is completed (which cannot happen unless all the packets are received and aggregated by the root). On the other hand, a too large timeout might increase the latency of the packets and the completion time of the allreduce. However, this is noticeable only for small allreduces, as we show in Section 5.2.3.

3.1.2. Broadcast

When the *broadcast* starts, the root retrieves from the descriptor the list of the ports from which it received the data (i.e., the list of its children), forwards the aggregated data on those ports, and then deallocates the descriptor. After receiving a reduced packet from its parent, a switch forwards the packet to its children and deallocates the descriptor. In a nutshell, Canary reserves resources in the switches dynamically and strictly for the time required to complete the *reduce* and *broadcast* phases. A switch allocates a descriptor when it receives the first packet going to the root and deallocates it when it receives the packet coming down from the root. Eventually, the reduced data reaches the hosts that started the reduction, and the allreduce operation terminates.

We can notice that, whereas Canary dynamically routes packets in the *reduce* phase, in the *broadcast* phase, it forwards them on the same paths used in the *reduce* phase. A fully dynamic multicast would require explicit deallocation of resources because packets might cross different switches than those used in the *reduce* phase. This would add unnecessary complexity to the design of the algorithm because, as we show in Section 5, Canary can still bypass most of the congestion in the network. Also, as we discuss in Section 3.1.3, the reduction tree is dynamically rebuilt on a packet-by-packet basis, thus reducing the probability of finding persistent congestion in the *reduce* phase.

3.1.3. Packetization

We assumed so far that all the data to be reduced by a host would fit in a single network packet. However, this is usually not the case, and data is often larger than the MTU (*Maximum Transmission Unit*). Thus, each host assigns a unique identifier (*id*) to each packet it sends, as shown in Fig. 4. Packets with the same *id* belong to the same *reduction block* and must be aggregated together by the switch. The switches can now process multiple blocks concurrently and store a separate descriptor for each block in a table indexed by *id*. Alongside the data accumulator, the list of children, and the timer, the descriptor also contains the block *id*.

For the moment, we assume that there is always space available to store the descriptor. We then describe in Section 3.2 how Canary works when the descriptor cannot be stored. To further improve load balancing, Canary aggregates each block in a different root, determined before starting the application (e.g., the hosts could select the roots in a round-robin way).

3.1.4. Leader host

Programmable switches have limited resources and can only perform simple operations. As a consequence, it is not possible to fully implement complex tasks such as tracking and retransmission of lost packets or recovery from other switches failures, and Canary delegates these tasks to a *leader host*, similarly to what happens in other algorithms [15]. Reduction packets are still sent towards the root and aggregated on its path, as we described above. However, the root switch sends the aggregated data to the leader host as soon as the timeout expires (or when all the expected data is received).

If we consider the example in Fig. 3, we could use switch *S4* as root switch, and either *H2* or *H3* as leader host. The leader does not send its data on the network. Instead, it waits for data to arrive from *S4*, after which it aggregates the received data with its own, and then starts the *broadcast* phase. It is worth remarking that Canary still relies on a root switch that should aggregate as much data as possible. Ideally, the root should send to the leader host only one fully aggregated packet, thus avoiding having the leader receive multiple packets per block, which would reduce the operation's bandwidth.

Although this solution increases the latency because packets need to cross the network stack of the leader [48], it allows us to partition tasks between hosts and switches. The switches only perform simple tasks, like aggregating data packets in a best-effort fashion. On the other hand, the leader host handles more complex operations like the retransmission of lost packets (Section 3.3). To reduce the latency required to cross the network stack, leader functionalities could be offloaded to programmable NICs [49–52] or implemented on a high-performing network stack such as DPDK [53]. Moreover, because Canary sends each reduction block to a different leader, the pressure on individual hosts is reduced. Namely, suppose N hosts participate in the reduction. In that case, each host will be the leader only for 1 block out of N, thus receiving data at a much slower rate than line rate, increasing the time budget available for performing leader tasks.

3.2. Resource management

Switches have limited memory resources and could not allocate memory for all the reductions concurrently executed over the network. Because Canary relies on adaptive routing, it cannot reserve any resource because the paths that packets will take are not known a priori, and reserving resources on all the network switches would unnecessarily increase resources occupancy. Instead, Canary stores block descriptors in a static array and, when a packet is received, the switch maps the *id* to a specific array location (e.g., by using a hash function). If the location is empty, the switch stores data in the accumulator and initializes the descriptor. Otherwise, if the stored accumulator belongs to the same *id*, the switch aggregates the data carried by the packet with that in the accumulator. If the stored accumulator belongs to a different *id*, then we have a *collision*. Collisions might happen because the hash function might map to the same location reduction blocks with different *ids* (and that do not need to be aggregated together).

3.2.1. Collisions management

If there is a collision, the switch cannot store the descriptor of the new block. In principle, the switch could forward the packet to the next hop, delegating the aggregation to the following switches towards the root. However, in this case, the switch would not be able to participate in the subsequent *broadcast* phase as it did not store the descriptor (containing the list of children) for that *id*. For example, in Fig. 5 (❶)

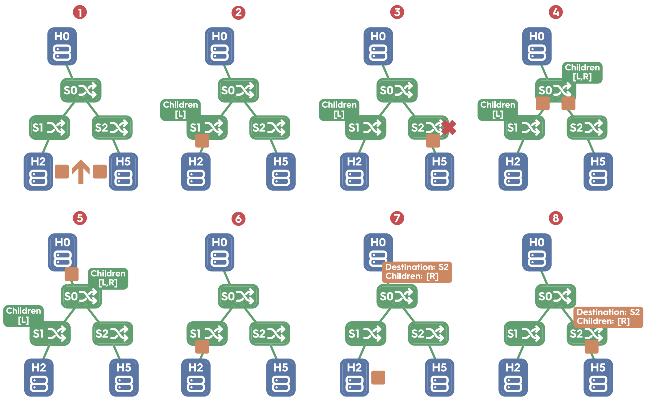


Fig. 5. Collision and tree restoration.

hosts *H0* (leader), *H2*, and *H5* want to reduce their data. *S1* receives data from *H2* and experiences no collisions (❷). *S2* receives data from *H5* (❸) but detects a collision. Let us assume that *S2* just forwards the data to the next hop and the reduction progresses as normal from that point on.

During the *broadcast* phase (❹), *S2* will not be able to forward data to its children because, due to the collision during the reduction phase, it was not able to store the descriptor that would contain, among others, the child port identifier. Ultimately, *H5* will not receive the reduced data. In general, if a switch cannot store the block descriptor because of a collision, the entire subtree rooted at that switch will not be reachable during the *broadcast* phase. It is worth remarking that each switch needs to store the ports connecting it to its children. Indeed, the leader host cannot just insert the addresses of the children of all the switches in the packet because this would be linear in the number of hosts participating in the reduction.

To solve this problem, we adopt a simple but effective solution called *tree restoration*. After a conflict, the switch inserts its address in the packet alongside the identifier of the port from which it received it. Then, it forwards the packet directly to the leader host (the packet is marked and ignored by all the other switches on the path). After the reduction, the leader host knows the unreachable subtrees: i.e., it has a list of switches and respective ports from which they received packets generating collisions. During the *broadcast* phase, the leader host uses this information to send an additional packet to these switches, allowing them to bootstrap a local broadcast, restoring the subtree. Other than the reduced data, these packets also carry the list of ports on which the switch must forward the data (e.g., encoded as a bitmap).

