Elmo: Source Routed Multicast for Public Clouds

Muhammad Shahbaz*
Stanford University

Lalith Suresh
VMware

Jennifer Rexford
Princeton University

Nick Feamster
Princeton University

Ori Rottenstreich
Technion

Mukesh Hira
VMware

ABSTRACT

We present Elmo, a system that addresses the multicast scalability problem in multi-tenant datacenters. Modern cloud applications frequently exhibit one-to-many communication patterns and, at the same time, require sub-millisecond latencies and high throughput. IP multicast can achieve these requirements but has control- and data-plane scalability limitations that make it challenging to offer it as *a service for hundreds of thousands of tenants*, typical of cloud environments. Tenants, therefore, must rely on unicast-based approaches (*e.g.*, application-layer or overlay-based) to support multicast in their applications, imposing bandwidth and end-host CPU overheads, with higher and unpredictable latencies.

Elmo scales network multicast by taking advantage of emerging programmable switches and the unique characteristics of datacenter networks; specifically, the hypervisor switches, symmetric topology, and short paths in a datacenter. Elmo encodes multicast group information inside packets themselves, reducing the need to store the same information in network switches. In a three-tier data-center topology with 27,000 hosts, Elmo supports a million multicast groups using an average packet-header size of 114 bytes, requiring as few as 1,100 multicast group-table entries on average in leaf switches, and having a traffic overhead as low as 5% over ideal multicast.

CCS CONCEPTS

• **Networks → Network protocol design**; **Programmable networks**; **In-network processing**; **Data center networks**;

KEYWORDS

Multicast; source routing; programmable parsing; bitmap encoding; P4; PISA; PISCES

ACM Reference Format:
Muhammad Shahbaz*, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. 2019. Elmo: Source Routed Multicast for Public Clouds. In *ACM SIGCOMM 2019 Conference (SIGCOMM '19), August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341302.3342066

*Work done while at Princeton University and VMware.

SIGCOMM '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-5956-6/19/09…$15.00
https://doi.org/10.1145/3341302.3342066

1 INTRODUCTION

To quote the Oracle Cloud team [4], we believe that the lack of multicast support among public cloud providers [21, 59, 90] is a huge "missed opportunity." First, IP multicast is in widespread use in enterprise datacenters to support virtualized workloads (*e.g.*, VXLAN and NVGRE) [46, 75, 113] and by financial services to support stock tickers and trading workloads [5, 15, 23]. These enterprises cannot easily transition from private datacenters to public clouds today without native multicast support. Second, modern datacenter applications are rife with *point-to-multipoint* communication patterns that would naturally benefit from native multicast. Examples include streaming telemetry [87, 89, 93, 96], replicated state machines [77, 78, 100], publish-subscribe systems [53, 58, 63, 110], database replication [7], messaging middleware [1, 9], data-analytics platforms [82], and parameter sharing in distributed machine learning [81, 86]. Third, all these workloads, when migrating to public clouds, are forced to use host-based packet replication techniques instead, such as application- or overlay-multicast [28, 35, 47, 67, 111]—indeed, many commercial offerings exist in this space [4, 20, 115]. This leads to inefficiencies for both tenants and providers alike (§5.2.1); performing packet replication on end-hosts instead of the network not only inflates CPU load in datacenters, but also prevents tenants from sustaining high throughputs and low latencies for multicast workloads.

A major obstacle to native multicast support in today's public clouds is the inherent data- and control-plane scalability limitations of multicast [75]. On the data-plane side, switching hardware supports only a limited number of multicast group-table entries, typically thousands to a few tens of thousands [38, 71, 83, 91]. These group-table sizes are a challenge even in small enterprise datacenters [17], let alone at the scale of public clouds that host *hundreds of thousands of tenants* [3]. On the control-plane side, IP multicast has historically suffered from having a "chatty" control plane [4, 10, 12], which is a cause for concern [4] in public clouds with a shared network where tenants introduce significant churn in the multicast state (*e.g.*, due to virtual machine allocation [24, 61] and migration [40, 45]). Protocols like IGMP and PIM trigger many control messages during churn and periodically query the entire broadcast domain, while the SDN-based solutions [83, 88] suffer from a high number of switch updates during churn.

In this paper, we present the design and implementation of Elmo, a system that overcomes the data- and control-plane scalability limitations that pose a barrier to multicast deployment in public clouds. Our key insight is that emerging programmable data planes [29, 33, 36] and the unique characteristics of datacenters (namely, hypervisor switches [37, 104], symmetric topology, and short paths [22, 60, 92]), enable the use of efficient *source-routed*

multicast, which significantly alleviates both the pressure on switching hardware resources and that of control-plane overheads during churn.

For data-plane scalability, hypervisor switches in Elmo simply encode the forwarding policy (*i.e.*, multicast tree) of a group in a packet header as opposed to maintaining group-table entries inside network switches—hypervisor switches do not have the same hard restrictions on table sizes like network switches have. By using source-routed multicast, Elmo accommodates groups for a large number of tenants running myriad workloads; if group sizes remain small enough to encode the entire multicast tree in the packet, there is practically no limit to the number of groups Elmo can support. Our encodings for multicast groups are compact enough to fit in a header that can be processed at line rate by programmable switches being deployed in today's datacenters [29, 33, 36].

For control-plane scalability, our source-routing scheme reconfigures groups by only changing the information in the header of each packet, an operation that only requires issuing an update to the source hypervisor(s). Since hypervisors support roughly 10–100x more forwarding-table updates per second than network switches [76, 97], Elmo absorbs most of the reconfiguration load at hypervisors rather than the physical network.

Source-routed multicast, however, is technically challenging to realize for three key reasons, which Elmo overcomes. First, the multicast tree encoding in the packet must be compact. Second, the protocol must use minimal state on network switches. Third, switches must be able to process the encoding in the packet at line rate.

Prior solutions fall short of meeting these scalability goals in different ways. They either cannot scale to a large number of groups without exhausting switch resources like group- and flow-table entries (IP multicast [43], Li et al. [83]). Or, they expect unorthodox switching capabilities that are infeasible to implement in today's datacenters, and yet, only work for smaller networks (BIER [117]) or with small group sizes (SGM [31]). Or, they have high traffic and end-host overheads, and therefore, cannot support multicast at line rate [28, 35, 47, 67, 111]. In comparison, Elmo achieves its data- and control-plane scalability goals without suffering from the above limitations, while also having several desirable features for public clouds: it provides address-space isolation, a necessity in virtualized environments [75]; it is multipath friendly; and its use of source-routing stays internal to the provider with tenants issuing standard IP multicast data packets. A detailed comparison appears in §6.

We present the following contributions in this paper. First, we develop a technique for compactly encoding multicast groups that are subtrees of multi-rooted Clos topologies (§3), the prevailing topology in today's datacenters [22, 60, 92]. These topologies create an opportunity to design a multicast group encoding that is compact enough to encode inside packets and for today's programmable switches to process at line rate. Second, we optimize the encoding so that it can be efficiently implemented in both hardware and software targets (§4). Our evaluation shows that our encoding facilitates a feasible implementation in today's multi-tenant datacenters (§5). In a datacenter with 27,000 hosts, Elmo scales to millions of multicast groups with minimal group-table entries and control-plane update overhead on switches. Elmo supports applications that use multicast without modification; we demonstrate two such applications: publish-subscribe (§5.2.1) and host telemetry (§5.2.2).

