Integration and QoS of Multicast Traffic in a Server-Rack Fabric with 640 100G Ports

Nikolaos Chrysos, Fredy Neeser
IBM Research–Zurich
{cry,nfd}@zurich.ibm.com

Brian Vanderpool, Mark Rudquist
IBM STG, Rochester, USA
{vanderp,rudquist}@us.ibm.com

Kenneth Valk, Todd Greenfield, Claude Basso
IBM STG, Rochester, USA
{kmvalk,toddg}@us.ibm.com
basso2@fr.ibm.com

ABSTRACT

Flexible datacenters rely on high-bandwidth server-rack fabrics to allocate their distributed computing and storage resources anywhere, anyhow, and anytime demanded. We describe the multicast architecture of a distributed server-rack fabric, which is arranged around a spine-leaf topology and connects 640 Ethernet ports running at 100G. To cope with the immense fabric speed, we resort to hierarchical, tree-based replication, facilitated by specially commissioned fabric-end ports. At each (port-to-port) leg of the tree, a frame copy is forwarded after a request-grant admission phase and is ACKed by the receiver. To save on bandwidth, we use a packet cache in our input-queued switching-nodes, which replicates asynchronously forwarded frames thus tolerating the variable-delay in the admission phase. Because the cache has limited size, we loosely synchronize the multicast subflows to protect the cache from thrashing. We describe our policies for lossy classes, which segregate and provide fair treatment to multicast subflows. Finally, we show that industry-standard Level2 congestion control does not adapt well to one-to-many flows, and demonstrate that the methods that we implement achieve the best performance.

Categories and Subject Descriptors

C.2.1 [**Computer-communication networks**]: Network Architecture and Design

Keywords

Datacenter fabrics; server-rack interconnects; multicast

1. INTRODUCTION

Flexible performance-optimized datacenters (PoDs) rely on high-bandwidth server-rack fabrics that expedite the on-demand allocation of the available but distributed computing and storage capacity. Such versatile systems can be dynamically tailored to the specific needs of tenants or applications. At the same time, the datacenter users can capitalize on network support for one-to-many (multicast) communication. Relevant enterprise and high-performance computing applications include provisioning of financial servers, caching of critical or popular data, n-redundant file stores, publish-subscribe services, one-to-many collectives, all of which push data to large numbers of receivers simultaneously. Efficient network-level multicast transmission tapers off the server load per message and moderates the network utilization.

In this paper we describe the multicast architecture of a distributed server-rack fabric, based on a *flattened spine-leaf topology* with 640 Ethernet ports at 100G. The provision of sophisticated traffic management at the targeted scale demands prudent and intelligent architectures. Our system concentrates the frame processing, replication, and scheduling functions at the network edges (leaf switches), which are integrated inside the racks. This decision drastically simplifies the spines, which in our design are ultra-high-radix, shallow-buffered packet switches that enable dense high-bandwidth connectivity.

Multicast traffic dramatically escalates the processing and forwarding rates at fabric ports. In our implementation, a port can process and forward a new (unicast) frame every 6.6 ns (the equivalent of 64B at 100G). Effectively, for 512B frames or larger, a port can fanout to 8 peers at full 100G speed. For system-level multicast groups, consisting of several hundreds of listeners, we do (non-router-assisted [1]) hierarchical multicasting in hardware, which seamlessly integrates multicast with unicast processing, and narrows the ACK implosion problem. The replication nodes are drawn from a pool of specially commissioned fabric-end ports (*surrogates*), thus releasing multicast destinations (user or terminal ports) from the extra load. There is one surrogate port per leaf switch, receiving frames from the fabric and forwarding copies to local multicast destination ports and to remote surrogate ports, deeper in the hierarchy. The fabric employs credit-based flow control, from ingress to egress, to prevent queue overflows. To recover from soft errors anywhere along the path, CRC, sequence numbers, and unicast retransmissions on the failing port-pair leg are utilized.

We denote the number of terminal destinations of a multicast flow (i.e., the cardinality of its fanout set, F) as $|F|$. As seen at a particular (source or surrogate) port, x, the *local fanout set*, $f(x)$, of a multicast flow consists of a combination of destinations ($\in F$) and surrogate ports. The port forwards the multicast flow by spawning $|f(x)|$ unicast-oriented *subflows*.

ANCS'14, October 20–21, 2014, Los Angeles, CA, USA.
Copyright 2014 ACM 978-1-4503-2839-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2658260.2658278.

To forward a unicast frame from one port to another, we use a reliable, distributed, *request-grant/credit-ACK* protocol. To transmit a frame, a port must first request and be granted credits for the egress buffer at the target leaf switch. Effectively, we forward the $|f|$ multicast copies of a frame asynchronously from one another—i.e., fanout or call splitting [2, 3, 4]. On the other hand, typical input-queued crossbars achieve line rate only when all copies are sent at the same time. In our design, we achieve line rate forwarding with a *multicast packet cache* that replicates the asynchronously forwarded copies. The packet cache is located in front of the crossbar of edge switches, and can use up to eight crossbar ports to simultaneously forward eight multicast subflows at 100G. Because the cache has limited size, it may have to evict useful packets. We loosely synchronize the *Virtual Output Queues (VOQs)* that hold the copies of a multicast flow to ward off such unwanted evictions.

A flow control action is imminent when a frame arrives at a source port or when a frame loops back from the egress to the ingress side of a surrogate port. If the frame belongs to a lossy traffic class, we may decide to drop it when we have run out of payload buffers. But besides the payload, the *multicast headers* that we maintain for frame copies consume space. To prevent a congested subflow, which makes slow progress, from consuming too many headers, we may also reject frame copies that target congested destinations, using a per-VOQ drop. We have opted for an advantageous drop-from-head policy, which overcomes the negative interactions between tail-drop and VOQ synchronization. In addition, we randomize the drops, in a random-early-detection fashion, to maintain fairness across the subflows.

If the arriving frame belongs to a lossless traffic class, the port may issue a per-priority PAUSE message to avoid losses. At ingress ports, these messages are standard Ethernet PAUSEs that hold off the upstream network interface from sending new frames from the indicated priority. At surrogate ports, PAUSEs are fabric-internal messages, that nevertheless perform a similar function: they prevent the egress side of the surrogate, which receives fabric frames, from forwarding new frames to the ingress side.

With lossless traffic classes, proper congestion management is a key enabling technology. Without it, a congested multicast flow can choke the fabric (at a given priority traffic class). First, we demonstrate that the industry-standard method, *Quantized Congestion Notifications (QCN)*, does not work properly with one-to-many multicast flows. Then we demonstrate that our congestion management architecture, first presented and evaluated for unicast traffic in [5], which *(a)* places the QCN congestion points at fabric inputs instead of fabric outputs, and *(b)* modifies the standard-QCN marking scheme, works smoothly with multicast flows.

Our contributions are the following:

- We report the seamless integration of end-to-end reliable multicasting in a massive 100G Ethernet server-rack fabric, implemented in 32 nm technology.
- We *(i)* propose a packet caching scheme that extends the multicast capabilities of input-queued crossbars by replicating *delayed* frame copies, and *(ii)* describe a productive subflow synchronization scheme.
- We present efficient flow control schemes for lossy multicast classes, demonstrate that industry-standard QCN does not work well with one-to-many flows, and propose a solution that mitigates the shortcomings of QCN.
- We evaluate the performance of our implementation using several micro-benchmarks, full-system broadcasts, as well as uniformly-destined, large multicast trees.

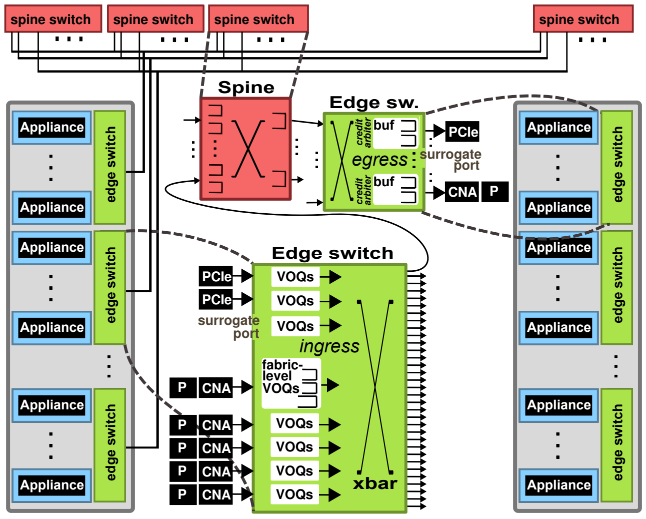


Figure 1: A distributed server-rack fabric connecting the appliances in multiple racks (only two racks are shown). The fabric has input (ingress) and output (egress) buffers at the network-edge switches, and small packet buffers at the spines. One port in every edge switch is a surrogate, used in hierarchical multicast replication.