In our example, when the collision is detected (❸), *S2* forwards the packet to the leader host, together with its address and the port number (*R*) from which the packet has been received. After the reduction phase, the leader host *H0* starts the *broadcast* as usual (❺). The broadcast packet eventually reaches *S2*, but it is not able to progress since *S2* is missing information about its subtree (❻). The leader host also sends a *"restoration"* packet to *S2* (❼), making it able to forward the data on port *R* towards *H5* (❽).

This approach works even if, after some collisions, the entry becomes available and subsequent packets are successfully stored. In that case, the switch stores identifiers of some children in the block descriptor on the switch, and hence they will be reached by the normal *broadcast* phase. Others, i.e., the ones that generated conflicts, will be reached through the tree restoration process.

Because of the extra network traffic generated after a collision (e.g., restoration packets), this approach can lower the throughput (i.e., the leader host receives more packets due to missed aggregations). However, collisions only happen if a switch receives multiple packets with different *ids* mapping to the same descriptor entry in the same time window. If this performance penalty is not acceptable, Canary can avoid collisions entirely by setting a limit on the number of concurrent allreduces, and by statically mapping *ids* to descriptor array entries.

3.2.2. Switch memory occupancy modeling

We now analyze how much switch memory an allreduce can occupy. A block descriptor is allocated in a switch when the first packet of that block is received and deallocated when the fully aggregated data is received. For this reason, switches at the bottom level of the tree keep descriptors allocated for the longest time and, to estimate the maximum memory occupancy, we model the memory occupancy of those switches.

We denote the network bandwidth with b, the network diameter with d, the 1-hop delay with l, the timeout with t (i.e., how much time a switch waits before sending the partially aggregated data to the next hop), and the time required to the leader to aggregation its data with r. Then, the time between the allocation and the deallocation of a descriptor can be measured as $2d(l+t)+r$. Each descriptor contains the aggregated data, plus a few more bytes for storing the root address and other information. Thus, we can approximate the size of a descriptor with the MTU m. By using Little's Law, if we assume to send MTU-sized packets, we can estimate the number of bytes occupied by descriptors as:

$$\frac{b}{m}(2d(l+t)+r)m = b(2d(l+t)+r)$$

Recent networks have a diameter of up to five and a per-hop latency of around 300 nanoseconds [22], and programmable NICs can perform tasks similar to those performed by the leader host in around one microsecond [50]. Thus, on a 100 Gbps network with a one-microsecond allreduce timeout, each allreduce might store up to 175KiB in each switch crossed by its packets.

It is worth remarking that the memory occupancy is independent of the actual size of the data to be reduced because Canary aggregates data block-by-block, and the bandwidth–delay product bounds the number of in-flight blocks. Also, the occupancy does not depend on the number of hosts participating in the reduction. Indeed, each switch stores only one descriptor for each block, independently of how many packets (one from each child) are aggregated in that block.

3.3. Packet loss and fault tolerance

Canary treats packet losses and switches failures in the same way. Indeed, in both cases, the leader does not receive some packets (if the loss/failure occurs in the reduce phase), or the hosts do not receive the reduced data (if the loss/failure occurs in the broadcast phase). Without loss of generality, we describe how to manage packet losses since the same approach is also used for managing switches failures.

To detect a loss, all the hosts (excluding the leader) set a timeout for each packet right before transmitting it. When the reduced data arrives, the host deletes the timer. If the timeout expires, a retransmission request is issued. If the leader receives a retransmission request, two situations might occur. If the leader entirely reduced the data, the packet was lost during the broadcast phase, and the reduced data is retransmitted to the host that issued the request. Otherwise, if the leader only partially reduced the data, some packet was lost in the reduce phase.

In this case, the leader does not know which packet was lost. Indeed, to know which packets contributed to the partially reduced data, each switch would need to keep a bitmap of all the hosts that contributed to the data reduced so far, which would be linear in the number of hosts participating in the reduction. However, having this bitmap in the packet is infeasible because allreduces might span thousands of hosts [1,54], and existing programmable switches can only process a few hundred bytes per packet (Section 6). Accordingly, because the leader cannot determine which packet should be re-transmitted, it broadcasts a failure message. Upon the reception of this message, the hosts re-issue the reduction of that packet with a different *id* (or they can reduce that packet only by using a host-based reduction algorithm). To avoid overloading the network with reduction packets, the hosts fall back to a host-based reduction after a given number of failed retransmissions.

It is worth remarking that when a host terminates the reduction, it cannot simply modify or deallocate the reduced data because the other hosts might not have successfully terminated the reduction yet. Accordingly, it must preserve the part of the data for which it was the leader to re-transmit the packets in case any retransmission request arrives. For small-size reductions, the leader can store a copy of the data and deallocate it when it receives a fully reduced packet for a subsequent reduction. Indeed, the hosts can start the subsequent reduction only if they have already completed the previous one. If there are no subsequent reductions, an explicit completion notification is required (for example, by issuing a barrier). Because preserving the data between subsequent reductions could potentially double the memory consumption, for large reductions the hosts always explicitly notify the completion. The explicit notification introduces a marginal latency overhead compared to the allreduce and allows the data to be deallocated or modified immediately after the notification, not requiring any additional copy.

Although Canary can autonomously manage switch failures without re-issuing the entire allreduce operation, host failures must be managed at a higher layer (e.g., with checkpoint/re-start solutions).

3.4. Multitenancy

The switch does not have any knowledge of the different applications or users running on the system and simply aggregates together packets with the same *id*. To support multiple applications, each of them must generate unique *id*s. Thus, Canary *id*s are built by concatenating an identifier of the application (e.g., generated by the workload manager) and an identifier that each application increases for every subsequent packet.

It is worth remarking that having multiple concurrent allreduces does not necessarily increase the amount of data that a switch needs to store. Indeed, a descriptor is allocated only on the switches traversed by the packets belonging to the corresponding block and strictly for the time needed to reduce that block. In a nutshell, running multiple concurrent allreduces, each on a few hosts (thus connected through a few switches), might consume the same amount of resources of a single allreduce on a higher number of hosts (thus spanning over more switches).

3.5. Summary

We now wrap up Canary design. First, each host splits the data in multiple packets, each marked with a unique *id*. When a switch receives a data packet, it maps the *id* to a specific entry of the array containing the descriptor for that *id*. If the entry is available, the switch stores in the table the data carried by the packet, updates the list of children, starts a timer, and drops the packet. If a descriptor with the same *id* is already present in the entry, the switch accumulates the data, updates the list of children, and drops the packet. If a descriptor with a different *id* already occupies the entry, the switch inserts in the packet its address and the identifier of the port from which it received the packet, before forwarding it to the leader.

When the timer of a descriptor expires, the switch sends to the next hop the data contained in the descriptor. The port is selected based on the root address stored in the switch using any available load balancing algorithm. If a packet arrives and the timeout for that *id* already expired, the switch updates the list of children and forwards the packet to the next hop. Eventually, when the leader starts the *broadcast* phase, the switch receives the fully reduced data.

If there is a descriptor for that *id*, the switch forwards the packet to all its children and removes the descriptor from the table. If the switch does not have a descriptor for that *id* (because it could not store it due to a collision), it drops the packet. Later, it receives the data from the leader specifying the children to which the packet should be forwarded.

At some point, all the data will reach the tree's leaves. If a packet is lost or a switch fails, some leaves send a retransmission request to the leader. When the leader receives a retransmission request, it can either re-transmit the fully reduced data or require that block to be reduced from scratch (if it did not already entirely reduce the data).