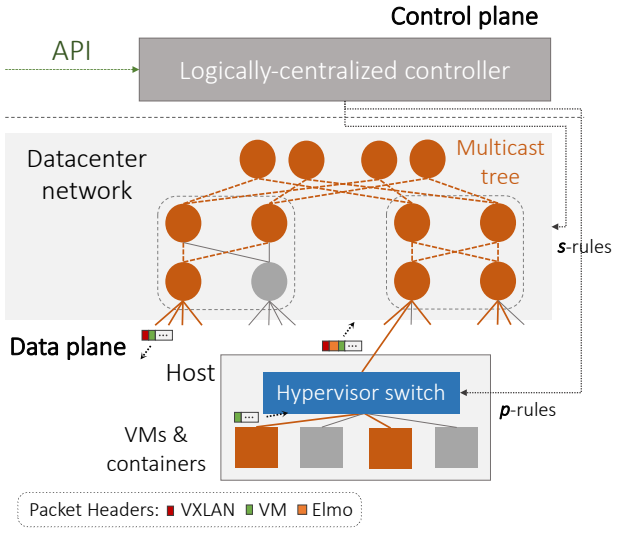


Figure 1: Elmo's architecture. A multicast tree (in orange) is encoded as *p*- and *s*-rules, which are installed in hypervisor and network switches respectively.

Lastly, we note that the failures of native multicast approaches have fatigued the networking community to date. We, however, believe that today's datacenters and, specifically, programmable software and hardware switches provide a fresh opportunity to revisit multicast; and Elmo is a first attempt at that. This work does not raise any ethical issues.

2 ELMO ARCHITECTURE

In Elmo, a *logically-centralized controller* manages multicast groups for tenants by installing flow rules in *hypervisor switches* (to encapsulate packets with a compact encoding of the forwarding policy) and the *network switches* (to handle forwarding decisions for groups too large to encode entirely in the packet header). Performing control-plane operations at the controller and having the hypervisor switches place forwarding rules in packet headers, significantly reduces the burden on network switches for handling a large number of multicast groups. Figure 1 summarizes our architecture.

Logically-centralized controller. The logically-centralized controller receives join and leave requests for multicast groups via an application programming interface (API). Cloud providers already expose such APIs [57] for tenants to request VMs, load balancers, firewalls, and other services. Each multicast group consists of a set of tenant VMs. The controller knows the physical location of each tenant VM, as well as the current network topology—including the capabilities and capacities of the switches, along with unique identifiers for addressing these switches. Today's datacenters already maintain such soft state about network configuration at the controller [92] (using fault-tolerant distributed directory systems [60]). The controller relies on a high-level language (like P4 [32, 34])

to configure the programmable switches at boot time so that the switches can parse and process Elmo's multicast packets. The controller computes the multicast trees for each group and uses a control interface (like P4Runtime [95]) to install match-action rules in the switches at run time. When notified of events (*e.g.*, network failures and group membership changes), the controller computes new rules and updates only the affected switches.[1] The controller uses a clustering algorithm for computing compact encodings of the multicast forwarding policies in packet headers (§3.2).

Hypervisor switch. A software switch [51, 99, 104], running inside the hypervisor, intercepts multicast data packets originating from VMs. The hypervisor switch matches the destination IP address of a multicast group in the flow table to determine what actions to perform on the packet. The actions determine: (i) where to forward the packet, (ii) type of encapsulation protocol to tunnel the packet (*e.g.*, VXLAN [85]), and (iii) what Elmo header to push on the packet. The Elmo header consists of a list of rules (packet rules, or *p*-rules for short)—each containing a set of output ports along with zero or more switch identifiers—that intermediate network switches use to forward the packet. These *p*-rules encode the multicast tree of a given group inside the packet, obviating the need for network switches to store a large number of multicast forwarding rules or require updates when the tree changes. Hypervisor switches run as software on physical hosts, they do not have the hard constraints on flow-table sizes and rule update frequency that network switches have [51, 76, 97, 99]. Each hypervisor switch only maintains flow rules for multicast groups that have member VMs running on the same host, discarding packets belonging to other groups.

Network switch. Upon receiving a multicast data packet, a physical switch (or network switch) running inside the network simply parses the header to look for a matching *p*-rule (*i.e.*, a *p*-rule containing the switch's own identifier) and forwards the packet to the associated output ports, as well as popping *p*-rules when they are no longer needed to save bandwidth. When a multicast tree is too large to encode entirely in the packet header, a network switch may have its own group-table rule (called a switch rule, or *s*-rule for short). As such, if a packet header contains no matching *p*-rule, the network switch checks for an *s*-rule matching the destination IP address (multicast group) and forwards the packet accordingly. If no matching *s*-rule exists, the network switch forwards the packet based on a default *p*-rule—the last *p*-rule in the packet header. Elmo encodes most rules inside the packet header (as *p*-rules), and installs only a small number of *s*-rules on network switches, consistent with the small group tables available in high-speed hardware switches [38, 71, 91]. The network switches in datacenters form a tiered topology (*e.g.*, Clos) with leaf and spine switches grouped into pods, and core switches. Together they enable Elmo to encode multicast trees efficiently.

[1]Today's datacenter controllers are capable of executing these steps in sub-second timescales [92] and can handle concurrent and consistent updates to tens of thousands of switches [92, 114].

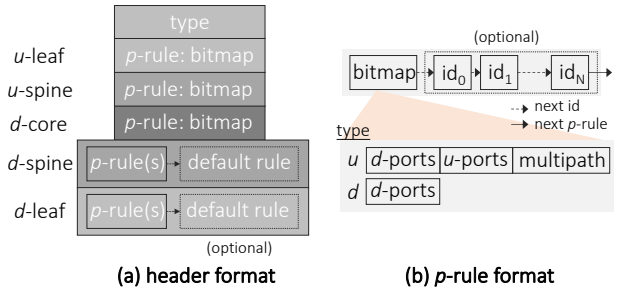


Figure 2: Elmo's header and *p*-rule format. A header consists of a sequence of upstream (*u*) leaf and spine *p*-rules, and the downstream (*d*) core, spine and leaf *p*-rules. The type field specifies the format of a *p*-rule bitmap, and the multipath flag in the upstream bitmap tells the switch to use multipathing when forwarding packets upstream.

3 ENCODING MULTICAST TREES

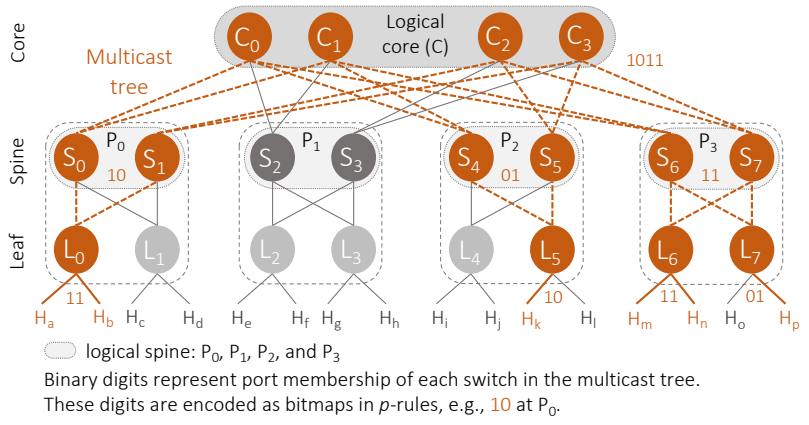
Upon receiving a multicast data packet, a switch must identify what set of output ports (if any) to forward the packet while ensuring it is sent on every output port in the tree and as few extra ports as possible. In this section, we first describe how to represent multicast trees efficiently, by capitalizing on the structure of datacenters (topology and short paths) and capabilities of programmable switches (flexible parsing and forwarding).

3.1 Compact and Simple Packet Header

Elmo encodes a multicast forwarding policy efficiently in a packet header as a list of *p*-rules (Figure 2a). Each *p*-rule consists of a set of output ports—encoded as *bitmap*—along with a list of switch identifiers (Figure 2b). A bitmap further consists of upstream and downstream ports along with a multipath flag, depending upon the type field. Network switches inspect the list of *p*-rules to decide how to forward the packet, popping *p*-rules when they are no longer needed to save bandwidth. We introduce five key design decisions (***D1–5***) that make our *p*-rule encoding both *compact* and *simple* for switches to process.