The remainder is structured as follows. In Sec. 2, we present the underlying server-rack fabric, and the port-level forwarding of multicast traffic. Continuing with Sec. 3, we describe the crossbar packet cache and the VOQs synchronization mechanisms. Section 4 describes our congestion management for lossy and lossless traffic classes. Section 5 presents the formation of system-level replication trees using surrogate ports. Computer simulations throughout these sections are used to evaluate the performance of the system and to demonstrate important trade-offs. Finally, we discuss our key design points in Sec. 6, and we conclude in Sec. 7.

2. FABRIC & FRAME PROCESSING

In this section we present the fabric topology, and outline how multicast processing fits in the picture.

2.1 Server-rack fabric

In this paper we set out to enable efficient multicast traffic support in the holistically-designed server-rack fabric shown in Fig. 1, which provides 640 100G Ethernet and 256 PCIe Gen3 (x8) ports. Abstractly seen, the fabric is a *large switch* built around a spine-leaf (fat-tree) topology. The main bulk of memory storage is provided at fabric-input and fabric-output (or ingress and egress) buffers in leaf switches, in a similar fashion with *Combined Input-Output Queued (CIOQ)* switches.

The leaf switches are integrated into the backplane of a cluster of server racks. Every leaf constitutes the network-edge point for five (5) servers, offering 100 Gb/s bandwidth to each on a single Ethernet link. Additionally it offers one surrogate and two PCIe ports, for a total of eight (8) main ports. Shown in Fig. 1 are also the *Converged Network Adapters (CNAs)*, which, for Ethernet traffic, constitute the interface between the servers (P) and the fabric ports.

The leaf (or edge) switches, coordinated by a central control unit, are responsible for MAC learning, frame replication and forwarding, thus collectively acting as a distributed, high-capacity bridge. Only Ethernet ports participate in multicast communications. The forwarding tables of the fabric are populated by snooping the multicast-group conversations between hosts and routers that are carried through Internet Group Management Protocol (IGMP) messages.

At its ingress side, an edge switch stores the incoming frames in fabric-level (i.e., network-level) VOQs, segments the frames into variable-size fabric-internal packets with a size up to 256B, and injects the packets into the interconnect. The journey of a packet (or cell) inside the interconnect sets off at a crossbar in the sourcing edge switch, goes through a spine, and terminates at the crossbar of the target edge switch. A packet can use any of the available edge-to-spine links, as enforced by a packet-level spraying routing mechanism that overcomes the limitations of Equal-Cost-Multi-Path (ECMP) alternatives [6]. Link-level retry is used when traveling between leaf and spine switches. The original frames are reassembled at their target edge switch, in egress buffers, and are forwarded to their destination in order. (At surrogate ports, the forwarded frames are looped back to the ingress side of the surrogate.) The source ports receive an acknowledgment for each such forwarded frame to release the ingress memory occupied. If a frame experiences a soft error, its source port will retransmit it after a timeout.

Internally, the fabric uses hop-by-hop credit-based flow control, thus obviating buffer overflows. The egress buffers are flow controlled using a scheduled, port-to-port, credit protocol: to inject a frame inside the fabric, a VOQ must first request and be granted credits from the output-port arbiter located at the target edge switch. The arbiter hands out credits, which correspond to packet slots (buffer units) in its local egress memory, using a variant of deficit round-robin. The per-VOQ requests, grants, and ACKs, are 10-byte messages, using the same links as data.

As shown in Fig. 1, the crossbar inside edge switches has a dedicated 130 Gb/s interface (36 Bytes at 454 MHz) for every main port. As described in Sec. 3, an Ethernet or surrogate port can combine all these crossbar interfaces, eight in total, to fanout to eight (8) new ports at 100G. The crossbar additionally provides eight (8) 130 Gb/s interfaces, each attached to four (4) optical links that connect to the spines at 25 Gb/s—please observe the crossbar-internal speedup.

The spines are cell-based, CIOQ *switching elements*, with small input and output buffers (16 packets per port), oblivious of the higher-level protocols. They reside in separate chassis, and provide 136 25 Gb/s bidirectional ports that can be arbitrarily connected with leaf switches. In the configuration that we consider, there are 128 edge switches, with 32 links, each connected to one of 32 available spines. Thus, for Ethernet traffic, this configuration features an over-provisioning ratio of 8:5 (=32·25:5·100), which accommodates the internal overheads, leaving some headroom to compensate for scheduling inefficiencies.

The fabric *has been implemented* using 32 nm technology, in 19.7x19.7 mm^2 for the leaf nodes and 18.4x18.4 mm^2 for the spines. The ingress and egress payload buffers in edge switches use inexpensive on-chip EDRAMs and are partitioned in buffer pools, per port and per *virtual lane (VL)*.

Table 1: System parameters

	Parameter	*Value*
Ingress terminal	shared buf. T_{th} (lossy)	185 KB
Ingress terminal	per-VL buf. (lossless)	227 KB
Ingress surrogate	shared buf. (lossy)	84 KB
Ingress surrogate	per-VL buf. (lossless)	32 KB
Egress terminal	per-VL buf.	140 KB
Egress surrogate	per-VL buf.	100 KB
PAUSE (ingress)	STOP Q_{hi} (lossless)	160 KB
PAUSE (ingress)	GO Q_{low} (lossless)	90 KB
Term. VOQ drop	V_{th} (lossy)	60 KB
Surr. VOQ drop	V_{th} (lossy)	50 KB
Header pool high	per-VL H_{hi} (lossy)	500 hdrs
Header pool low	per-VL H_{low} (lossy)	470 hdrs
Header pool	per-VL (lossless)	291 hdrs
Multicast cache	number of entries C	128 packets
VOQ sync	block threshold	18 packets
VOQ sync	resume threshold	12 packets
QCN (ingress)	Q_{eq} (queue equil.)	90 KB
QCN (egress)	Q_{eq} (queue equil.)	50 KB
QCN	w (weight for ΔQ [7])	2
QCN	I_s (sampling interval)	150 KB

Note that each VL corresponds to a single Ethernet priority level, configured to provide either lossy or lossless service. Lossy VLs share their payload buffers at the ingress. For lossless traffic classes, each VL is allocated private buffers, and the ingress terminal port issues a (per priority level) PAUSE message to its upstream CNA when the aggregate occupancy of the VOQs in the corresponding VL exceeds a threshold value, Q_{hi}. Furthermore, the CNAs implement QCN rate limiters, which throttle the injections of an Ethernet flow in response to received congestion notification messages. The fabric has QCN congestion points at both fabric outputs and fabric inputs.

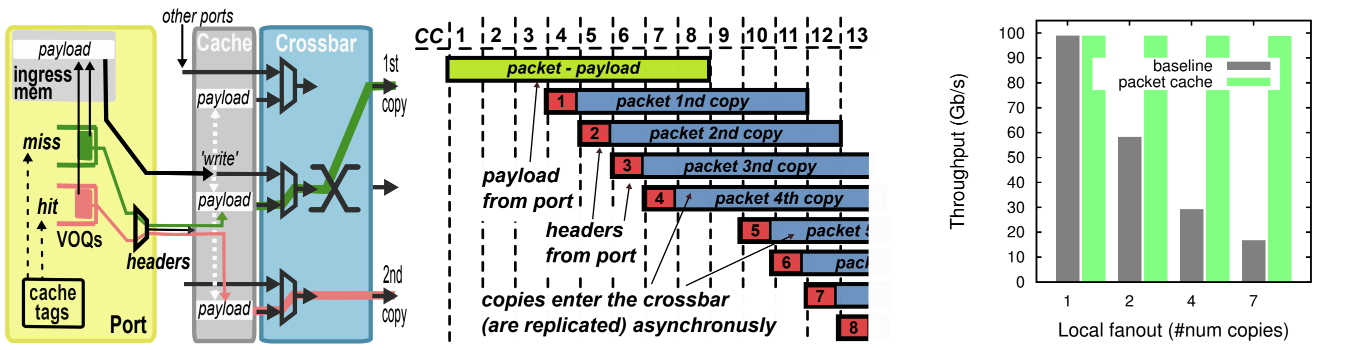
Although this switching fabric may look far more complicated than a standalone switch module, its port-to-port scheduled flow control, coupled with its true multi-path routing and its internal over-provisioning, make it behave as a large, fair, CIOQ switch.