4. Implementation

We implemented Canary using a state-of-the-art Intel Tofino programmable switch. Although existing programmable switches can process terabits of data per second (1.2 billion packets per second for each pipeline), they limit the type of computation they can perform on packets. These limitations drove some of the main choices in our design. For example, having one of the hosts acting as the reduction leader pushes some complexity to the host and keeps the code executed by the switch as simple as possible.

Programmable switches process packets through multiple pipelines, each composed of multiple stages. In our implementation, we use the first stages to determine if the packet is a reduce or a broadcast packet and check if the packet generates any collision when accessing the descriptors table. We then use subsequent stages to read/write data. Because the goal of Canary is to leverage the speed of such programmable devices, all the data is managed in the *data plane* and stored into registers. Content-addressable memory (CAM) [55] is usually available but can only be updated from the *control plane* [28,56]. We do not use CAM for storing the data because interactions with the control plane would significantly increase the latency [57].

4.1. Packet format

Because we are using programmable switches, we define a custom packet format for Canary packets. To reduce the packet overhead, Canary sends packets directly on top of Ethernet. However, any other encapsulation could be used, and Canary packets could be sent on top of IP or UDP. A Canary packet is composed of the following fields:

- **Destination (4 bytes)** IP address of the leader host. Canary uses the same routing tables used for IP routing to determine how to reach the destination.
- **Id (4 bytes)** Unique identifier of the packet.
- **Counter (2 bytes)** Number of reduced packets (Fig. 3).
- **Hosts (2 bytes)** Number of hosts participating in the reduction (Fig. 3).
- **Children (4 bytes)** When a switch cannot store a packet because of a collision, this field carries the identifier of the port from which the packet was received (Section 3.2).
- **Switch Address (2 bytes)** When a switch cannot store a packet because of a collision, this field carries the switch address (Section 3.2).
- **Bypass (1 bit)** If set, the switch should not further process the packet but only forward it to the next hop.
- **Multicast (1 bit)** If set, the packet must be multicast to the children of the switch.
- **Padding (6 bits)** Used to pad the packet size to a multiple of 8 bits.
- **Data (128 bytes)** Data to be reduced.

4.2. Multicasting

Multicasts could, in principle, have an impact on the capacity to run our allreduce algorithm at line rate. Indeed, if the switch generates m packets for each packet it receives, it might decrease the achievable bandwidth by a factor of m. However, we observe that if a switch multicasts a packet on m ports (its children), this is because it previously aggregated the data coming from m ports. Accordingly, the switch forwards, on average, one packet for each packet it receives.

The switch keeps a table associating to each port the corresponding one-hot encoding. Every time a packet is received, the one-hot encoding of the input port is retrieved through a TCAM (pre-configured in the control plane) and added to the bitmap storing the children. Thus, when a packet needs to be multicasted, the switch knows that it must send it to all the ports set in the children bitmap.

Programmable switches require multicast groups to be pre-configured by specifying the association between a group identifier and a list of ports on which the switch will send packets directed to that group. In our case, the group identifier could simply be the bitmap associated with the specific list of ports. For example, let us suppose we have a switch with eight ports and that we want to multicast on the ports `[0,2,3,5]`. The binary representation of this list is `00101101`. So we would have to set up a rule such as `00101101 -> [0,2,3,5]`. However, Canary uses adaptive routing, and the switches multicast the packets to the same ports from which they have been received, which are not known a priori. Accordingly, we should store all the possible combinations of ports, which is exponential in the number of ports.

Because this requires too many resources, to reduce the storage requirements, we divide the children bitmap in *shards*. For example, the children bitmap `00101101` can be divided into two shards of 4 bits each. We prepend to each shard its index so that the two shards become `1 0010` and `0 1101`. We then store the association between all the possible shard values (that, in this case, would be $2 \cdot 2^4$) and the corresponding list of ports. In our example, we would have the rules `10010 -> [5]` and `01101 -> [0,2,3]`. This technique reduces the number of multicast groups to store in the switch tables from 2^p, to $2^{\frac{p}{s}} \cdot s$, where p is the number of ports, and s is the number of shards. For example, on a 64 port switch with four shards, this requires using 256 thousand entries, which is far within reach of current programmable switches [58], as we also show in Section 5.1.

4.3. Timeouts

As described in Section 3.1.1, Canary relies on timeouts to avoid statically setting up the reduction tree. In our Canary implementation, every time the switch receives a reduction *id* for the first time, it stores in the descriptor the current timestamp (alongside the data to reduce and the other information). Some programmable switches [28] provide one or multiple *packet generators* that can generate packets at a predefined rate and with predefined content. For example, we can set up the packet generators so to generate *clock* packets. Every time one of these packets is received, the switch can check an entry of the table and, if expired, send the content in that entry to its parent. Alternatively, Canary could be implemented on programmable switches that support timer-based events [59].

5. Evaluation

In this section, we evaluate the performance of Canary, first by analyzing its performance on a Tofino switch (Section 5.1), and then by simulating it on a large network (Section 5.2) connecting 1024 hosts through 64 switches.

5.1. Single switch implementation

To verify the feasibility of our design, we implemented and validated our P4 Canary prototype on a Tofino Wedge 100BF-32X switch [28] with 32 100 Gbps ports. This allowed us to understand existing switches' limitations and drive our design choices. We were able to allocate enough registers to allocate up to 32K descriptors, allowing us to run at least 25 concurrent allreduces of different tenants or applications (Section 3.2.2). Our P4 Canary implementation only uses 14.38% of the switch SRAM. However, Canary uses most of the Arithmetic Logic Units (ALU), up to 81.25% of those available on the switch, to aggregate the elements carried in the packets. Canary leaves enough resources available for running it alongside traffic load balancing algorithms such as flowlet switching [37] that, on the same switch, uses 2.26% of the available SRAM and 0.04% of the ALUs [60].

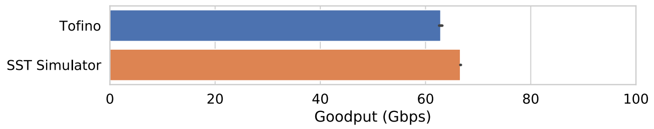


Fig. 6. Goodput (Gbps) of our P4 Canary prototype and of our implementation in the SST simulator, when sending packets with 128 bytes of useful payload.

To measure the goodput of our P4 prototype implementation, we connected two hosts equipped with 100 Gbps Mellanox ConnectX-5 NICs to the Tofino switch. We emulate a leaf switch that receives the data to be reduced from the two hosts, aggregates it, and forwards the aggregated result to the next switch in the reduction tree. The two hosts inject the data using Moongen [61], a DPDK [53] wrapper.

We benchmark a 4MiB allreduce and report the goodput in Fig. 6. We also report the goodput achieved by our simulation infrastructure (that we describe in Section 5.2) in the same setup. It is worth remarking that, due to the limited number of *match-action* tables available in existing programmable switches, we can store up to 32 4-bytes elements. Accordingly, each packet contains 128 bytes of useful payload and 57 bytes of headers (19 bytes of Canary header, 14 bytes of Ethernet header, and 24 bytes of framing overhead).