Throughout this section, we use a three-tier multi-rooted Clos topology (Figure 3a) with a multicast group stretching across three pods (marked in orange) as a running example. The topology consists of four core switches and pods, and two spine and leaf switches per pod. Each leaf switch further connects to eight hosts. Figure 3b shows two instances of an Elmo packet, originating from host H_a and H_k, for our example multicast tree (Figure 3a) after applying all the design optimizations, which we now discuss in detail.

D1: Encoding switch output ports in a bitmap. Each *p*-rule uses a simple bitmap to represent the set of switch output ports (typically, 48 ports) that should forward the packet (Figure 2b). Using a bitmap is desirable because it is the internal data structure that network switches use to direct a packet to multiple output ports [33]. Alternative encoding strategies use destination group members, encoded as bit strings [117]; bloom filters, representing link memberships [69, 101]; or simply a list of IP addresses [31] to



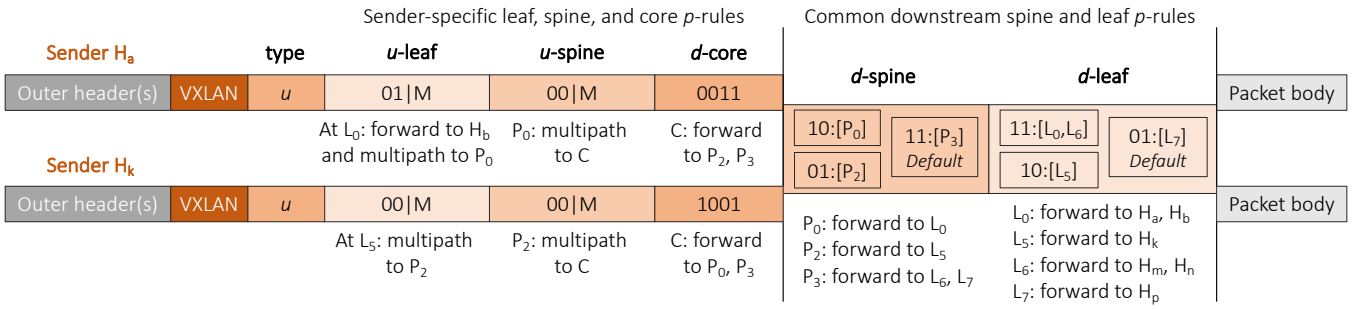
Downstream spine and leaf assignments with varying degree of redundancy (R) and *s*-rules.

R = 0		R = 2
#*s*-rules = 0	#*s*-rules = 1	#*s*-rules = 0, 1
(*p*) 10:[P_0] (*p*) 01:[P_2] (*d*) 11:[P_3]	(*p*) 10:[P_0] (*p*) 01:[P_2] (*s*) 11:[P_3]	(*p*) 10:[P_0] (*p*) 11:[P_2,P_3]
(*p*) 11:[L_0,L_6] (*p*) 10:[L_5] (*d*) 01:[L_7]	(*p*) 11:[L_0,L_6] (*p*) 10:[L_5] (*s*) 01:[L_7]	(*p*) 11:[L_0,L_6] (*p*) 11:[L_5,L_7]

#*p*-rules = 2 (and max two switches per *p*-rule)

legends: *p*-rule (*p*), *s*-rule (*s*), and default *p*-rule (*d*)

(a)



(b)

Figure 3: (a) An example multicast tree on a three-tier multi-rooted Clos topology with downstream spine and leaf *p*- and *s*-rules assignments for a group. (b) Elmo packets with $R = 0$ (*i.e.*, no redundant transmission per downstream *p*-rule) and *#s-rules* $= 0$ assignment. The packet originating from a sender is forwarded up to the logical core C using the sender-specific upstream *p*-rules, and down to the receivers using the common downstream *p*-rules (and *s*-rules). For example, L_0 forwards the packet from sender H_a to the neighboring receiver H_b while multipathing it to the logical spine P_0 (S_0, S_1) using the upstream *p*-rule: 01|M. In the downstream path, the packet arriving at P_2 is forwarded to L_5 using the *p*-rule: 01:[P_2]. Furthermore, each switch pops the corresponding *p*-rules and updates the type field before sending it to the next layer.

identify the set of output ports. However, these representations cannot be efficiently processed by network switches without violating line-rate guarantees (discussed in detail in §6).

Having a separate *p*-rule—with a bitmap and an identifier for each switch—for the multicast group in our example three-tier Clos topology (Figure 3a) needs a header size of 161 bits. For identifiers, we use two bits to identify the four core switches and three bits for spine and leaf switches, each.

D2: Encoding on the logical topology. Instead of having separate *p*-rules for each switch in the multicast tree, Elmo exploits the tiered architecture and short path lengths[2] in today's datacenter topologies to reduce the number of required *p*-rules to encode in the header. In multi-rooted Clos topologies, such as our example topology (Figure 3a), leaf-to-spine and spine-to-core links use multipathing. All spine switches in the same pod behave as one giant logical switch (forwarding packets to the same destination leaf switches), and all core switches together behave as one logical core switch (forwarding packets to the same pods). We refer to the *logical topology* as one where there is a single logical spine switch per pod (*e.g.*, P_0), and a single logical core switch (C) connected to pods.

We order *p*-rules inside a packet by layers according to the following topological ordering: *upstream leaf, upstream spine, core, downstream spine, and downstream leaf* (Figure 2a). Doing so also accounts for varying switch port densities per layer (*e.g.*, in Figure 3a, core switches connect to four spine switches and leaf switches connect to two hosts). Organizing *p*-rules by layers together with other characteristics of the logical topology allow us to further reduce header size and traffic overhead of a multicast group in four key ways:

a. Single p-rule per logical switch: We only require one *p*-rule per logical switch, with all switches belonging to the same logical group using not only the same bitmap to send packets to output ports, but also requiring only one logical switch identifier in the *p*-rule. For example, in Figure 3a, switches S_0 and S_1 in the logical pod P_0 use the same *p*-rule, 10:[P_0].

[2] *e.g.*, a maximum of five hops in the Facebook Fabric topology [22].

b. Forwarding using upstream p-rules: For switches in the upstream path, *p*-rules contain only the bitmap—including the downstream and upstream ports, and a multipath flag (Figure 2b: *type* = *u*)—without a switch identifier list. The multipath flag indicates whether a switch should use the configured underlying multipathing scheme (*e.g.*, ECMP, CONGA [19], or HULA [73]) or not. Otherwise, the upstream ports are used for forwarding packets upward to multiple switches in cases where no single spine or core has connectivity to all members of a multicast group (*e.g.*, due to network failures, §3.3). In Figure 3b, for the Elmo packet originating from host H_a, leaf L_0 will use the upstream *p*-rule (01|M) to forward the packet to the neighboring host H_b as well as multipathing it to the logical spine P_0 (S_0, S_1).

c. Forwarding using downstream p-rules: The only switches that require upstream ports represented in their bitmaps are the leaf and spine switches in the upstream path. The bitmaps of all other switches only require their downstream ports represented using bitmaps (Figure 2b: *type* = *d*). The shorter bitmaps for these switches, therefore, reduce space usage even further. Note, ports for upstream leaf and spine switches, including core, differ based on the source; whereas, downstream ports remain identical within the same multicast group. Therefore, Elmo generates a common set of downstream *p*-rules for leaf and spine switches that all senders of a multicast group share (Figure 3b).

d. Popping p-rules along the path: A multicast packet visits a layer only once, both in its upstream and downstream path. Grouping *p*-rules by layer, therefore, allows switches to pop all headers of that layer when forwarding a packet from one layer to another. This is because *p*-rules from any given layer are irrelevant to subsequent layers in the path. This also exploits the capability of programmable switches to decapsulate headers at line rate, discussed in §4. Doing so further reduces traffic overhead. For example, L_0 will remove the upstream *p*-rule (01|M) from the Elmo packet of sender H_a before forwarding it to P_0 (Figure 3b).