Small-scale tests have been performed on real hardware. In this paper, we report performance results from clock-cycle accurate simulations. Table 1 summarizes the key simulation parameters. We consider 40-meter links between the spines and leaf switches—a full deployment of our system may use even smaller links.

2.2 Queuing and scheduling of multicast frames

Frame replication and forwarding are executed at fabric-input and surrogate ports. Surrogates look and act like other ports, but they neither send nor accept external traffic. Instead, they loopback the frames that they receive from other source or surrogate ports, replicating them for local and remote destinations. The multicast headers of these copies are stored in the VOQs at the ingress side of the surrogate.

Receiving a multicast frame, a port looks up its MAC address in a local Forwarding Database (FDB) to obtain



(a) Multicast cache (b) Parallel forwarding of copies (c) Throughput with and without the multicast cache

Figure 2: (a) Replication of packets using the multicast cache and the input-queued crossbar. The cache is first instructed to write the packet payload, and immediately receives the first header. The second copy, transmitted at an arbitrary point in time, does not use the payload interface. Instead, only its header is sent to the cache, which combines it with the stored payload and transfers the copy to the target fabric port. The cache can use multiple crossbar input ports to forward packet copies in parallel. (b) The 256B payload is transmitted from the port in 8 cycles. The header for the first copy always travels 3 clock cycles behind the payload. Additional copies are forwarded by issuing one header per cycle. Effectively, up to eight 256B packet copies can be transmitted every eight cycles. In the figure, the headers for copies 5 to 8 delay for a couple of cycles, to demonstrate the asynchronous replication of copies using the multicast cache. (c) Throughput of a 99 Gb/s multicast flow for varying local fanout $|f|$.

a Multicast ID (MID), which is then used to access a local multicast/broadcast table (MCBC). The MCBC outputs the local fanout set of the frame. Effectively, following the payload of a frame, a port additionally receives one *multicast header* per clock cycle, identifying the surrogates and destinations that are waiting for a copy. To a large extent, every incoming header is treated as a new *unicast frame* arrival. At most, 8 multicast headers per frame are received at 100G ports.

We adopt a VOQ-based queuing architecture for multicast traffic. The payload of the frame is written into the intended buffer pool, using a single linked list of available (256B) *buffer units*. Every associated multicast header, which now also points to the head buffer unit, is en-queued into the target VOQ. Each VOQ is associated with an egress fabric port and a virtual lane, and may store packet copies from different multicast flows.

After inserting a multicast header in its VOQ, the source (fabric-input or surrogate) port may issue a request to the target port, and will be able to forward a new copy after receiving a grant from the corresponding output arbiter. As with unicast traffic, the receiving port also ACKs the proper receipt of frames to their source port. After all copies of a frame have been acknowledged, the source port releases the space that the frame occupied in its ingress payload pool.

One widely used protocol for multicast in input-queued switches is *one-shot scheduling* (or non-fanout splitting), in which a port forwards all copies of a packet simultaneously, together with the full list of target destinations. In these systems, the switch interconnect is responsible for replicating the packet [8]. This strategy may minimize the bandwidth overhead, but stipulates that all copies will be granted and residing at the *Head Of Line (HOL)* position of their VOQ at the same time.

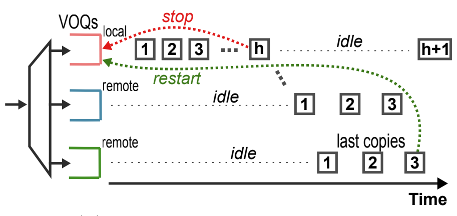
In our distributed, asynchronous system, the copies of a multicast frame experience variable arbitration latencies, and may not all be eligible for transmission at the same time. Furthermore, one-shot scheduling may result in waiting for a grant from a congested port before delivering a frame to ports that are ready to accept it. While this blocking may be anticipated with non-selective (e.g., FIFO) queuing, our VOQ architecture can do better. Therefore, we have opted for a strict (call) fanout splitting, unicast-oriented, delivery mode, where every copy is forwarded independently.

Overall, our VOQ-based multicast queuing organization, in combination with fanout splitting scheduling, is free of HOL-blocking, providing markedly superior performance than plain FIFO [9]. In addition, it uniformly integrates unicast with multicast processing, abating the complexity of the design. At the same time, it seamlessly propagates the benefits of reliable quality-of-service, which we have in place for unicast traffic, to one-to-many flows.

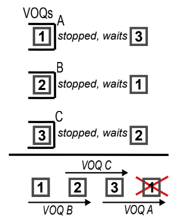
3. ASYNCHRONOUS PACKET REPLICATION

The fanout splitting scheduling that we use would typically require a port-to-fabric bandwidth proportional to the number of local copies ($|f|$) of multicast frames. In our design, the interface of a port to the fabric interconnect is a port-speed line that ends at the corresponding input of the input-queued crossbar inside the edge switch (refer to Fig. 1). Thus, having to send a frame $|f|$ times over this interface, the bandwidth of a 100G flow will drop proportionally. To overcome this limitation, we use a caching mechanism inside the fabric, which stores and replicates packets (frame's segments).

As shown in Fig. 2(a), the edge switch writes the payload of the port-injected packets inside a *multicast cache*. The cache is physically located at the inputs of the crossbar, and is shared among all local Ethernet (and surrogate) ports. Only one port can write a packet into the cache at a time. Each port maintains the *cache tags* that indicate which of its previously sent packets are present in the cache. When it decides to inject a new copy, the port looks up its tags to see



(a) Subflows synchronization



(b) Deadlock freedom

Figure 3: (a) The synchronization mechanism on a multicast flow with 3 copies, for $S_{\text{low}} = S_{\text{hi}} - 2$. The local subflow is suspended after sending the first S_{hi} packets ahead of other subflows. It is restored after the two remote subflows have sent their first 3 copies. (b) A hypothetical but impossible deadlock situation.

if the copy is presently cached. If this is a hit, the port will issue the corresponding multicast header, together with a cache address, instructing the cache to replicate the packet. In case of a cache miss, the port will send the packet payload again. If additional copies are to be sent, the payload can be written in the cache for a second time.

In our implementation, for tags we use the address of the payload at the ingress port memory. The cache lines, which keep the packets' payload, are 256B and mirrored in eight (8) SRAM blocks, one for each crossbar input attached to the local Ethernet, surrogate, and PCIe ports (refer to Fig.1). Each such multicast cache mirror can independently feed a 100 Gb/s subflow into the crossbar.

As shown in Fig.2(b), the time of a 256B packet at a crossbar input (or output) is eight (8) clock cycles. In this time period, a port can issue eight (8) multicast headers, one for each copy. Effectively, the cache can deliver the eight copies of a multicast flow in parallel.

Ideally, the multicast cache will evict a packet after all the copies have been forwarded. However, because the cache has limited space and it is shared among all Ethernet ports in the edge switch, we also use a least-recently-used (LRU) eviction policy. The sourcing port is informed about the premature packet eviction, and forwards the next copy through the payload interface.

Figure 2(c) depicts the throughput of a single multicast flow, sending 1522B frames, as a function of its fanout. As can be seen, without the multicast cache (baseline), the throughput drops linearly with the fanout. In contrast, using the cache, the throughput is at 100G for any fanout.

3.1 Approximate synchronization of subflows

Because the cache is shared among multiple ports, it is important to use it judiciously. The following scenario reveals some of the impending troubles. Consider that we launch a 100G multicast flow that targets three (3) destinations, one local, in the source edge switch, and two remote. Due to the different propagation delays in the request-grant loops of the corresponding subflows, by the time that a remote one receives a first grant, the local subflow may have already forwarded enough frames to fill up the cache.

In this scenario, the granted packets of remote subflows may result in cache misses, limiting the achievable bandwidth by a factor of two. Furthermore, these late packets will compete for cache space with packets of the running-ahead local subflow. With the cache being such wildly thrashed, every copy may have to use the payload interface.