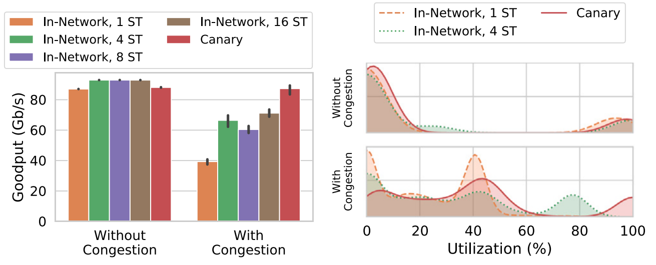
Programmable switches are composed of multiple processing pipelines, and some existing P4 prototypes [4,15] partially overcome this limitation by striping the packet across pipelines. For example, the first pipeline would store the first 32 elements, then recirculate the packet to the second pipeline, which would store the following 32 elements, and so on. In this way it is possible to have up to 128 [15] or 256 elements [4] per packet. However, to have enough recirculation bandwidth and avoid packet drops, some switch ports must be dedicated to packets recirculations only.

Additionally, it is reasonable to assume that the number of match-action tables in each pipeline stage will increase with the next generations of programmable switches, thus allowing processing more elements per packet. For example, some programmable switches already provide the possibility to process up to 48 elements per pipeline [60]. For these reasons, in the following, we run simulations with 256 elements per packet for all the in-network algorithms.

5.2. Large network simulations

To evaluate Canary performance at scale, we modified the SST simulator [62,63] so that switches can modify the packets they receive before forwarding them. We build in the simulator a two-level fat-tree network [27]. The network comprises 32 switches at the bottom level, each with 64 ports (32 connected to the hosts and 32 to the switches at the upper level). The top level of the fat-tree comprises 32 switches, each with 32 ports (one port connected to each of the switches at the bottom level). Both the hosts and the switches have 100 Gbps network interfaces.

We calibrated the simulator so that hosts can inject packets into the network at line rate and so that the switches can aggregate the data at the same speed as our Tofino prototype (as we show in Fig. 6). The simulated network uses up/down routing. When packets flow from hosts to upper levels of the fat-tree, each switch can select one among multiple *up* ports. By default, each switch sends packets on a default *up* port (selected depending on the packet destination). If the output port buffer has an occupancy higher than 50% of its capacity, the switch forwards the packet on the *up* port with the smallest number of enqueued bytes.



(a) Goodput (Gbps).

(b) Links utilization (%).

Fig. 7. Goodput and links utilization when 512 hosts execute an allreduce and 512 hosts generate congestion. *ST:* Static Tree(s).

To compare our solution with the state-of-the-art, we implemented in the simulator the following allreduce algorithms:

- **Ring** The bandwidth-optimal host-based *ring allreduce* algorithm [17]. This solution does not rely on any in-network compute capability.
- **In-Network, N static trees** An in-network algorithm using static reduction trees. We consider either the case when a single tree is used, similar to what is done by SHARP [16,19], SwitchML [4], and ATP [15], and also the case where N trees are used and each block is sent on a different tree in a round-robin way, similar to what done by PANAMA [18].
- **Canary** The in-network algorithm we propose in this work, which dynamically builds reduction trees.

To analyze the impact of congestion on the allreduce performance, we split the hosts into two sets. While some hosts run the allreduce, the remaining hosts generate network congestion by executing a random uniform injection traffic pattern. In this pattern, each host sends a message to a randomly selected host and receives a message from another randomly selected host. Each host changes its random peer throughout the execution to assess the ability of Canary to react to dynamically changing congestion patterns. We execute each test five times, each time randomly allocating the hosts executing the allreduce and those generating the congestion. When using static reduction trees, we also randomly pick the roots of those trees.

5.2.1. Comparison with the static trees approach

First, we report in Fig. 7(a) the comparison between Canary and the in-network algorithm using one or multiple static trees. We allocated 512 hosts to the allreduce and used the remaining 512 hosts to generate congestion. We report the bandwidth with and without congestion for a 4MiB allreduce. Whereas in the absence of congestion, the performance of all the approaches is comparable, when introducing congestion Canary performs significantly better than the solutions using statically configured reduction trees. Indeed, whereas solutions using static trees are severely affected by congestion, Canary does not experience any performance degradation. For this reason, we observe performance improvements up to 2x compared to solutions using a single reduction tree and up to 40% compared to those using multiple trees. Moreover, we also observe that using more than four trees for solutions relying on static trees leads only to a marginal performance improvement. For these reasons, in the subsequent analysis, we consider a solution using four static trees, as in the original PANAMA paper [18].

We also report in Fig. 7(b) the distribution of links utilization (each sample is a network link). For the sake of readability, we only report those of Canary, one static tree, and four static trees. We observe that when there is no congestion, there are no significant differences between the three approaches, and each link is either idle (0% utilization) or fully utilized (around 90% utilization). However, when we introduce congestion, we observe that Canary is characterized by fewer idle links and better distributes the traffic over the available links.

At first sight, it might seem like the in-network solution with one static tree does not fully utilize any network link because there are no humps around 80%–100% utilization. However, by analyzing the data more in detail, we found two links with utilization greater than 80%. Because these two links are shared between the in-network allreduce and the application that generates congestion, this is enough to slow down the in-network allreduce by more than 50% (Fig. 7(a)) and to reduce the overall network utilization. Indeed, in the presence of congestion, we observed an average network utilization (computed as the average of all the links utilization) of 40.2% for Canary, 29.5% for the in-network allreduce with four trees, and 20.9% for the one with one tree.

5.2.2. Goodput for different congestion intensity

We now analyze the performance of Canary when changing the number of hosts generating congestion by comparing it to the host-based bandwidth-optimal ring allreduce and the in-network allreduce using static trees. We report the results of this analysis in Fig. 8. We ran a 4MiB allreduce, and we increase the number of hosts executing the allreduce from 5% to 75% of the 1024 hosts available in the system. The hosts not executing the allreduce generate congestion through a random uniform communication pattern. First, we observe that Canary consistently improves performance compared to other solutions.

When using only 5% of the hosts for the allreduce (thus using 95% of the hosts to generate congestion), Canary performance only decreases by 20%. In contrast, the performance of the in-network static solutions decreases by 66% when using a single tree and by 47% when using four trees. When increasing the number of hosts executing the allreduce (thus decreasing congestion), the performance gap shrinks, but Canary still provides 2x improvement compared to the single static tree solution, and 23% improvement compared to the solution using four reduction trees. Eventually, we observe that in some cases, the congestion decreases the performance of the single tree solution so much that it does not provide any performance advantage compared to the host-based ring allreduce, as also outlined in other recent works [18].

5.2.3. Runtime for different data sizes

We now analyze the allreduce runtime for different data sizes. We allocate 20% of the hosts to the allreduce, whereas the remaining 80% generates congestion. We report in Fig. 9 the runtime (in microseconds) with and without congestion. We observe that for small allreduces, Canary is characterized by a higher runtime because the switch only forwards (aggregated) packets after the timeout period expires. When increasing the size of the data exchanged by the allreduce, the performance advantage of Canary increases because the runtime of large allreduces is dominated by the bandwidth, and the extra latency introduced by the timeout mechanism becomes negligible. We also observe that 1KiB and 256KiB ring allreduces have the same runtime. Indeed, the ring allreduce is the host-based bandwidth-optimal allreduce algorithm, but, for small messages, its runtime is dominated by latency and setup of communication phases [5,17].

5.2.4. Multiple concurrent allreduces

Using statically configured trees also significantly decreases the aggregation bandwidth when multiple tenants (or multiple applications) concurrently run allreduce operations (e.g., multiple training jobs). Therefore, we equally partitioned the system between multiple co-running allreduce operations to analyze this effect. Furthermore, because most existing in-network allreduce algorithms statically partition the switch resources across the tenants [4,16,34], to have a fair comparison, we adopt a similar approach also in Canary.