In our example (Figure 3a), encoding on the logical topology drops the header size to 83 bits (a reduction of 48% from *D1*).

D3: Sharing a bitmap across switches. Even with a logical topology, having a separate *p*-rule for each switch in the downstream path could lead to very large packet headers, imposing bandwidth overhead on the network. In addition, network switches have restrictions on the packet header sizes they can parse (*e.g.*, 512 bytes [33]), limiting the number of *p*-rules we can encode in each packet. To further reduce header sizes, Elmo assigns multiple switches within each layer (enabled by *D2*) to the same *p*-rule, if the switches have the same—or similar—bitmaps. Mapping multiple switches to a single bitmap, as a bitwise OR of their individual bitmap, reduces header sizes because the output bitmap of a rule requires more bits to represent than switch identifiers; for example, a datacenter with 27,000 hosts has approximately 1,000 switches (needing 10 bits to represent switch identifiers), whereas switch port densities range from 48 to 576 (requiring that many bits) [19]. The algorithm to identify sets of switches with similar bitmaps is described in §3.2.

We encode the set of switches as a simple *list* of switch identifiers, as shown in Figure 2b. Alternate encodings, such as bloom filters [30], are more complicated to implement—requiring a switch to account for false positives, where multiple *p*-rules are a "match." To keep false-positive rates manageable, these approaches lead to large filters [83], which is less efficient than having a list, as the number of switches with similar bitmaps is relatively small compared to the total number of switches in the datacenter network.

With *p*-rule sharing, such that the bitmaps of assigned switches differ by at most two bits (*i.e.*, $R = 2$, §3.2), logical switches P_2 and P_3 (in Figure 3a) share a downstream *p*-rule at the spine layer. At the leaf layer, L_0 shares a downstream *p*-rule with L_6 and L_5 with L_7. This further brings down the header size to 62 bits (a decrease of 25% from *D2*).

D4: Limiting header size using default p-rules. A default *p*-rule accommodates all switches that do not share a *p*-rule with other switches (*D3*). Default *p*-rules act as a mechanism to limit the total number of *p*-rules in the header. For example, in Figure 3a, with $R = 0$ and no *s*-rules, leaf switch L_7 gets assigned to a default *p*-rule. The default *p*-rules are analogous to the lowest priority rule in the context of a flow table. They are appended after all the other *p*-rules of a downstream layer in the header (Figure 2a).

The output bitmap for a default *p*-rule is computed as the bitwise OR of port memberships of all switches being mapped to the default rule. In the limiting case, the default *p*-rule causes a packet to be forwarded out of all output ports connected to the next layer at a switch (packets *only* make progress to the destination hosts); thereby, increasing traffic overhead because of the extra transmissions.

D5: Reducing traffic overhead using s-rules. Combining all the techniques discussed so far allows Elmo to represent any multicast tree without using *any* state in the network switches. This is made possible because of the default *p*-rules, which accommodate any switches not captured by other *p*-rules. However, the use of the default *p*-rule (and bitmap sharing across switches) results in extra packet transmissions that increase traffic overhead.

To reduce the traffic overhead without increasing header size, we exploit the fact that switches already support multicast group tables. Each entry, an *s*-rule, in the group table matches a multicast group identifier and sends a packet out on multiple ports. Before assigning a switch to a default *p*-rule for a multicast group, we first check if the switch has space for an *s*-rule. If so, we install an *s*-rule in that switch, and assign only those switches to the default *p*-rule that have no spare *s*-rule capacity. For example, in Figure 3a, with *s*-rule capacity of one entry per switch and $R = 0$, leaf switch L_7 now has an *s*-rule entry instead of the default *p*-rule, as in the previous case (*D4*).

3.2 Generating *p*- and *s*-Rules

Having discussed the mechanisms of our design, we now explain how Elmo expresses a group's multicast tree as a combination of *p*- and *s*-rules. The algorithm is executed once per downstream layer for each group. The input to the algorithm is a set of switch identifiers and their output ports for a multicast tree (*input bitmaps*).

Constraints. Every layer needs its own *p*-rules. Within each layer, we ensure that no more than H_{max} *p*-rules are used. We budget a separate H_{max} per layer such that the total number of *p*-rules is within a header-size limit. This is straightforward to

compute because: (i) we bound the number of switches per p-rule to K_{max}—restricting arbitrary number of switches from sharing a p-rule and inflating the header size—so the maximum size of each p-rule is known a priori, and (ii) the number of p-rules required in the upstream direction is known, leaving only the downstream spine and leaf switches. Of these, downstream leaf switches use most of the header capacity (§5).

A network switch has space for at most F_{max} s-rules (typically thousands to a few tens of thousands [38, 83, 91]), a shared resource across all multicast groups. For p-rule sharing, we identify groups of switches to share an output bitmap where the bitmap is the bitwise OR of all the input bitmaps. To reduce traffic overhead, we bound the total number of spurious transmissions resulting from a shared p-rule to R, where R is computed as the sum of Hamming Distances of each input bitmap to the output bitmap. Even though this does not bound the overall redundant transmissions of a group to R, our evaluation (§5) shows that it is still effective with negligible traffic overhead over ideal multicast.

Clustering algorithm. The problem of determining which switches share a p-rule maps to a well-known MIN-K-UNION problem, which is NP-hard but has approximate variants available [112]. Given the sets, $b_1, b_2, \ldots, b_n$, the goal is to find K sets such that the cardinality of their union is minimized. In our case, a set is a bitmap—indicating the presence or absence of a switch port in a multicast tree—and the goal is to find K such bitmaps whose bitwise OR yields the minimum number of set bits.

Algorithm 1 shows our solution. For each group, we assign p-rules until H_{max} p-rules are assigned or all switches have been assigned p-rules (Line 3). For p-rule sharing, we apply an approximate MIN-K-UNION algorithm to find a group of K input bitmaps (Line 4) [112]. We then compute the bitwise OR of these K bitmaps to generate the resulting output bitmap (Line 5). If the output bitmap satisfies the traffic overhead constraint (Line 6), we assign the K switches to a p-rule (Line 7) and remove them from the set of unassigned switches (Line 8), and continue at Line 3. Otherwise, we decrement K and try to find smaller groups (Line 10). When $K = 1$, any unassigned switches receive a p-rule each. At any point if we encounter the H_{max} constraint, we fallback to computing s-rules for any remaining switches (Line 13). If the switches do not have any s-rule capacity left, they are mapped to the default p-rule (Line 15).

Algorithm 1 Clustering algorithm for each layer of a group

Constants: R, H_{max}, K_{max}, F_{max}
Inputs: Set of all switches S, Bitmaps $B = b_i \forall i \in S$
Outputs: p-rules, s-rules, and default-p-rule

1: p-rules $\leftarrow \varnothing$, s-rules $\leftarrow \varnothing$, default-p-rule $\leftarrow \varnothing$
2: unassigned $\leftarrow B$, $K \leftarrow K_{max}$
3: **while** unassigned $\neq \varnothing$ and $|p\text{-rules}| < H_{max}$ **do**
4: bitmaps $\leftarrow$ approx-min-k-union(K, unassigned)
5: output-bm $\leftarrow$ Bitwise OR of all $b_i \in$ bitmaps
6: **if** dist(b_i, output-bm) $\leq R \; \forall \; b_i \in$ bitmaps **then**
7: p-rules $\leftarrow$ p-rules $\cup$ bitmaps
8: unassigned $\leftarrow$ unassigned $\setminus$ bitmaps
9: **else**
10: $K \leftarrow K - 1$
11: **for all** $b_i \in$ unassigned **do**
12: **if** switch i has $|s\text{-rules}| < F_{max}$ **then**
13: s-rules $\leftarrow$ s-rules $\cup \{b_i\}$
14: **else**
15: default-p-rule $\leftarrow$ default-p-rule $\mid b_i$
return p-rules, s-rules, default-p-rule

3.3 Ensuring Reachability via Upstream Ports under Network Failures

Network failures (due to faulty switches or links) require recomputing upstream p-rules for any affected groups. These rules are specific to each source and, therefore, can either be computed by the controller or, locally, at the hypervisor switches—which can scale and adapt more quickly to failures using host-based fault detection and localization techniques [72, 102].