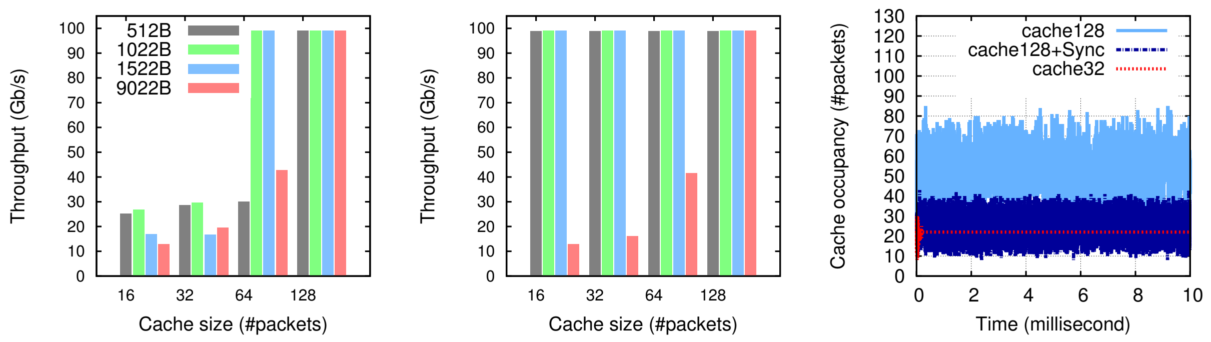
Our first remedy is to overprovision the cache, so that it fits the largest request-to-grant *round-trip time (rtt)* worth of packets, assuming no contention. However, in practice, when a slightly delayed subflow experiences cache misses, it can be pushed even further behind, because, for improved efficacy, the ports prioritize frames that hit in the cache. As a result, even transient small contention can sometimes desynchronize the subflows. In addition, instead of allocating more cache entries for a single flow, it is preferable to enable efficient operation with the least possible space.

In our system, we can loosely synchronize the VOQs that carry copies from the same multicast flow, as a measure against cache thrashing. The method is schematically shown in Fig.3(a). We use a small *VOQ sync CAM* which is searched for VOQ identifiers (13 bits), and keeps a *pkts_ahead_cnt* field. The algorithm starts when a VOQ begins forwarding a frame copy. If this is *not* the last copy of the frame, and the VOQ does not have an entry in the sync CAM already, we allocate a new entry for it. Afterward, we increase the *pkts_ahead_cnt* field in the VOQ entry by the number of packets in the frame. If the *pkts_ahead_cnt* now exceeds threshold S_{hi}, we suspend the corresponding VOQ, waiting for the ones lagging behind to catch up. When a VOQ forwards the last copy of a frame, we identify all VOQs in the sync CAM that have sent this frame already, and decrement their *pkts_ahead_cnts* by the corresponding number of packets. Suspended VOQs whose *pkts_ahead_cnt* now cross threshold S_{low} are restored. Suspended VOQs are also restored after a timeout (3 microseconds), or when their sync CAM entry is released. VOQ entries are released when they are evicted or when their *pkts_ahead_cnts* becomes zero.

Figures 4(a,b) present the receive throughput of a multicast flow, sourced at 99 Gb/s, which targets 4 local and 4 remote destinations, for varying cache sizes, C (16 to 128 packets), and frame sizes (512B to 9022B). Parameters S_{low} and S_{hi} are set to 18 and 10 packets respectively. In our system, the base request-to-grant rtt of local subflows is $\sim$100 ns, and that of remote subflows $\sim$1100 ns, which amounts to 55 256B packets at 100 Gb/s. As can be seen in the figures, without VOQ synchronization, for cache sizes smaller than 64 packets, the throughput can drop below 30 Gb/s because the local subflows fill up the cache before the remote subflows have sent their copies. In contrast, with VOQ synchronization, even a 16-packet cache delivers full throughput, with the exception of large Jumbo (9022B) frames (36 packets), which push the cache stress to the limit.

Besides the actual payload buffers in the cache, which are implemented in dense modern SRAMs [8 (cache mirrors) Cx256x8-bit arrays], a considerable cost is incurred in cache tags. These are 14-bit memory buffer-unit identifiers, stored in latches, requiring a Cx14-bit CAM at each port. As a reasonable trade-off between cost and performance, we sized the cache at $C = 128$ packet entries.

In Fig 4(c), we depict the evolution of cache occupancy in time for 1522B frames. As can be seen, with VOQ synchronization, a 128-packet cache has less than 40 occupied entries for 1522B frames, creating headroom that can be allocated to multicast flows from neighbor ports. At the other extreme, without synchronization, a 32-packet multicast cache fills up—its occupancy is around 20 packets because the cache controller selects one entry with no out-



(a) No VOQs synchronization (b) With VOQs synchronization (c) 1522B frames

Figure 4: (a,b) Throughput of a multicast flow with local fanout 8, heading to 4 local and 4 remote destinations, with varying cache and frame sizes.(c) Cache-occupancy time series.

standing headers to evict in every clock cycle with less than 10 entries available.

Multiple concurrent flows: The synchronization mechanism introduces dependencies between VOQs: A VOQ can be blocked waiting for another one to transmit a frame. When many multicast flows are active at the same port, their subflows can be arbitrarily distributed to VOQs. Effectively, impertinent subflows may block one another. For example, a local subflow, in VOQ A, may be blocked waiting for its sibling remote subflow, thus blocking another VOQ carrying irrelevant subflows. Nevertheless, for ports experiencing a large aggregate fanout, from multiple flows, the synchronization mechanism is automatically disabled, as discussed next. When a new VOQ that sends a frame finds the sync CAM full, we evict one busy entry using an LRU policy. An unexpected benefit comes from these evictions. Because the sync CAM is small (24 entries), at ports with many sending VOQs, it will systematically evict busy entries and unblock those that were suspended. We have verified this behavior in our experiments in Sec 5.2.

Below we further prove that the VOQ dependencies cannot form circles and therefore deadlocks. Consider the circular dependency depicted in Fig 3(b). Three frames are shown: Frames 3 and 1 are sent by VOQ A, frames 1 and 2 by VOQ B, and frames 2 and 3 by VOQ C. These three frames do not necessarily belong to the same multicast flow, and their indexes do not necessarily reflect their arrival order. All VOQs have been suspended by the synchronization mechanism. In particular, for VOQ A to send frame 1, VOQ C must first forward frame 3. Similarly, VOQs B and C are blocked waiting for VOQs A and B to forward frames 1 and 2, respectively. Effectively, a cyclical dependency is formed and the system is in deadlock. However, as shown by the inlet of Fig. 3(b), each VOQ forwards frames in FIFO order. It follows that: According to VOQ B, frame 1 has arrived before frame 2, for C, frame 2 has arrived before 3, and for A, frame 3 has arrived before 1. Because it is impossible to simultaneously satisfy all VOQs' orders, deadlocks cannot occur.

4. FLOW CONTROL AT FABRIC EDGE

So far we have described the forwarding and scheduling of multicast frames at fabric ports. In this section, we turn our attention to the flow control mechanisms, which shape the arriving traffic. Together these mechanisms determine the rates and the backlogs of the multicast subflows.

4.1 Lossy traffic

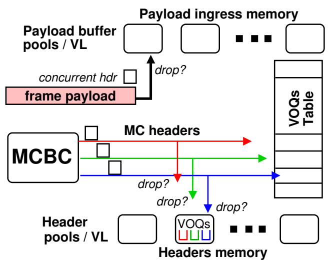
Figure 5(a) depicts a multicast frame arriving at a terminal or surrogate port. Shown are the per VL payload and multicast header pools. The payload is accompanied by a *concurrent header*, which specifies a VL, and thus the intended buffer pool. In lossy traffic classes, the payload is dropped when the occupancy of buffers that are shared by lossy VLs exceeds threshold T_{th}.

If the payload is accepted, the MCBC outputs the multicast headers which carry private VL identifiers and are stored in their corresponding header pools. No headers follow a dropped payload. The VOQ table contains one entry for every active VOQ at the port, with fields such as the outstanding requests, grants, sequence numbers,etc. Additionally every VOQ table entry has a pointer to the memory location that stores the HOL multicast header of the corresponding VOQ. The remaining multicast headers in the same VOQ are connected below, along a single linked list.

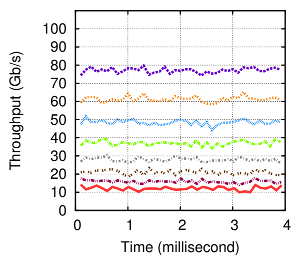
The copies of a single frame may fall at most into two different VLs: one for those targeting a final destination and one for those targeting a surrogate deeper in the hierarchy. For each arriving header, a separate drop decision is made. A header is dropped when its header pool reaches an occupancy threshold (H_{th}), or when the total available memory for multicast headers is full. If the header is accepted, a slot in the headers memory is occupied until the corresponding copy is received and ACKed by its (terminal or surrogate) target port.

In our implementation, the MCBC outputs the MC headers at a fixed sequence: if a flow heads to destinations *d1* to *d8*, the port always receives first the header for *d1* and last the one for *d8*. This fixed sequence triggers the following fairness issue. If the occupancy of the target header pool is close to its drop threshold, the "first" headers have a better chance of being accepted than the rest.