We report in Fig. 10(a) the average goodput across all the concurrent allreduce operations. First, we observe that when increasing the number of concurrent allreduces (thus decreasing the number of hosts allocated to each allreduce), the average goodput of the ring

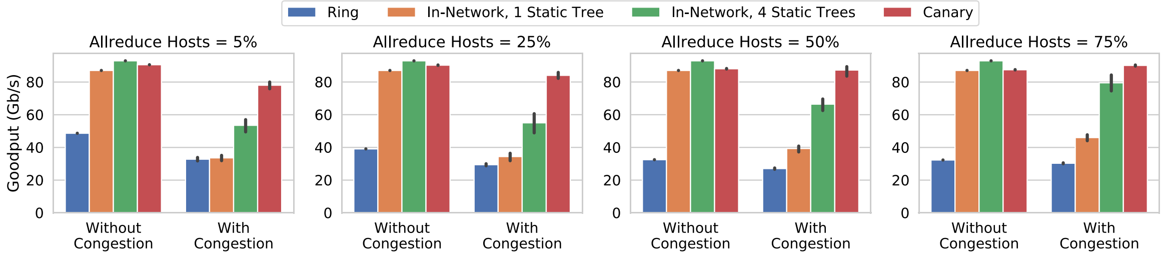


Fig. 8. 4MiB allreduce goodput (the higher the better) for different hosts count. The hosts not performing allreduce generate random uniform traffic to introduce congestion.

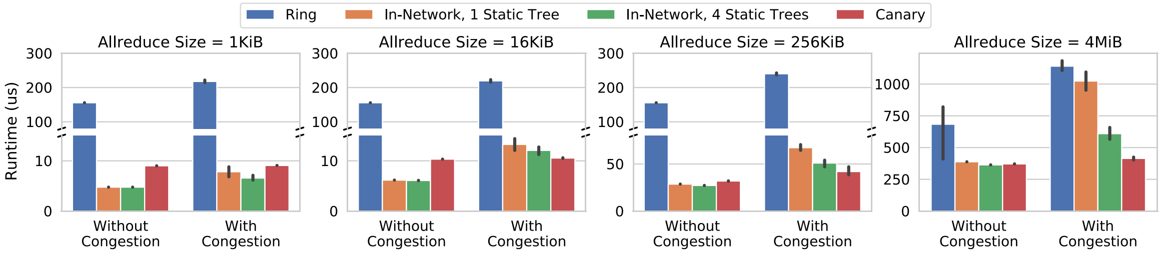
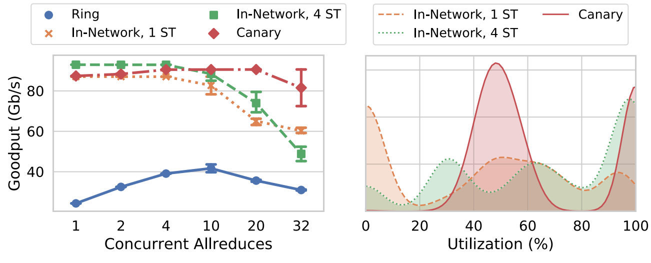


Fig. 9. Allreduce runtime (the lower the better) for different message size, when 20% of the hosts are allocated to the allreduce, and 80% to an application generating random uniform traffic.



(a) Goodput (Gbps).

(b) Links utilization (%).

Fig. 10. Average goodput of multiple concurrent 4MiB allreduces (left), and link utilization when running 20 concurrent allreduces (right). *ST:* Static Tree(s).

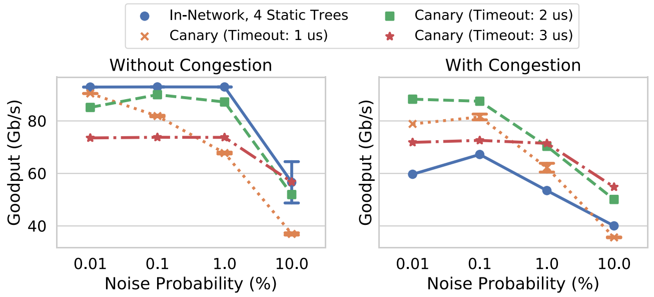


Fig. 11. Goodput of a 4MiB allreduce executed on 512 hosts, in a scenario where before sending a packet a host might add a delay of 1us with a given probability.

allreduce increases. This is a known effect [17] because the performance of the ring allreduce increases when decreasing the number of hosts. However, the performance drops when running more than ten concurrent allreduces due to the increased congestion. We also observe the performance of in-network static allreduce algorithms drops by 40% when increasing the number of concurrent allreduce operations due to congestion. On the contrary, Canary is almost unaffected and allows up to 32 concurrent allreduces at 80 Gbps each.

By analyzing the distribution of the links utilization in Fig. 10(b), we observe that Canary is characterized by the lowest number of idle links. We also observe that using four static trees improves the links utilization compared to using only one static tree. However, as shown in Fig. 10(a), this is still not sufficient to avoid congestion because multiple allreduces concurrently use some links.

We observed an average network utilization of 67.2% for Canary, 62.9% for the static in-network allreduce with four trees, and 21.8% for the one with one tree. Although Canary and the solution using four trees lead to a similar average network utilization, Canary performs better because it distributes the traffic more evenly across the network (e.g., it does not have any link in the 70%–90% range of utilization).

5.2.5. *Impact of timeout and noise*

One of the key points of Canary is the use of timeouts to decide when a reduction block has been fully aggregated and can be sent to the next hop in the reduction tree. As described in Section 3.1.1, if the timeout is too short, or if a packet is delayed for any reason (e.g. OS noise [64]), the straggler packet is sent to the next hop right after it is received. Although this guarantees that all the packets are eventually successfully aggregated, it might introduce some performance penalty, because a switch now sends more packets than the optimal.

To analyze this scenario, we execute a 4MiB allreduce on 512 hosts, with and without congestion, by comparing it with the in-network allreduce using four static trees and by analyzing the performance for different values of the timeout. Also, every time a host sends a packet, it has a given probability (*noise probability*) of delaying the transmission by 1 microsecond. We report the results of this analysis in Fig. 11, by showing the runtime when changing the noise probability from 0.01% to 10% (i.e., each host on average delays 10% of the packets it sends by 1 microsecond).

When there is no congestion on the network, we observe that Canary is characterized by a lower goodput, as also observed in the previous experiments. Because in this experiment we randomly delay packets by one microsecond, the scenario with a one-microsecond timeout generates several stragglers, decreasing the Canary goodput. This effect is less visible for larger timeouts, which can absorb differences in packet delays.

We also highlight how the performance does not increase nor decrease monotonically with the timeout value. Indeed, a long timeout unnecessarily increases packet latency, whereas a short timeout generates stragglers. However, even if we have a $3x$ difference between the timeout values, we observe at most a 30% difference in the performance. To further mitigate the timeout impact, a possible future extension would be to dynamically select the timeout based on the current network conditions.

Last, when introducing congestion, Canary instead outperforms the static in-network allreduce regardless of the noise probability and timeout values. Indeed, even if stragglers are generated, their impact on the performance is compensated by the fact packets are forwarded on less-congested paths.

6. Discussion

This section discusses some of the limitations of existing programmable switches and their impact on Canary.