When a failure happens, a packet may not reach some members of a group via any spine or core network switches using the underlying multipath scheme. In this scenario, the controller deactivates multipathing using the multipath flag (*D2*)—doing so does not require updating the network switches. The controller disables the flag in the bitmap of the upstream p-rules of the affected groups, and forwards packets using the upstream ports. Furthermore, to identify the set of possible paths that cover all members of a group, we reuse the same greedy set-cover technique as used by Portland [92] and therefore do not expand on it in this paper; for a multicast group G, upstream ports in the bitmap are set to forward packets to one or more spines (and cores) such that the union of reachable hosts from the spine (and core) network switches covers all the recipients of G. In the meantime, hypervisor switches gracefully degrade to unicast for the affected groups to mitigate transient loss. We evaluate how Elmo performs under failures in §5.1.3.

4 ELMO ON PROGRAMMABLE SWITCHES

We now describe how we implement Elmo to run at line rate on both network and hypervisor switches. Our implementation assumes that the datacenter is running P4 programmable switches like PISCES [104] and Barefoot Tofino [29].[3] These switches entail multiple challenges in efficiently parsing, matching, and acting on p-rules.

4.1 Implementing on Network Switches

In network switches, typically, a parser first extracts packet headers and then forwards them to the match-action pipeline for processing. This model works well for network protocols (like MAC learning and IP routing) that use a header field to lookup match-action rules in large flow tables. In Elmo, on the other hand, we find a matching p-rule from within the packet header itself. Using match-action tables to perform this matching is prohibitively expensive (Appendix A). Instead, we present an efficient implementation by exploiting the *match-and-set* capabilities of parsers in modern programmable data planes.

Matching p-rules using parsers. Our key insight here is that we match p-rules inside the switch's parser, and in doing so, no

[3] Example P4 programs for network and hypervisor switches based on the datacenter topology in Figure 3a are available on GitHub [6].

Scalability: Elmo scales to millions of multicast groups—supporting hundreds of dedicated groups per tenant—with minimal group-table usage and control-plane update overhead on network switches (§5.1)	In a multi-rooted Clos topology having 27,000 hosts and one million multicast groups, with group sizes based on a production trace: (i) 95-99% of groups can be encoded using an average *p*-rule header of 114 bytes (min 15, max 325) without using a default *p*-rule (Figure 4 and 5, *left*). (ii) Spine and leaf switches use only a mean (max) of 3,800 (11,000) and 1,100 (2,900) *s*-rules (Figure 4 and 5, *center*). (iii) *Despite the additional header*, traffic overhead is kept within 34% and 5% of the ideal for 64-byte and 1,500-byte packets, respectively (Figure 4 and 5, *right*). (iv) The average (max) update load on hypervisor, leaf, and spine switches is 21 (46), 5 (13), and 4 (7), respectively; core switches don't require any updates (Table 2).
Applications run unmodified, and benefit from reduced CPU and bandwidth utilization for multicast workloads (§5.2)	We run ZeroMQ (a publish-subscribe system) and sFlow (a monitoring application) on top of Elmo. Elmo enables these systems to scale to hundreds of receivers while maintaining constant CPU and bandwidth overhead at the transmitting VM (Figure 6).
End-host resource requirements: Elmo adds negligible overheads to hypervisor switches (§5.3)	A PISCES-based hypervisor switch encapsulates *p*-rules and forwards packets at line rate on a 20 Gbps link (Figure 7).

Table 1: Summary of results.

longer require a match-action stage to search *p*-rules at each switch—making switch memory resources available for other use, including *s*-rules. The switch can scan the packet as it arrives at the parser. The parser linearly traverses the packet header and stores the bits in a header vector based on the configured parse graph. Parsers in programmable switches provide support for setting metadata at each stage of the parse graph [32, 33, 94]. Hence, enabling basic *match-and-set* lookups inside the parsers.

Elmo exploits this property, extending the parser to check at each stage—when parsing *p*-rules—to see if the identifier of the given *p*-rule matches that of the switch. The parser parses the list of *p*-rules until it reaches a rule with "next *p*-rule" flag set to 0 (Figure 2b), or the default *p*-rule. If a matching *p*-rule is found, the parser stores the *p*-rule's bitmap in a metadata field and skips checking remaining *p*-rules, jumping to the next header (if any).

However, the size of a header vector (*i.e.*, the maximum header size a parser can parse) in programmable switch chips is also fixed. For RMT [33] the size is 512 bytes. We show in §5.1, how Elmo's encoding scheme easily fits enough *p*-rules using a header-size budget of 325 bytes (mean 114) while supporting millions of groups, with each tenant having around three hundred dedicated groups on average. The effective traffic overhead is low, as these *p*-rules get popped with every hop.

Forwarding based on p- and s-rules. After parsing the packet, the parser forwards metadata to the ingress pipeline, which includes a bitmap, a matched flag (indicating the presence of a valid bitmap), and a default bitmap. The ingress pipeline checks for the following cases in its control flow: if the matched flag is set, write the bitmap metadata to the queue manager [33], using a *bitmap_port_select* primitive[4]; else, lookup the group table using the destination IP address for an *s*-rule. If there is a match, write the *s*-rule's group identifier to the queue manager, which then converts it to a bitmap. Otherwise, use the bitmap from the default *p*-rule.

The queue manager generates the desired copies of the packet and forwards them to the egress pipeline. At the egress pipeline, we execute the following post-processing checks. For leaf switches, if a packet is going out toward the host, the egress pipeline invalidates all *p*-rules indicating the de-parser to remove these rules from the packet before forwarding it to the hosts. This offloads the burden at the receiving hypervisor switches, saving unnecessary CPU cycles spent to decapsulate *p*-rules. Otherwise, the egress pipeline invalidates all *p*-rules up to the *p*-rules(s) of the next-hop switch before forwarding the packet.

4.2 Implementing on Hypervisor Switches

In hardware switches, representing each *p*-rule as a separate header is required to match *p*-rules in the parsing stage. However, using the same approach on a hypervisor switch (like PISCES [104]) reduces throughput because each header copy triggers a separate DMA write call. Instead, to operate at line rate, we treat all *p*-rules as one header and encode it using a single write call (§5.3). Not doing so decreases throughput linearly with increasing number of *p*-rules.

5 EVALUATION

In this section, we evaluate the data- and control-plane scalability requirements of Elmo. Table 1 summarizes our results.

5.1 Scalability

5.1.1 Experiment setup. We now describe the setup we use to test the scale of the number of multicast groups Elmo can support and the associated traffic and control-plane update overhead on switches.

Topology. The scalability evaluation relies on a simulation over a large datacenter topology; the simulation places VMs belonging to different tenants on end hosts within the datacenter and assigns

[4]The queue manager is currently capable of receiving metadata pertaining to the multicast group identifier to use. We propose to, instead, use this mechanism to directly receive the output port bits as metadata. Supporting this simply re-uses existing switch ASICs and only incurs an additional area of 0.0005% (0.0015%) on a 64 (256) port switch. For reference, CONGA [19] and Banzai [106] consume 2% and 12% additional area, respectively.