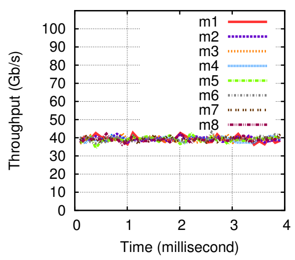
In the tests that follow, we set the header pool threshold at a low value (100 headers), to artificially induce multicast header drops. We configured a single multicast flow, at 99 Gb/s, targeting 1 local and 7 remote destinations. In this experiment, we have found that, for 99 Gb/s load, the remote copies are ACKed after approximately 3500 ns. There-



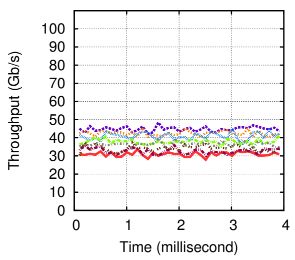
(a) Payload and header storage at ports



(b) Performance of baseline



(c) Randomized header arrivals



(d) RED

Figure 5: (a) Buffering of payload and multicast headers at fabric ports. (b-d) Per-subflow throughputs of a multicast flow heading to one local and seven remote destinations. In this scenario, we purposely reduce the header pool threshold so as to induce drops. 1522B frames.

fore, with 100 total headers available in the pool, there are 12.5 headers available per subflow. Each multicast header carries one 1522B frame, which amounts to 120 ns at 100 Gb/s. Therefore, the expected throughput of the subflows is $\frac{120\cdot 12.5}{3500} = 42.6$ Gb/s.

In reality, the subflow throughputs exhibit a wide spread, as shown in Fig. 5(b). One subflow is at 80 Gb/s, implying that it got ~8 in 10 of its headers accepted. At the other end of the spectrum, one subflow only had ~1 in 10 headers accepted and a throughput slightly above 10 Gb/s. The ordering of the subflows in the throughput ranking, from top to bottom, follows precisely the arrival order of subflow headers. This happens because, as the headers arrive at the port in bursts of eight, it is very likely that, in the interim between bursts, headers are released from the pool, making some space for the first headers of the next burst but not sufficient for all.

Figure 5(c) verifies that by randomizing the order of headers with every frame, the subflows receive exact fair throughputs, close to 40 Gb/s. Nevertheless, to simplify the implementation, we eventually opted for a *Random Early Detection (RED)* drop strategy. In particular, we use two thresholds, H_{hi} and H_{low}. Headers are accepted when the header pool occupancy is below H_{low}, and are dropped when it exceeds H_{hi}. For occupancies in between those thresholds, headers are dropped with probability p. Our tests have shown that optimal performance is obtained for $p = 0.5$: Too low a probability p, and the scheme is ineffective. On the other hand, too high a drop probability, and H_{low} becomes a hard drop threshold. As can be seen in Fig. 5(d), using the RED method with $H_{\text{hi}} = 100$, $H_{\text{low}} = 70$, and $p = 0.5$, we get reasonably good performance.

Optimized VOQ-drop policy: The drop functions discussed in the previous section treat all subflows equally. For lossy priority levels, we have augmented our VOQ architecture with a selective drop mechanism: Once its backlog exceeds a predefined threshold V_{th}, a VOQ starts dropping copies. Effectively, this equalizes the subflows' arrival rates with their service rates. Doing so, we prevent a congested subflow from monopolizing its header pool, and from holding off the release of shared payload buffers.

In the following experiments, we configured a multicast flow at 99 Gb/s, heading to 4 local and 4 remote destinations. Initially, the multicast flow is alone, receiving full throughput. Between 1 to 2 ms, its first local destination is targeted by 7 unicast flows, by 3 unicast flows between 2-3 ms, and by 1 unicast flow between 2-4 ms. In this experiment, all unicast and multicast flows are of the same lossy priority level, and are fairly served at fabric-egress ports. As also shown in Table I, the shared lossy payload buffers at the ingress terminal ports are $T_{\text{th}} = 740$.

In the first experiment, shown in Fig. 6(a), we deactivated VOQ synchronization. As can be seen, the per pool drop policies alone entail capping all subflows to the bandwidth of the most congested one. Even worse, because there is no VOQ synchronization, the fast-moving subflows fill up the cache. Effectively, when the congestive episode elapses at 4 ms, the throughput is still bounded at 60 Gb/s. This is corrected by VOQ synchronization in Fig. 6(b), but still the congested subflow dictates the rates of all.

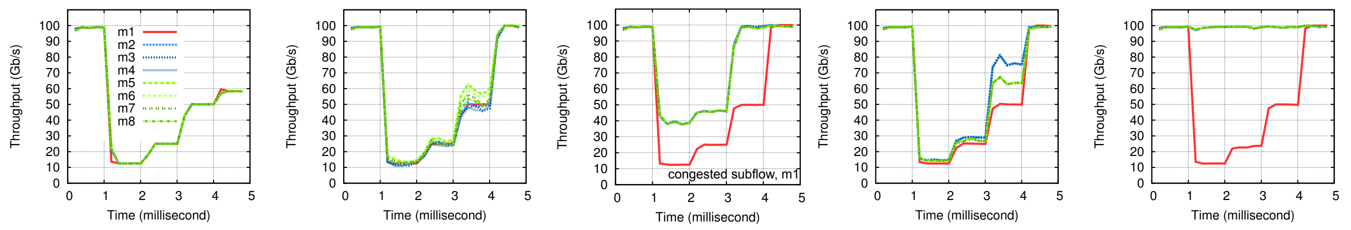
In Fig. 6(c), we activate VOQ tail-drop, keeping VOQ synchronization silent: VOQs with backlog $\geq V_{\text{th}} = 240$ buffer units (60 KB) don't accept new headers. As shown in the figure, between 2-3 ms, the non-congested subflows reach their fair share (99 Gb/s). Nevertheless, while the congestion is more severe in 1-3 ms, throughputs are far from optimal.

Furthermore, enabling VOQ synchronization in Fig. 6(d) actually does more bad than good, nullifying the benefits of VOQ drop. That was to be expected, since VOQ synchronization equalizes the service rates of the subflows. Hence, on one hand, the VOQ drop policy can nicely segregate the subflows. On the other hand, VOQ synchronization has to slow down the non-congested subflows, but prevents cache thrashing. Fortunately, we can reconcile the two methods, and get the best from each, by modifying the VOQ drop policy from tail-drop to drop-from-head.

Figure 6(e) verifies the excellent performance of drop-from-head. The throughputs of the fast subflows are *virtually unaffected* by the congested one. As explained in greater detail by the caption of Fig. 7, this happens because, after dropping the next-to-send frame of a slow VOQ, we can decrease the *pkts_ahead_cnt* of fast but blocked VOQs, therefore resuming their progress.

4.2 Congestion control for lossless traffic

For lossless service classes (priority levels), drops are not permitted anywhere in the fabric, including at fabric ports. To avoid drops, the ingress ports issue PAUSE messages when a pool exceeds an occupancy threshold, Q_{hi}. The PAUSE messages from fabric-ingress ports (i.e., at the source of the multicast tree) are per Ethernet priority level and are



(a) No VOQ drop, no VOQ sync (b) No VOQ drop, VOQ sync (c) VOQ tail-drop, no VOQ sync (d) VOQ tail-drop, VOQ sync (e) VOQ drop-from-head, VOQ sync

Figure 6: Throughputs of the multicast subflows heading to 4 local and 4 remote destinations. In time period 1 to 2 ms, a local destination is also targeted by 7 unicast flows, in 2 to 3 ms, it is targeted by 3 unicast flows, and by 1 unicast flow in 3 to 4 ms. 1522B frames.

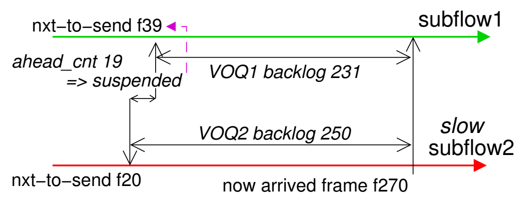


Figure 7: Illustration explaining why drop-from-head works better with VOQ synchronization than tail-drop. For clarity, consider single-packet frames and assume that the ACK arrives immediately after sending the frame copy from its source port. Furthermore, for the sake of simplicity, assume that $S_{\text{hi}} = S_{\text{low}} = 18$: Subflow 2 is congested, however due to VOQ synchronization, its distance from the fast subflow 1 is bounded. Currently, subflow 1 is suspended as its *pkts_ahead_cnt* exceeds 18. At the same time, the tail-drop function is about to reject the new frame f270 for subflow 2, because the corresponding VOQ 2 has a backlog of 250 buffer units. A first observation is that dropping frame f270 cannot unblock VOQ 1. In contrast, the drop-from-head policy will drop frame f20 instead of f270. The VOQ synchronization mechanism regards such dropped-from-head frames as having been forwarded. Effectively, after dropping f20 from VOQ 2, the *pkts_ahead_cnt* of VOQ 1 will be decremented by the number of packets in f20, and subflow 1 will be unblocked. From there on, the fast subflow sends frames at full speed, the slow one drops (from head) those that stay behind, and their "next-to-send" pointers run side-by-side.

received by the CNAs. In contrast, the PAUSE messages from surrogate ports are routed to the egress side of the surrogate, forestalling the loop back of new frames at a given VL. The egress VL pools at surrogate and destination ports, of either lossy or lossless classes, are flow controlled by the port-to-port, request-grant credit protocol.