Collisions. If two packets with two different *id*s map to the same table entry, Canary forwards the second packet directly to the leader host, generating extra network traffic and potentially reducing the performance (Section 3.2). For this reason, collisions should happen as rarely as possible. To reduce the number of collisions, in principle Canary could use slightly more sophisticated schemes like Cuckoo hashing [65] or double hashing [66]. However, due to the lack of iterative constructs and limited resources, this is not possible on existing programmable switches. As an alternative, the administrator can limit the number of concurrent allreduces and statically partition the table, as done in most in-network allreduce algorithms.

Packet size. Most existing programmable switches can only process a limited number of data elements per packet, based on the number of physical resources available. Although, as discussed in Section 5.1, this number can be increased by exploiting recirculations, it also requires dedicating most of the switch ports to packet recirculations [4,15]. As an alternative, Canary could be implemented on different programmable switch architectures that do not have limitations on the number of elements that can be processed per packet [34].

Floating-point arithmetic. Most programmable switches do not provide floating-point units [67] and, for this reason, most in-network reduction solutions targeting programmable switches assume that the values to be reduced are converted to fixed-point arithmetic before being transmitted over the network [4,15,32]. It has been shown that such techniques do not significantly impact the convergence of deep learning training, and thus they could seamlessly be used with Canary.

Support for other collectives. Although we focused on the allreduce, a similar approach could be used for other collective operations. For example, a *reduce* can be easily implemented by selecting as leader node the destination of the *reduce*, and by skipping the broadcast phase. Similarly, a *barrier* can be implemented by having a 0-bytes allreduce, and a *broadcast* by having the node acting as the source of the broadcast sending data to the leader host, thus skipping the data aggregation phase.

Leader failure. Failures of the hosts acting as leaders can be managed with checkpoint/re-start solutions. Indeed, the leader is one of the hosts participating in the allreduce and, if it fails, its data is lost and cannot be recovered as it happens in the case of a switch/link failure. This, however, is also true for any allreduce algorithm (both host-based or in-network) in case of the failure of one of the hosts involved in the reduction. Alternatively, the leader could run on a server not used by any host (e.g., in the SDN controller(s)). However, this would pose scalability challenges in case of multiple in-network allreduce issued by different jobs, and would not allow load balancing between different leaders (see Section 3.1.4).

Fragmentation. We assume that application-level messages are split into IP packets (see Section 3.1.3), each with its own Canary header. We enforce packets (including Canary header) to be no larger than MTU to avoid fragmentation, which would significantly complicate the design.

Other topologies. For the sake of simplicity, we described and evaluated our algorithm on fat tree topologies. However, a similar approach could be used on other topologies, since an aggregation tree is naturally formed when sending packets from the hosts to the root.

Retransmission delays. If we assume a retransmission timeout of $2 \cdot RTT$ (where RTT is the *Round Trip Time* between a host and the leader), in the worst case a new reduction for a given block will be re-issued after $3 \cdot RTT$. Indeed, a host needs $2 \cdot RTT$ before issuing a retransmission request. The retransmission request arrives at the leader after $RTT/2$, and broadcasts to all the hosts a failure message for that block, that the hosts receive after $RTT/2$.

7. Conclusion

In this work, we designed, implemented, and evaluated Canary, the first congestion-aware in-network allreduce algorithm. We first shown the impact that network congestion can have on the performance of in-network allreduce algorithm, up to the point where they exhibit lower performance than host-based allreduce. For this reason, by relying on timeouts, Canary can dynamically route packets to avoid congested links, aggregating them in a best-effort fashion.

We carefully partitioned Canary functionalities between hosts and switches, and we described a prototype P4 implementation, that we evaluated on a Tofino switch. We then simulated our solution on a 1024 nodes network with 64 switches, showing improvements up to 2x compared to in-network solutions using a single reduction tree and up to 47% compared to solutions using multiple trees. Furthermore, we have shown that these results are consistent across different congestion intensities, proving that Canary is an effective solution in avoiding congestion when running in-network allreduces.

CRediT authorship contribution statement

Daniele De Sensi: Conceptualization, Methodology, Software, Validation, Investigation, Writing – original draft, Visualization. **Edgar Costa Molero:** Methodology, Software, Validation, Investigation, Writing – review & editing. **Salvatore Di Girolamo:** Methodology, Software, Writing – review & editing. **Laurent Vanbever:** Resources, Writing – review & editing. **Torsten Hoefler:** Conceptualization, Methodology, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We would like to thank Vladimir Gurevich for the helpful comments and feedback. This work has been partially funded by the European Union's Horizon Europe programme project RED-SEA (grant no. 955776). Daniele De Sensi was supported by an ETH Postdoctoral Fellowship, Switzerland (19-2 FEL-50), and by Sapienza University, Italy under the SEED-2022 funding scheme.

References

[1] T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, ACM Comput. Surv. 52 (4) (2019) http://dx.doi.org/10.1145/3320060.

[2] S. Gottlieb, W. Liu, W. Toussaint, R. Renken, R. Sugar, Hybrid-molecular-dynamics algorithms for the numerical simulation of quantum chromodynamics, Phys. Rev. D 35 (8) (1987) 2531–2542, http://dx.doi.org/10.1103/PhysRevD.35.2531.

[3] M. Frigo, S.G. Johnson, The design and implementation of FFTW3, Proc. IEEE 93 (2) (2005) 216–231, http://dx.doi.org/10.1109/JPROC.2004.840301.

[4] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D.R.K. Ports, P. Richtárik, Scaling distributed machine learning with in-network aggregation, in: Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 21, 2021.

[5] B. Klenk, N. Jiang, G. Thorson, L. Dennison, An in-network architecture for accelerating shared-memory multiprocessor collectives, 2020, http://dx.doi.org/10.1109/ISCA45697.2020.00085.

[6] Nvidia, NVIDIA ampere architecture in-depth, https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

[7] InfiniBand Trade Association, InfiniBand roadmap – Charting speeds for future needs, https://www.infinibandta.org/infiniband-roadmap-charting-speeds-for-future-needs/.

[8] F. Seide, H. Fu, J. Droppo, G. Li, D. Yu, 1-bit stochastic gradient descent and application to data-parallel distributed training of speech DNNs, in: Interspeech 2014, 2014.

[9] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, T. Hoefler, Sparcml: High-performance sparse communication for machine learning, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3295500.3356222.

[10] T. Hoefler, A. Lumsdaine, W. Rehm, Implementation and performance analysis of non-blocking collective operations for MPI, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07, Association for Computing Machinery, New York, NY, USA, 2007, http://dx.doi.org/10.1145/1362622.1362692.

[11] X. Lian, W. Zhang, C. Zhang, J. Liu, Asynchronous decentralized parallel stochastic gradient descent, in: J.G. Dy, A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, ICML 2018, StockholmsmäSsan, Stockholm, Sweden, July 10–15, 2018, in: Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 3049–3058, URL http://proceedings.mlr.press/v80/lian18a.html.

[12] S. Li, T. Ben-Nun, S.D. Girolamo, D. Alistarh, T. Hoefler, Taming unbalanced training workloads in deep learning with partial collective operations, in: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 45–61, http://dx.doi.org/10.1145/3332466.3374528.

[13] J. Geng, D. Li, Y. Cheng, S. Wang, J. Li, HiPS: Hierarchical parameter synchronization in large-scale distributed machine learning, in: Proceedings of the 2018 Workshop on Network Meets AI & ML, NetAI '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–7, http://dx.doi.org/10.1145/3229543.3229544.