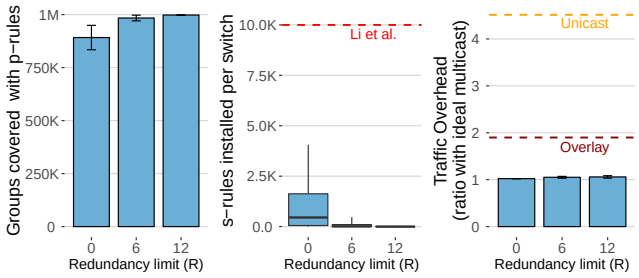


Figure 4: Placement strategy with no more than 12 VMs of a tenant per rack. *(Left)* Number of groups covered using non-default *p*-rules. *(Center)* *s*-rules usage across switches (the horizontal dashed line show rule usage for the scheme by Li et al. scheme [83] with no limit on VMs of a tenant per rack). *(Right)* Traffic overhead relative to ideal (horizontal dashed lines show unicast (top) and overlay multicast (bottom)).

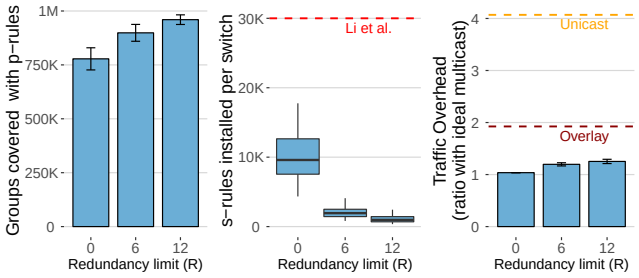


Figure 5: Placement strategy with no more than one VM of a tenant per rack. *(Left)* Number of groups covered using non-default *p*-rules. *(Center)* *s*-rules usage across switches (the horizontal dashed line show rule usage for the scheme by Li et al. [83] with no more than one VM of a tenant per rack). *(Right)* Traffic overhead relative to ideal (horizontal dashed lines show unicast (top) and overlay multicast (bottom)).

multicast groups of varying sizes to each tenant. We simulate using a Facebook Fabric topology—a three-tier topology—with 12 pods [22]. A pod contains 48 leaf switches each connected to 48 hosts. Thus, the topology with 12 pods supports 27,648 hosts, in total. (We saw qualitatively similar results while running experiments for a two-tier leaf-spine topology like that used in CONGA [19].)

Tenant VMs and placement. Mimicking the experiment setup from Li et al. [83]; the simulated datacenter has 3,000 tenants; the number of VMs per tenant follows an exponential distribution, with min=10, median=97, mean=178.77, and max=5,000; and each host accommodates at most 20 VMs. A tenant's VMs do not share the same physical host. Elmo is sensitive to the placement of VMs in the datacenter; which is typically managed by a placement controller [108], running alongside the network controller [14, 16]. We, therefore, perform a sensitivity analysis using a placement strategy where we first select a pod uniformly at random, then pick a random leaf within that pod and pack up to $\boldsymbol{P}$ VMs of that tenant under that leaf. P regulates the degree of co-location in the placement. We evaluate for $P = 1$ and $P = 12$ to simulate both dispersed and clustered placement strategies. If the chosen leaf (or pod) does not have any spare capacity to pack additional VMs, the algorithm selects another leaf (or pod) until all VMs of a tenant are placed.

Multicast groups. We assign multicast groups to each tenant such that there are a total of one million groups in the datacenter. The number of groups assigned to each tenant is proportional to the size of the tenant (*i.e.*, number of VMs in that group). We use two different distributions for groups' sizes, scaled by the tenant's size. Each group's member (*i.e.*, a VM) is randomly selected from the VMs of the tenant. The minimum group size is five. We use the group-size distributions described in Li's et al. paper [83]. We model the first distribution by analyzing the multicast patterns of an IBM WebSphere Virtual Enterprise (***WVE***) deployment, with 127 nodes and 1,364 groups. The average group size is 60, and nearly 80% of the groups have fewer than 61 members, and about 0.6% have more than 700 members. The second distribution generates tenant's groups' sizes that are uniformly distributed between the minimum group size and entire tenant size (***Uniform***).

5.1.2 Data-Plane Scalability.

Elmo scales to millions of multicast groups with minimal group-table usage. We first describe results for the various placement strategies under the IBM's WVE group size distribution. We cap the *p*-rule header size at 325 bytes (mean 114) per packet that allows up to 30 *p*-rules for the downstream leaf layer and two for the spine layer while consuming an average header space of 22.3% (min 3.0%, max 63.5%) for a chip that can parse a 512-byte packet header (*e.g.*, RMT [33])—leaving 400 bytes (mean) for other protocols, which in enterprises [55] and datacenters [54] consume about 90 bytes [56]. We vary the number of redundant transmissions (R) permitted due to *p*-rule sharing. We evaluate: (i) the number of groups covered using only the non-default *p*-rules, (ii) the number of *s*-rules installed, and (iii) the total traffic overhead incurred by introducing redundancy via *p*-rule sharing and default *p*-rules.

Figure 4 shows groups covered with non-default *p*-rules, *s*-rules installed per switch, and traffic overhead for a placement strategy that packs up to 12 VMs of a tenant per rack ($P = 12$). *p*-rules suffice to cover a high fraction of groups; 89% of groups are covered even when using $R = 0$, and 99.8% with $R = 12$. With VMs packed closer together, the allocated *p*-rule header sizes suffice to encode most multicast trees in the system. Figure 4 (*left*) also shows how increasing the permitted number of extra transmissions with *p*-rule sharing allows more groups to be represented using only *p*-rules.

Figure 4 (*center*) shows the trade-off between *p*-rule and *s*-rule usage. With $R = 0$, *p*-rule sharing tolerates no redundant traffic. In this case, *p*-rules comprise only of switches having precisely same bitmaps; as a result, the controller must allocate more *s*-rules, with 95% of leaf switches having fewer than 4,059 rules (mean 1,059). Still, these are on average 9.4 (max 2.5) times fewer rules when compared to the scheme by Li et al. [83] with no limit on the VMs of a tenant packed per rack ($P = All$). (Aside from these many group-table entries, Li's et al. scheme [83] also requires $O(\#Groups)$ flow-table entries for group aggregation.) Increasing R to 6 and 12 drastically decreases *s*-rule usage as more groups are handled using only *p*-rules. With $R = 12$, switches have on average 2.7 rules, with a maximum of 107.

Figure 4 (*right*) shows the resulting traffic overhead assuming 1,500-byte packets. With $R = 0$ and sufficient *s*-rule capacity, the resulting traffic overhead is identical to ideal multicast. Increasing R increases the overall traffic overhead to 5% of the ideal. Overhead is modest because even though a data packet may have as much as 325 bytes of *p*-rules at the source, *p*-rules are removed from the header with every hop (§3.1), reducing the total traffic overhead. For 64-byte packets, the traffic overhead for WVE increases only to 29% and 34% of the ideal when $R = 0$ and $R = 12$, still significantly improving over overlay multicast[5] (92%) and unicast (406%).