With lossless traffic, saturation trees can be created. As a response, IEEE has recently standardized a congestion control scheme for Ethernet (Layer-2) networks, called *Quantized Congestion Notification (QCN, 802.1Qau)* [7]. Standard QCN performs congestion detection at switch output (fabric-egress) queues. Each congestion point samples the arriving frames, and when it detects congestion, it issues a *congestion notification message (CNM)* for the Ethernet flow of the most recently sampled frame. In response to the received CNM, the QCN rate limiter, implemented at the CNA, decreases the flow's injection rate.

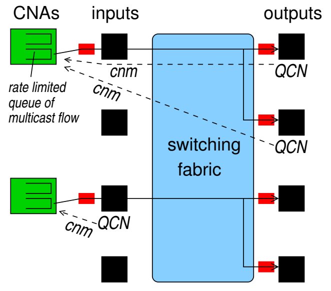
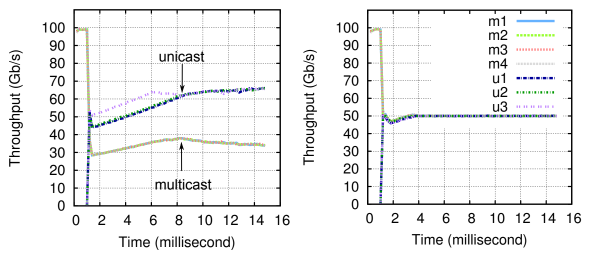


Figure 8: With QCN congestion points at outputs (industry-standard solution), a single multicast frame from the CNA may generate two CNMs, misleading QCN. In contrast, with QCN congestion points at inputs, each multicast frame yields at most one CNM.

However, as shown in Fig.8, using the standard QCN strategy, a *multicast frame* heading to two destinations can be sampled twice, and generate two ($|F|$ in general) CNMs. This multiplication of CNMs can breed unfair treatment of multicast flows. In Fig. 9(a), we have configured one such flow heading to four local destinations, and three unicast flows, from remote leaf nodes, each one targeting a distinct destination of the multicast flow. As can be seen, with QCN congestion points at the outputs (fabric-egress buffers), the unicast flows receive less CNMs and therefore get almost twice more bandwidth than the multicast flow.

In a previous publication [5], we found that, for unicast traffic, placing the congestion points at the inputs instead of outputs improves QCN's fairness and reduces the bandwidth overhead of CNMs. Furthermore, for multicast traffic, this alternative will generate at most one CNM per multicast frame, whatever the fanout of the flow may be. Indeed, as shown in Fig. 9(b), with QCN congestion points at fabric inputs (ingress buffers), the multicast flow performs equally with the unicast ones.

Our next experiment tests two multicast flows coming from the same fabric source port. Flow T1 heads to local destinations 1 and 2, and flow T2 heads to local destinations 3 and 4. Three (3) unicast flows, from remote leaf nodes, target the first destination, 1, of multicast flow T1. Figure 10 plots the per (sub)flow throughputs. Multicast flow T1 is



(a) QCN (standard) at outputs
(b) QCN at inputs

Figure 9: The throughputs of multicast subflows m1, m2, m3, m4, and unicast u1, u2, u3, shown separately for QCN congestion points at inputs and at outputs. Unicast flow u_i targets the same destination as subflow m_i.

expected to receive a bandwidth of 25 Gb/s, because that is the fair share of its most congested subflow. Without QCN (i.e., PAUSE-only in Fig.10(a)), T1 enters the fabric input at full speed, depleting it of available buffers. In response, the port exerts PAUSE to the CNA, which indiscriminately blocks both T1 and the non-congested flow T2.

QCN's promise is that, by throttling T1's departures from the CNA at 25 Gb/s, it will perform better than the PAUSE-only solution. However, QCN with congestion points at the inputs, shown in Fig.10(b), behaves no better than PAUSE alone. In this experiment, we used the QCN-standard flow marking scheme, which sends the CNMs to the most recently received frames. In our design, we have corrected this shortcoming with an *occupancy sampling* flow-marking scheme for QCN, which sends the CNMs to the flow with the largest occupancy in the monitored payload buffer pool [5]. Figures 10(c,d) demonstrate that our rectified flow-marking scheme, with congestion points at the inputs, performs in par with standard QCN at switch outputs. In addition, as previously verified in Fig. 9, it adapts favorably to multicast one-to-many flows.

5. MULTICAST REPLICATION TREES

Figure 11 depicts an arbitrary, multi-hop, replication tree. In this example, some copies are delivered directly from the source port (one in the source leaf and one in a remote one), other copies pass through one surrogate port, and some final copies pass through two surrogate ports.

Assignment of virtual lanes: As said before, each VL corresponds to one priority level, but two or more VLs may be used for the same priority. We use multiple VLs per priority level to separate the traffic and avoid deadlocks in replication trees. In particular, the per VL pools at the ingress side of a surrogate port can be seen as the (flow controlled for lossless VLs) extension of the corresponding (per VL) pools at the egress side of the surrogate. To avoid deadlocks, we increment the frame's VL when it is injected into the fabric from a terminal or surrogate port. Effectively, a frame with priority level P holding space in a pool of VLP_i can only wait for space in a higher VLP_j, $j > i$. By imposing this partial order on the resources, we prevent the corresponding circular dependencies and the ensuing deadlocks in replication trees [10].

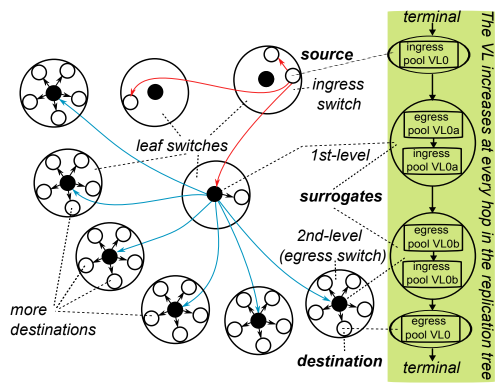


Figure 11: One possible replication tree. VL assignment for a frame at the source, 1st/2nd level surrogates, and destination. The frame belongs to priority 0 and uses the payload pool of VL0 at its source and destination ports.

Our replication trees have up to four surrogates, thus, a frame starting with VLP_0 at its source port can reach its destination port with VLP_5. However, because at the last hop in the frame's path (from a source or surrogate port to the final destination), we reuse the starting virtual lane of the frame, we need only 4 VLs per priority level.

An example is shown at the inlet of Fig.11. The payload of a frame with Ethernet priority $P = 0$ is stored in payload pool VL0 at its source port, VL0_a at the 1st-level surrogate, VL0_b at the 2nd-level surrogate, and VL0 again at its egress port. Not shown in the figure are the per-VL multicast header pools. At the source port, the header of the copy is stored in header pool VL0_a, and in header pools VL0_b and VL0 at the 1st- and 2nd-level surrogate, respectively.

Expansion speed: Each (terminal or surrogate) port can fanout to a maximum of $|f| = 8$ ports at full 100G speed. Therefore, in theory, N leaf nodes can be reached in logarithmic, $\log_{|f|} N$, time. In practice, a surrogate port will forward some copies to its local ports, offering replication at the egress switch without consuming extra leaf-spine bandwidth. This however narrows the effective fanout towards new leaf nodes. Assuming that the source and surrogate ports always replicate to all local destinations (4 at source and 5 at surrogates), a message that passes through n surrogates can reach $D(n)$ terminal destinations, where,

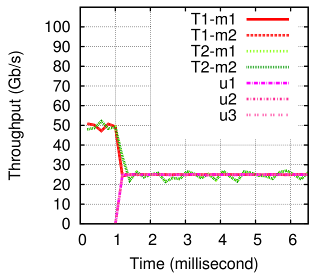
$$D(n) = 4 + 4 \cdot 5 \cdot \sum_{i=0}^{n-1} 3^i \qquad (1)$$

As said above, our hierarchical replication exploits up to $n = 4$ levels of surrogates, thus a multicast flow is limited to 804 terminal destinations. This is adequate for our 4-rack system, which provides a total of 640 user Ethernet ports at 100 Gb/s[1].