[14] S. Wang, J. Geng, D. Li, Impact of synchronization topology on DML performance: Both logical topology and physical topology, IEEE/ACM Trans. Netw. 30 (2) (2022) 572–585, http://dx.doi.org/10.1109/TNET.2021.3117042.

[15] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, M. Swift, ATP: In-network aggregation for multi-tenant learning, in: 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 21, USENIX Association, 2021, pp. 741–761, URL https://www.usenix.org/conference/nsdi21/presentation/lao.

[16] R.L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, E. Zahavi, Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction, in: Proceedings of COM-HPC 2016: 1st Workshop on Optimization of Communication in HPC Runtime Systems - Held in Conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis, Institute of Electrical and Electronics Engineers Inc., 2017, pp. 1–10, http://dx.doi.org/10.1109/COMHPC.2016.006.

[17] P. Patarasuk, X. Yuan, Bandwidth optimal all-reduce algorithms for clusters of workstations, J. Parallel Distrib. Comput. 69 (2) (2009) 117–124, http://dx.doi.org/10.1016/j.jpdc.2008.09.002, URL http://www.sciencedirect.com/science/article/pii/S0743731508001767.

[18] N. Gebara, P. Costa, M. Ghobadi, PANAMA: In-network aggregation for shared machine learning clusters, in: Conference on Machine Learning and Systems, MLSys 2021, 2021, URL https://www.microsoft.com/en-us/research/publication/panama-in-network-aggregation-for-shared-machine-learning-clusters/.

[19] R.L. Graham, L. Levi, D. Burredy, G. Bloch, G. Shainer, D. Cho, G. Elias, D. Klein, J. Ladd, O. Maor, A. Marelli, V. Petrov, E. Romlet, Y. Qin, I. Zemah, Scalable hierarchical aggregation and reduction protocol (SHARP)TM streaming-aggregation hardware design and evaluation, in: Lecture Notes in Computer Science, Vol. 12151 LNCS, Springer, 2020, pp. 41–59, Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics.

[20] D. De Sensi, T. De Matteis, K. Taranov, S. Di Girolamo, T. Rahn, T. Hoefler, Noise in the clouds: Influence of network performance variability on application scalability, Proc. ACM Meas. Anal. Comput. Syst. 6 (3) (2022) http://dx.doi.org/10.1145/3570609.

[21] S. Jha, A. Patke, J. Brandt, A. Gentile, B. Lim, M. Showerman, G. Bauer, L. Kaplan, Z. Kalbarczyk, W. Kramer, R. Iyer, Measuring congestion in high-performance datacenter interconnects, in: 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 20, USENIX Association, Santa Clara, CA, 2020, pp. 37–57, URL https://www.usenix.org/conference/nsdi20/presentation/jha.

[22] D. De Sensi, S. Di Girolamo, K.H. McMahon, D. Roweth, T. Hoefler, An in-depth analysis of the slingshot interconnect, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20, IEEE Press, 2020.

[23] D. De Sensi, S. Di Girolamo, T. Hoefler, Mitigating network noise on dragonfly networks through application-aware routing, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3295500.3356196.

[24] T. Groves, Y. Gu, N.J. Wright, Understanding performance variability on the aries dragonfly network, in: 2017 IEEE International Conference on Cluster Computing, CLUSTER, 2017, pp. 809–813, http://dx.doi.org/10.1109/CLUSTER.2017.76.

[25] Nvidia, Mellanox Onyx User Manual, https://docs.nvidia.com/networking/pages/viewpage.action?pageId=15046723.

[26] Nvidia, End-to-End QoS configuration for mellanox switches (SwitchX) and adapters, https://enterprise-support.nvidia.com/s/article/end-to-end-qos-configuration-for-mellanox-switches--switchx--and-adapters.

[27] C.E. Leiserson, Fat-trees: Universal networks for hardware-efficient supercomputing, IEEE Trans. Comput. C-34 (10) (1985) 892–901, http://dx.doi.org/10.1109/TC.1985.6312192.

[28] A. Agrawal, C. Kim, Intel Tofino2 – a 12.9Tbps P4-programmable ethernet switch, in: 2020 IEEE Hot Chips 32 Symposium, HCS, IEEE Computer Society, Los Alamitos, CA, USA, 2020, pp. 1–32, http://dx.doi.org/10.1109/HCS49909.2020.9220636, URL https://doi.ieeecomputersociety.org/10.1109/HCS49909.2020.9220636.

[29] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, R. Rajamony, The PERCS high-performance interconnect, in: Proceedings - 18th IEEE Symposium on High Performance Interconnects, HOTI 2010, 2010, pp. 75–82, http://dx.doi.org/10.1109/HOTI.2010.16.

[30] B. Alverson, E. Froese, L. Kaplan, D. Roweth, Cray® XC™ Series Network, Tech. Rep., 2012.

[31] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, T. Inoue, The tofu interconnect d, in: 2018 IEEE International Conference on Cluster Computing, CLUSTER, 2018, pp. 646–654, http://dx.doi.org/10.1109/CLUSTER.2018.00090.

[32] J. Fei, C.-Y. Ho, A.N. Sahu, M. Canini, A. Sapio, Efficient sparse collective communication and its application to accelerate distributed deep learning, in: Proceedings of SIGCOMM'21, 2021.

[33] K. Lakhotia, F. Petrini, R. Kannan, V. Prasanna, In network reductions on a multi dimensional HyperX, in: IEEE 28th Annual Symposium on High-Performance Interconnects, HOTI 2021, August 18–20, 2021, 2021.

[34] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, T. Hoefler, Flare: Flexible in-network allreduce, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21, 2021.

[35] M. Besta, M. Schneider, M. Konieczny, K. Cynk, E. Henriksson, S. Di Girolamo, A. Singla, T. Hoefler, FatPaths: Routing in supercomputers and data centers when shortest paths fall short, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, 2020.

[36] C. Hopps, Analysis of an equal-cost multi-path algorithm, 2009, Request for Comments, RFC 2992, RFC Editor, URL https://www.ietf.org/rfc/rfc2992.txt.

[37] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V.T. Lam, F. Matus, R. Pan, N. Yadav, G. Varghese, CONGA: Distributed congestion-aware load balancing for datacenters, in: Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 503–514, http://dx.doi.org/10.1145/2619239.2626316.

[38] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, T. Edsall, Let it flow: Resilient asymmetric load balancing with flowlet switching, in: 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 17, USENIX Association, Boston, MA, 2017, pp. 407–420, URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini.

[39] L.G. Valiant, A scheme for fast parallel communication, SIAM J. Comput. 11 (2) (1982) 350–361, http://dx.doi.org/10.1137/0211027.

[40] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, A. Vahdat, Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network, SIGCOMM Comput. Commun. Rev. 45 (4) (2015) 183–197, http://dx.doi.org/10.1145/2829988.2787508.

[41] S. Ghorbani, Z. Yang, P.B. Godfrey, Y. Ganjali, A. Firoozshahian, DRILL: Micro load balancing for low-latency data center networks, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 225–238, http://dx.doi.org/10.1145/3098822.3098839.

[42] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, D. Maltz, Per-packet load-balanced, low-latency routing for clos-based data center networks, in: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 49–60, http://dx.doi.org/10.1145/2535372.2535375.

[43] L. Shalev, H. Ayoub, N. Bshara, E. Sabbag, A cloud-optimized transport protocol for elastic and scalable HPC, IEEE Micro 40 (6) (2020) 67–73, http://dx.doi.org/10.1109/MM.2020.3016891.