Fundamentally, these results highlight the degree to which source routing takes state away from the switches, thereby improving over (1) Li et al. [83], which relies on large amounts of state in the switches, and (2) unicast-based schemes, which inflate traffic overhead by sending duplicate packets.

p-rule sharing is effective even when groups are dispersed across leaves. Thus far, we discussed results for when up to 12 VMs of the same tenant were placed in the same rack. To understand how our results vary for different VM placement strategies, we explore an extreme case where the placement strategy spreads VMs across leaves, placing no more than a single VM of a tenant per rack. Figure 5 (*left*) shows this effect. Dispersing groups across leaves requires larger headers to encode the whole multicast tree using only *p*-rules. Even in this case, *p*-rules with $R = 0$ can handle as many as 750K groups, since 77.8% of groups have less than 36 switches, and there are 30 *p*-rules for the leaf layer—just enough header capacity to be covered only with *p*-rules. The average (max) *s*-rule usage is still 3.3 (1.7) times less than Li's et al. SDN-based multicast approach [83] under this placement strategy. Increasing R to 12 ensures that 95.9% of groups are covered using *p*-rules. We see the expected drop in *s*-rule usage as well, in Figure 5 (*center*), with 95% of switches having fewer than 2,435 *s*-rules. The traffic overhead increases to within 25% of the ideal when $R = 12$, in Figure 5 (*right*), but still improving significantly over overlay multicast (92%) and unicast (406%).

p-rule sharing is robust to different group size distributions. We also study how the results are affected by different distributions of group sizes, using the Uniform group size distribution. We expect that larger group sizes will be more difficult to encode using only *p*-rules. We found that with the $P = 12$ placement strategy, the total number of groups covered using only *p*-rules drops to 814K at $R = 0$ and to 922K at $R = 12$. When spreading VMs across racks with $P = 1$, only 250K groups are covered by *p*-rules using $R = 0$, and 750K when $R = 12$. The total traffic overhead for 1,500-byte packets in that scenario increases to 11%.

Reducing s-rule capacity increases default p-rule usage if p-rule sizes are insufficient. Limiting the *s*-rule capacity of switches allows us to study the effects of limited switch memory on the efficiency of the encoding scheme. Doing so increases the number of switches that are mapped to the default *p*-rule. When limiting the *s*-rules per switch to 10,000 rules, and using the extreme $P = 1$ placement strategy, the uniform group size distribution experiences higher traffic overheads, approaching that of overlay multicast at $R = 0$ (87% vs 92%), but still being only 40% over ideal multicast at $R = 12$. Using the WVE distribution, however, brings down traffic overhead to 19% and 25% for $R = 6$ and $R = 12$, respectively. With the tighter placement of $P = 12$, however, we found the traffic overhead to consistently stay under 5% regardless of the group-size distribution.

[5]In overlay multicast, the source host's hypervisor switch replicates packets to one host under each participating leaf switch, which then replicates packets to other hosts under that leaf switch.

Switch	Elmo	Li et al. [83]
hypervisor	21 (46)	NE (NE)
leaf	5 (13)	42 (42)
spine	4 (7)	78 (81)
core	0 (0)	133 (203)

Table 2: The average (max) number of hypervisor, leaf, spine, and core switch updates per second with $P = 1$ placement strategy. Results are shown for WVE distribution. (NE: not evaluated by Li et al. [83])

Reduced p-rule header sizes and s-rule capacities inflate traffic overheads. Finally, to study the effects of the size of the *p*-rule header, we reduced the size so that the header could support at most 10 *p*-rules for the leaf layer (*i.e.*, 125 bytes per header). In conjunction, we also reduced the *s*-rule capacity of each switch to 10,000 and used the $P = 1$ placement strategy to test a scenario with maximum dispersement of VMs. This challenging scenario even brought the traffic overhead to exceed that of overlay multicast at $R = 12$ (123%). However, in contrast to overlay multicast, Elmo still forwards packets at line rate without any overhead on the end-host CPU.

Elmo can encode multicast policies for non-Clos topologies (*e.g.*, Xpander [109] or Jellyfish [105]); however, the resulting tradeoffs in the header-space utilization and traffic overhead would depend on the specific characteristics of these topologies. In a symmetric topology like Xpander with 48-port switches and degree $d = 24$, Elmo can still support a million multicast groups with a max header-size budget of 325 bytes for a network with 27,000 hosts. On the other hand, asymmetry in random topologies like Jellyfish can make it difficult for Elmo to find opportunities for sharing a bitmap across switches, leading to poor utilization of the header space.

5.1.3 Control-Plane Scalability.

Elmo is robust to membership churn and network failures. We use the same Facebook Fabric setup to evaluate the effects of group membership churn and network failures on the control-plane update overhead on switches.

a. Group membership dynamics. In Elmo, we distinguish between three types of members: senders, receivers, or both. For this evaluation, we randomly assign one of these three types to each member. All VMs of a tenant who are not a member of a group have equal probability to join; similarly, all existing members of the group have an equal probability of leaving. Join and leave events are generated randomly, and the number of events per group is proportional to the group size, which follows the WVE distribution.

If a member is a sender, the controller only updates the source hypervisor switch. By design, Elmo only uses *s*-rules if the *p*-rule header capacity is insufficient to encode the entire multicast tree of a group. Membership changes trigger updates to sender and receiver hypervisor switches of the group depending on whether upstream or downstream *p*-rules need to be updated. When a change affects *s*-rules, it triggers updates to the leaf and spine switches.

For one million join/leave events with one million multicast groups and $P = 1$ placement strategy, the update load on these switches remains well within the studied thresholds [64]. As shown in Table 2, with membership changes of 1,000 events per second, the average (max) update load on hypervisor, leaf, and spine switches is 21 (46), 5 (13), and 4 (7) updates per second, respectively; core switches don't require any updates. Whereas, with Li's et al. approach [83], the average update load exceeds 40 updates per second on leaf, spine, and core switches—reaching up to 133 updates per second for core switches (they don't evaluate numbers for hypervisor switches). Hypervisor and network switches can support up to 40,000 and 1,000 updates per second [76, 97] respectively, implying that Elmo leaves enough spare control-traffic capacity for other protocols in datacenters to function.

b. Network failures. Elmo gracefully handles spine and core switch failures forwarding multicast packets via alternate paths using upstream ports represented in the groups' *p*-rule bitmap (when a leaf switch fails, all hosts connected to it lose connectivity to the network until the switch is online again). During this period, hypervisor switches momentarily shift to unicast (within 3 ms [18]) for some groups to minimize transient loss while the network is reconfiguring (§3.3). In our simulations, up to 12.3% of groups are impacted when a single spine switch fails and up to 25.8% when a core switch fails. Hypervisor switches incur average (max) updates of 176.9 (1712) and 674.9 (1852) per failure event, respectively. We measure that today's hypervisor switches are capable of handling *batched* updates of 80,000 requests per second (on a modest server) and, hence, reconfigure within 25 ms of these failures.

Elmo's controller computes p- and s-rules for a group within a millisecond. Our controller consistently executes Algorithm 1 for computing *p*- and *s*-rules in less than a millisecond. Across our simulations, our Python implementation computes the required rules for each group in 0.20 ms ± 0.45 ms (SD), on average, for all group sizes with a header size limit of 325 bytes. Existing studies report up to 100 ms for a controller to learn an event, issue updates to the network, and for the network state to converge [92]. Elmo's control logic, therefore, contributes little to the overall convergence time for updates and is fast enough to support the needs of large datacenters today, even before extensive optimization.

5.2 Evaluating End-to-end Applications

We ran two popular datacenter applications on top of Elmo: ZeroMQ [63] and sFlow [96]. We ran both applications *unmodified* on top of Elmo and benefited from reduced CPU and bandwidth utilization for multicast workloads.

Testbed setup. The topology for this experiment comprises nine PowerEdge R620 servers having two eight cores Intel(R) Xeon(R) CPUs running at 2.00 GHz and with 32 GB of memory, and three

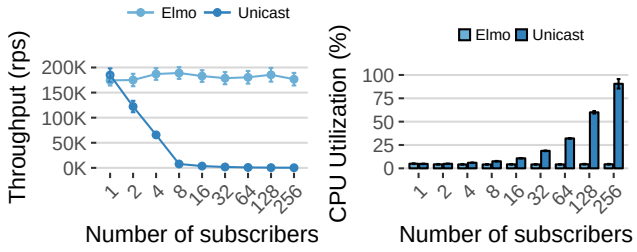


Figure 6: Requests-per-second and CPU utilization of a pub-sub application using ZeroMQ for both unicast and Elmo with a message size of 100 bytes.