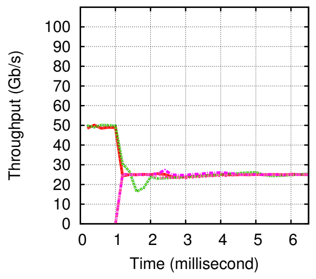
5.1 Bandwidth Overhead

In an ideal multicast replication scheme, each frame would travel to the spines only once, which would spawn one copy

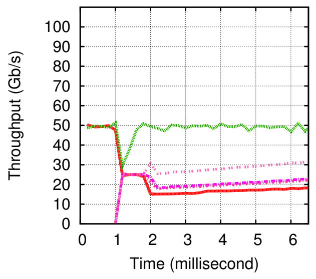
[1]Wider fanouts are possible in 2x40G port configurations. However, for clarity, we consider only 100G ports.



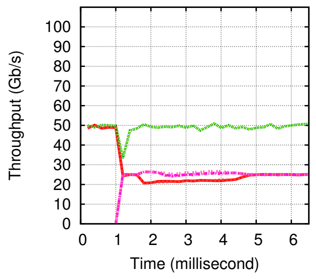
(a) Pause only (no QCN)



(b) QCN arrival sampling at inputs



(c) QCN arrival sampling at outputs



(d) QCN occupancy sampling at inputs

Figure 10: The throughputs of multicast subflows T1-m1, T1-m2, T2-m1, and T2-m2, as well as of unicast flows u1, u2, and u3. All unicast flows target the same terminal port with subflow T1-m1.

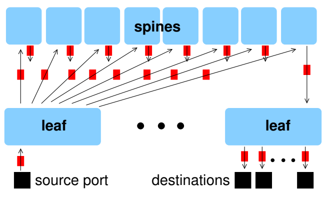


Figure 12: Bandwidth consumption of frame replication: (The diagram assumes single-path routing of frame's copies on 8 100G leaf-spine links, whereas our system actually sprays the load on 32 25G links.) The port sends the payload of the frame once to the ingress leaf switch, which outputs 8 copies to the spines. The latter send the copies to the target egress switches, which replicate them internally for all local destinations. Only 8 spines (32 in reality) and one egress switch are shown.

for every target egress (leaf) switch. Finally, the latter switches would fanout to all local destinations. Similarly, as shown in Fig.12, in our replication scheme, every egress switch receives each frame only once, replicating it internally. Therefore, we load the spine-to-leaf links as much as the ideal scheme. However, we consume extra bandwidth to send the frames to the spines as many times as we send them to the target egress switches. Therefore, the bandwidth consumption of our replication method, L, is, at most, two times the ideal L^* (we omit the full proof due to space limitations):

$$L \leq 2 \cdot L^* \qquad (2)$$

5.2 Concurrent provisioning of many servers

In our next experiment, we consider a typical application of multicast traffic. In particular, we source multicast traffic from 6 user ports in 3 leaf switches (2 sources per leaf). Every source hosts 10 multicast flows, each targeting all terminals in 20 egress leaf switches. Thus, every multicast flow has a fanout $|F| = 100$ $(20 \cdot 5)$. All multicast flows are configured on the same lossy priority level and use two levels of surrogates. The 20 target switches (5 with 1st-level surrogates and the rest 15 with 2nd-level surrogates) of each flow are randomly selected from a fixed set of 40 distinct switches. It follows that, with a load of λ Gb/s at source ports, the average demand at the destinations is $6 \cdot \frac{20}{40} \cdot \lambda$ Gb/s. Therefore, the maximum feasible load is 33.3 Gb/s.

From Eq.(1), for $n = 2$, a tree can deliver a message to at most 84 destinations. The 2-level trees in the current experiment offer a larger reach, because source ports do not deliver local copies. Effectively, with a fanout of 5 at the sources being dedicated for remote surrogates, our trees can extend to $5+5 \cdot 5 = 30$ egress leaf switches or 150 terminals.

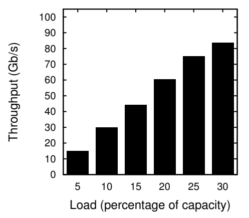
Figure 13(a) presents the average receive throughput at destinations against the input load. We see that the receive throughput at destinations grows linearly with the input load. However, for a source load of 30 Gb/s, the average receive throughput is 83 when it should be approximately 90 Gb/s. The last attested feasible source load is 25 Gb/s (i.e., 2.5 Gb/s per individual multicast flow), at which destinations are 76% utilized.

We conjecture that the cause for this throughput limitation is the statistical load imbalance at destinations. In simulations, we have seen that, even at 25 Gb/s input load, several destinations are fully loaded. With such multiple overloaded destinations, the ports are depleted of buffers, with an ensuing impact on throughput. We run the same experiment without VOQ synchronization, and the results are identical. In this configuration, every source port multiplexes 10 equal-loaded multicast flows, each having a local fanout of 5. Thus, on average, there are 50 VOQs active at every source, which will commonly overflow the sync CAM. Indeed, looking into the simulations, we verified that VOQs were never blocked for notably long periods.

Figure 13(b) presents the average frame delay as a function of the input load. As can be seen, at 5 Gb/s input load the delay is slightly above 4 microseconds, approaching 14 microseconds when the destinations are 82% utilized.

5.3 System-level broadcast

As shown in Fig. 14, a rack consists of two chassis, a chassis consists of four Chassis Interconnect Elements (CIEs), a CIE consists of four edge switches, and each edge switch has 5 user Ethernet ports at 100G. Although, replication trees may be created arbitrarily, we can exploit the topological affinity of ports to tame the configuration complexity. As shown in the figure, a broadcast flow can expand inside a rack using the Rack, Triplet, CIE, and egress-switch surrogates. The latter are not explicitly shown, but one of them is implied in every edge-switch with no outgoing arrows. Optimized configurations can replicate these surrogates in order to increase resiliency and performance.


(a) Receive throughput

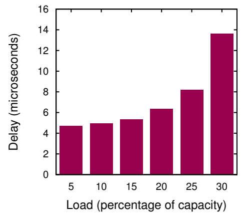

(b) Average frame delay

Figure 13: Performance under uniform multicast traffic. Sources are 6 100G ports. Each source hosts 10 equally-loaded multicast flows targeting 100 destinations in 20 randomly-selected egress switches.

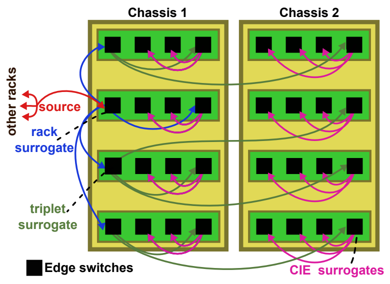


Figure 14: Hierarchical replication inside a rack (two chassis) using the Rack, Triplet, CIE, and egress-switch surrogates.

In our last experiment, we configured a flow broadcasting to all destinations in a full 4-rack system. Figure 15 presents the average receive throughputs at destinations against the input load of the broadcast flow, for 512B, 1522B, and 9022B (Jumbo) frames. As can be seen, in all cases the throughput grows linearly with the input load, with the exception of Jumbo frames that achieve a throughput up to 93 Gb/s.

Figure 16 depicts the average frame delay, measured from source to destination CNA. (Looking at the corresponding zero-load delays of *unicast* frames to remote (local) destinations, we measured 1.88 (0.143), 2.1 (0.35), and 3.6 (1.9) microseconds for 512B, 1522B, and 9022B frames, respectively.) We have separate plots for frames passing through 0, 1, 2, 3, and 4 levels of surrogates. As can be seen, the average delay increases as we move from 1 to 50 Gb/s load, but for 512B and 1522B frames, it stays almost constant when going from 50 to 99 Gb/s. The frame delay also increases with the number of surrogate levels. A significant fraction of these delays is spent scheduling and propagating the frames along the spine-leaf network at each hop of the surrogate tree. While not shown in this paper, our system can lower these delays by avoiding the request-grant admission phase when transmitting frames to non-congested destinations similar to [11, 12].

6. DISCUSSIONS

Implosion of control messages: The multiplication of (unicast) ACK and grant messages is an expected consequence of reliable scheduled multicasting. A similar effect, but which happens in the opposite direction, is due to unicast requests. In our system, we reduce the messages per port and the ensuing bandwidth overhead, through *(a)* hierarchical replication and *(b)* coalescing (combining) of the per-VOQ control messages.