[44] Microsoft, Scaling HPC applications – Adaptive routing, https://docs.microsoft.com/en-us/azure/virtual-machines/workloads/hpc/compiling-scaling-applications#adaptive-routing.

[45] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, P. Kalnis, In-network computation is a dumb idea whose time has come, in: Proceedings of the 16th ACM Workshop on Hot Topics in Networks, in: HotNets-XVI, Association for Computing Machinery, New York, NY, USA, 2017, pp. 150–156, http://dx.doi.org/10.1145/3152434.3152461.

[46] J. Meza, T. Xu, K. Veeraraghavan, O. Mutlu, A large scale study of data center network reliability, in: Proceedings of the Internet Measurement Conference 2018, IMC '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 393–407, http://dx.doi.org/10.1145/3278532.3278566.

[47] R. Singh, M. Mukhtar, A. Krishna, A. Parkhi, J. Padhye, D. Maltz, Surviving switch failures in cloud datacenters, SIGCOMM Comput. Commun. Rev. 51 (2) (2021) 2–9, http://dx.doi.org/10.1145/3464994.3464996.

[48] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, R. Agarwal, Understanding host network stack overheads, in: Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 65–77, http://dx.doi.org/10.1145/3452296.3472888.

[49] M.T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, D. Wentzlaff, Enabling programmable transport protocols in high-speed NICs, in: 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 20, USENIX Association, Santa Clara, CA, 2020, pp. 93–109, URL https://www.usenix.org/conference/nsdi20/presentation/arashloo.

[50] S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beranek, L. Benini, T. Hoefler, A RISC-V in-network accelerator for flexible high-performance low-power packet processing, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture, ISCA, 2021.

[51] W. Schonbein, R.E. Grant, M.G.F. Dosanjh, D. Arnold, INCA: In-network compute assistance, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3295500.3356153.

[52] S.D. Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, T. Hoefler, Network-accelerated non-contiguous memory transfers, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC19, 2019.

[53] Linux Foundation, DPDK: Data plane development kit, https://www.dpdk.org/.

[54] S. Chunduri, S. Parker, P. Balaji, K. Harms, K. Kumaran, Characterization of MPI usage on a production supercomputer, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, IEEE Press, 2018, http://dx.doi.org/10.1109/SC.2018.00033.

[55] K. Pagiamtzis, A. Sheikholeslami, Content-addressable memory (CAM) circuits and architectures: a tutorial and survey, IEEE J. Solid-State Circuits 41 (3) (2006) 712–727, http://dx.doi.org/10.1109/JSSC.2005.864128.

[56] N. Feamster, J. Rexford, E. Zegura, The road to SDN: An intellectual history of programmable networks, SIGCOMM Comput. Commun. Rev. 44 (2) (2014) 87–98, http://dx.doi.org/10.1145/2602204.2602219.

[57] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L.E. Li, M. Thottan, Measuring control plane latency in SDN-enabled switches, in: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Association for Computing Machinery, New York, NY, USA, 2015, http://dx.doi.org/10.1145/2774993.2775069.

[58] N.K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, S. Peter, Evaluating the power of flexible packet processing for network resource allocation, in: 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 17, USENIX Association, Boston, MA, 2017, pp. 67–82, URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/sharma.

[59] S. Ibanez, G. Antichi, G. Brebner, N. McKeown, Event-driven packet processing, in: Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 133–140, http://dx.doi.org/10.1145/3365609.3365848.

[60] Z. Hang, M. Wen, Y. Shi, C. Zhang, Programming protocol-independent packet processors high-level programming (P4HLP): Towards unified high-level programming for a commodity programmable switch, Electronics 8 (9) (2019) http://dx.doi.org/10.3390/electronics8090958, URL https://www.mdpi.com/2079-9292/8/9/958.

[61] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, G. Carle, MoonGen: A scriptable high-speed packet generator, in: Internet Measurement Conference 2015, IMC'15, Tokyo, Japan, 2015.

[62] A.F. Rodrigues, K.S. Hemmert, B.W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, B. Jacob, The structural simulation toolkit, SIGMETRICS Perform. Eval. Rev. 38 (4) (2011) 37–42, http://dx.doi.org/10.1145/1964218.1964225.

[63] Sandia National Laboratories, The structural simulation toolkit, http://sst-simulator.org/.

[64] T. Hoefler, T. Schneider, A. Lumsdaine, Characterizing the influence of system noise on large-scale applications by simulation, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, USA, 2010, pp. 1–11, http://dx.doi.org/10.1109/SC.2010.12.

[65] R. Pagh, F.F. Rodler, Cuckoo hashing, J. Algorithms 51 (2) (2004) 122–144, http://dx.doi.org/10.1016/j.jalgor.2003.12.002.

[66] L.J. Guibas, E. Szemeredi, The analysis of double hashing, J. Comput. System Sci. 16 (2) (1978) 226–274, http://dx.doi.org/10.1016/0022-0000(78)90046-6, URL https://www.sciencedirect.com/science/article/pii/0022000078900466.

[67] Y. Yuan, O. Alama, J. Fei, J. Nelson, D.R.K. Ports, A. Sapio, M. Canini, N.S. Kim, 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, 2022, pp. 683–700, URL https://www.usenix.org/conference/nsdi22/presentation/yuan.

Daniele De Sensi is an Assistant Professor at Sapienza University of Rome. He was previously a postdoctoral researcher with the Scalable Parallel Computing Laboratory, ETH Zurich, and at the University of Pisa, where he received the Ph.D. degree in 2018. He has authored and coauthored around 40 papers on high-performance interconnection networks, parallel computing, and power-aware computing.

Edgar Costa Molero is a final-year Ph.D. student in the Networked Systems Group at ETH Zürich, Switzerland. His research focuses on the management and development of the next generation of networks using programmable devices. He has worked on traffic engineering and anomaly detection in both legacy and SDN-based datacenter networks. He has also contributed to several publications and supervised various theses in his field. He holds a bachelor's degree in Information and Communication Technology from the Polytechnic University of Catalonia, Spain, and a master's degree in Electrical Engineering and Information Technology from ETH Zürich.

Salvatore Di Girolamo received the Ph.D. degree from ETH Zurich in 2021, where he worked on programmable network infrastructures. He is currently a senior software architect at NVIDIA. His research interests include scalable networks and parallel and distributed computing.

Laurent Vanbever is an associate professor at ETH Zürich and the leader of the Networked Systems Group (NSG). His research focuses on network programmability and Internet routing, with the goal of making computer networks more performant and easier to manage. He has a Ph.D. in computer science from the University of Louvain. He has published numerous papers in prestigious venues such as ACM SIGCOMM, USENIX NSDI, and NeurIPS.

Torsten Hoefler is a Professor of Computer Science at ETH Zurich, an Academia Europaea member, and an ACM and IEEE Fellow. His research interests revolve around ''Performance-centric System Design'' and include scalable networks, parallel programming techniques, and performance modeling. Torsten won best paper awards at SC10, SC13, SC14, SC19, SC22, EuroMPI'13, HPDC'15, HPDC'16, IPDPS'15, and others. He published numerous papers and authored chapters of the MPI-2.2 and MPI-3.0 standards. He received the IEEE CS Sidney Fernbach Award, the ACM Gordon Bell Prize, the ISC Jack Dongarra Award, the ETH Zurich Latsis prize, and both ERC starting and consolidator grants.