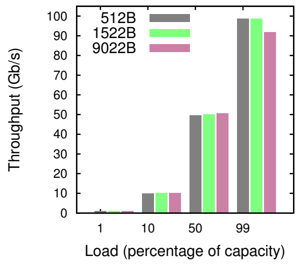


Figure 15: Average receive throughput at the destinations of a (broadcast) flow heading to 640 destinations against its input load. 100 Gb/s links.

Impact on unicast: One can mitigate the unwanted interferences at the ports by mapping unicast and multicast traffic to separate priority levels, each associated with a configurable service weight used for frame scheduling at fabric egress ports. In our architecture, doing so also segregates the unicast from the multicast VOQs.

Effect on neighbors: The frame replication via the multicast cache harnesses the available crossbar-input bandwidth of the neighboring Ethernet and PCIe ports. To alleviate the burden, we route the local copies, generated at a surrogate or terminal port, through special, one-to-one, crossbar links. This eliminates the impact of these local copies on the neighboring ports. Furthermore, we use (packet-size aware) weighted-round-robin scheduling [13]. One instance of the scheduler is located at every (ingress) crossbar input, and a configurable weight in the range $1-99$ defines the proportion of the link bandwidth that multicast copies can appropriate from unicast traffic originating from the corresponding terminal port.

Comparison to other switch architectures: Many Ethernet switch chips available today are based on a shared-memory architecture. These can forward frames copies by "simply" generating the corresponding headers and inserting them in the target output queues [14]. Our design recognizes that hierarchical replication will be indispensable for (reliable or not) hardware multicasting in the emerging high-speed datacenter fabrics. For example, even a moderate switch or fabric with 100 ports at 100G has to generate and process one 512B copy every 0.41 ns.

Variations of the request-grant/credit protocol used here exist in some chassis switch and router products [15, 16]. However, no performance evaluations nor detailed descriptions are available for these systems. In research papers, variations of these protocols have been described and evaluated in [17, 12], showing excellent QoS for unicast traffic. In our system, we use a *request-grant/credit-ACK* protocol to build a reliable server-fabric for performance-optimized datacenter clusters, and consider its multicasting performance. Bianco et. al. have studied multicast support for an asynchronous chassis switch using a request-grant credit protocol [18]. However, that system is remarkably smaller than ours (16 10G ports), and it is built around a single (spine) crossbar. Furthermore, [18] uses FIFO queuing of multicast

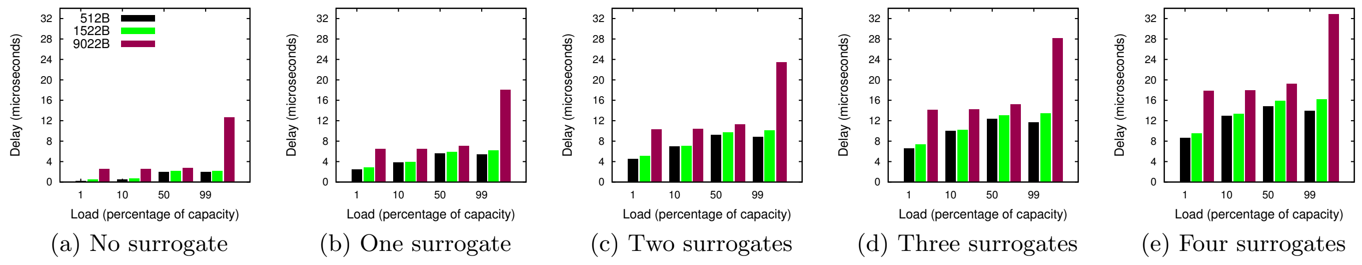


Figure 16: Average frame delay (CNA to CNA) in a broadcast flow heading to 640 destinations. We plot separately the delays for frames passing through 0, 1, 2, 3, and 4 levels of surrogates. 100 Gb/s links.

traffic, and cannot replicate delayed copies. Our system applies a favorable VOQ scheme, with accompanying congestion management for lossy and lossless priority levels, and can replicate asynchronously sent copies.

7. CONCLUSIONS

Our work builds upon a massive, performance-optimized server-rack fabric, with 640, equidistant, 100G Ethernet ports. We have described the uniform integration of multicast traffic, the obstacles we encountered and the solutions we implemented. We utilize hierarchical replication in hardware, facilitated by specially allocated fabric ports to cope with the enormous processing and forwarding rates. For the efficient replication of the delayed copies, we use a multicast cache in front of our input-queued switches, together with a VOQ synchronization mechanism. Finally, we develop advantageous flow and congestion control schemes for the multicast traffic running on lossy and lossless priority levels. Simulations on a detailed computer model evaluated the full 4-rack system and demonstrated its high performance abilities as well as many significant trade-offs.

8. ACKNOWLEDGMENTS

We would like to thank our colleagues in IBM Research–Zurich and in the hardware division of IBM (STG), and Anne-Marie Cromack for proofreading the manuscript. Special thanks to Cyriel Minkenberg, Mitch Gusat, David Shadivy, and Bill Holland, as well as to the anonymous reviewers that helped us improve the manuscript.

9. REFERENCES

[1] P. Radoslavov, C. Papadopoulos, R. Govindan, and D. Estrin, "A Comparison of Application-level and Router-assisted Hierarchical Schemes for Reliable Multicast," *IEEE/ACM Trans. Netw.*, vol. 12, no. 3, pp. 469–482, 2004.

[2] C.-K. Kim and T. T. Lee, "Call Scheduling Algorithms in a Multicast Switch," *IEEE Trans. Commun.*, vol. 40, no. 3, pp. 625–635, 1992.

[3] H. J. Chao, B.-S. Choe, J.-S. Park, and N. Uzun, "Design and Implementation of Abacus Switch: A Scalable Multicast ATM Switch," *IEEE JSAC*, vol. 15, no. 5, pp. 830–843, 1997.

[4] M. A. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri, "Multicast Traffic in Input-Queued Switches: Optimal Scheduling and Maximum Throughput," *IEEE/ACM Trans. Netw.*, vol. 11, no. 3, pp. 465–477, 2003.

[5] F. Neeser, N. Chrysos, R. Clauberg, D. Crisan, M. Gusat, C. Minkenberg, K. Valk, and C. Basso, "Occupancy Sampling for Terabit CEE Switches," in *Proc. IEEE Hot Interconnects*, 2012.

[6] N. Chrysos, F. Neeser, M. Gusat, C. Minkenberg, W. Denzel, and C. Basso, "All Routes to Efficient Datacenter Fabrics," in *Proc. INA-OCMC*, Berlin, Germany, January 2014.

[7] *802.1Qau - Virtual Bridged Local Area Networks - Amendment: Congestion Notification*, IEEE Std., 2010.

[8] B. Prabhakar, N. McKeown, and R. Ahuja, "Multicast Scheduling for Input-Queued Switches," *IEEE JSAC*, vol. 15, no. 5, pp. 855–866, 1997.

[9] D. Pan and Y. Yang, "FIFO-based Multicast Scheduling Algorithm For Virtual Output Queued Packet Switches," *IEEE Trans. Comp.*, vol. 54, no. 10, pp. 1283–1297, 2005.

[10] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks.* San Francisco, CA: Morgan Kaufmann Publishers Inc., 2003.

[11] C. Minkenberg and M. Gusat, "Design and Performance of Speculative Flow Control for High-Radix Datacenter Interconnect Switches," *Elsevier Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 680–695, Aug. 2009.

[12] N. Chrysos, "Congestion Management for Non-Blocking Clos Networks," in *Proc. ACM/IEEE ANCS*, Florida, USA, Dec. 2007.

[13] K. G. Harteros and M. Katevenis, "Fast parallel comparison circuits for scheduling," *Institute of Computer Science, FORTH*, 2002.

[14] F. M. Chiussi, Y. Xia, and V. P. Kumar, "Performance of Shared-Memory Switches under Multicast Bursty Traffic," *IEEE JSAC*, vol. 15, no. 3, pp. 473–487, 1997.

[15] P. Sindhu, P. Lacroute, M. Tucker, J. Weisbloom, and D. Winters, US Patent US 7,102,999 B2, Sep., 2006.

[16] O. Iny, US Patent US 7,619,970 B2, Nov., 2009.

[17] P. Pappu, J. Parwatikar, J. Turner, and K. Wong, "Distributed Queueing in Scalable High Performance Routers," in *Proc. IEEE INFOCOM*, San Francisco, USA, Apr. 2003.

[18] A. Bianco, P. Giaccone, E. M. Giraudo, F. Neri, and E. Schiattarella, "Multicast Support for a Storage Area Network Switch," in *Proc. IEEE GLOBECOM*, 2006.