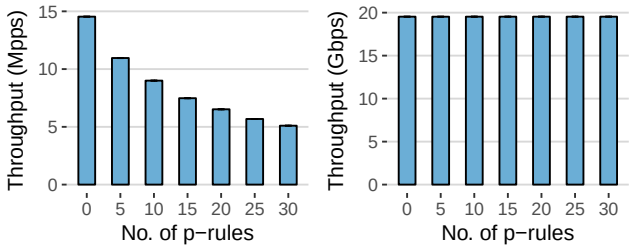


Figure 7: PISCES throughput in millions of packets per second (*left*) and Gbps (*right*) when adding different number of *p*-rules, expressed as a single P4 header.

dual-port Intel 82599ES 10 Gigabit Ethernet NICs. Three of these machines emulate a spine and two leaf switches; these machines run an extended version of the PISCES [104] switch with support for the *bitmap_port_select* primitive for routing traffic between interfaces. The remaining machines act as hosts running a vanilla PISCES hypervisor switch, with three hosts per leaf switch.

5.2.1 Publish-subscribe using ZeroMQ. We implement a publish-subscribe (pub-sub) system using ZeroMQ (over UDP). ZeroMQ enables tenants to build pub-sub systems on top of a cloud environment (like AWS [2], GCP [8], or Azure [11]), by establishing unicast connections between publishers and subscribers.

Throughput (rps). Figure 6 (*left*) shows the throughput comparison in requests per second. With unicast, the throughput at subscribers decreases with an increasing number of subscribers because the publisher becomes the bottleneck; the publisher services a single subscriber at 185K rps on average and drops to about 0.3K rps for 256 subscribers. With Elmo, the throughput remains the same regardless of the number of subscribers and averages 185K rps throughout.

CPU utilization. The CPU usage of the publisher VM (and the underlying host) also increases with increasing number of subscribers, Figure 6 (*right*). The publisher VM consumes 32% of the VM's CPU with 64 subscribers and saturates the CPU with 256 subscribers onwards. With Elmo, the CPU usage remains constant regardless of the number of subscribers (*i.e.*, 4.9%).

5.2.2 Host telemetry using sFlow. As our second application, we compare the performance of host telemetry using sFlow

with both unicast and Elmo. sFlow exports physical and virtual server performance metrics from sFlow agents to collector nodes (*e.g.*, CPU, memory, and network stats for docker, KVMs, and hosts) set up by different tenants (and teams) to collect metrics for their needs. We compare the egress bandwidth utilization at the host of the sFlow agent with increasing number of collectors, using both unicast and Elmo. The bandwidth utilization increases linearly with unicast, with the addition of each new collector. With 64 collectors, the egress bandwidth utilization at the agent's host is 370.4 Kbps. With Elmo, the utilization remains constant at about 5.8 Kbps (equal to the bandwidth requirements for a single collector).

5.3 End-host Microbenchmarks

We conduct microbenchmarks to measure the incurred overheads on the hypervisor switches when encapsulating p-rule headers onto packets (decapsulation at every layer is performed by network switches). We found Elmo imposes negligible overheads at hypervisor switches.

Setup. Our testbed has a host H_1 directly connected to two hosts H_2 and H_3. H_1 has 20 Gbps connectivity with both H_2 and H_3, via two 10 Gbps interfaces per host. H_2 is a traffic source and H_3 is a traffic sink; H_1 is running PISCES with the extensions for Elmo to perform necessary forwarding. H_2 and H_3 use MoonGen [50] for generating and receiving traffic, respectively.

Results. Figure 7 shows throughput at a hypervisor switch when encapsulating different number of p-rule headers, in both packets per second (pps) and Gigabits per second (Gbps). Increasing the number of p-rules reduces the pps rate, as the packet size increases, while the throughput in bps remains unchanged. The throughput matches the capacity of the links at 20 Gbps, demonstrating that Elmo imposes negligible overhead on hypervisor switches.

6 RELATED WORK

Table 3 highlights various areas where related multicast approaches fall short compared to Elmo in the context of today's cloud environments.

Wide-area multicast. Multicast has been studied in detail in the context of wide-area networks [27, 42, 43, 48, 103], where the lack of applications and deployment complexities led to limited adoption [49]. Furthermore, the decentralized protocols, such as IGMP and PIM, faced several control-plane challenges with regards to stability in the face of membership churn [49]. Over the years, much work has gone into IP multicast to address issues related to scalability [39, 69], reliability [25, 26, 52, 79], security [70], and congestion control [62, 116]. Elmo, however, is designed for datacenters which differ in significant ways from the wide-area context.

Data-center multicast. In datacenters, a single administrative domain has control over the entire topology and is no longer required to run the decentralized protocols like IGMP and PIM. However, SDN-based multicast is still bottlenecked by limited switch group-table capacities [38, 71, 91]. Approaches to scaling multicast groups in this context have tried using rule aggregation to share multicast entries in switches with multiple groups [44, 66, 83, 84]. Yet, these solutions operate poorly in cloud environments because

Feature \ Scheme	IP Multicast	Li et al. [83]	Rule aggr. [83]	App. Layer	BIER [117]	SGM [31]	Elmo
#Groups	5K	150K	500K	1M+	1M+	1M+	1M+
Group-table usage	high	high	mod	none	low	none	low
Flow-table usage*	none	mod	high	none	none	none	none
Group-size limits	none	none	none	none	2.6K	<100	none
Network-size limits: #hosts	none	none	none	none	2.6K	none	none
Unorthodox switch capabilities	no	no	no	no	yes	yes	no
Line-rate processing	yes	yes	yes	no	yes	no	yes
Address-space isolation	no	no	no	yes	yes	yes	yes
Multipath forwarding	no	lim	lim	yes	yes	yes	yes
Control overhead	high	low	mod	none	low	low	low
Traffic overhead	none	none	low	high	low	none	low
End-host replication	no	no	no	yes	no	no	no

Table 3: Comparison between Elmo and related multicast approaches evaluated against a group-table size of 5,000 rules and a header-size budget of 325 bytes. (∗ uses unicast flow-table entries for multicast.)

a change in one group can cascade to other groups, they do not provide address-space isolation (*i.e.*, tenants cannot choose group addresses independently from each other), and they cannot utilize the full bisection bandwidth of the network [83, 92]. Elmo, on the other hand, operates on a group-by-group basis, maintains address-space isolation, and makes full use of the entire bisection bandwidth.

Application/overlay multicast. The lack of IP multicast support, including among the major cloud providers [21, 59, 90], requires tenants to use inefficient software-based multicast solutions such as overlay multicast or application-layer mechanisms [28, 41, 47, 63, 110]. These mechanisms are built on top of unicast, which as we demonstrated in §5, incurs a significant reduction in application throughput and inflates CPU utilization. SmartNICs (like Netronome's Agilio [13]) can help offload packet-replication burden from end-hosts' CPUs. However, these NICs are limited in their capabilities (such as flow-table sizes and the number of packets they can clone). The replicated packets contend for the same egress port,

further restricting these NICs from sustaining line rate and predictable latencies. With native multicast, as in Elmo, end hosts send a single copy of the packet to the network and use intermediate switches to replicate and forward copies to multiple destinations at line rate.

Source-routed multicast. Elmo is not the first system to encode forwarding state inside packets. Previous work [69, 80, 101] have tried to encode link identifiers inside packets using bloom filters. BIER [117] encodes group members as bit strings, whereas SGM [31] encodes them as a list of IP addresses. Switches then look up these encodings to identify output ports. However, all these approaches require unorthodox processing at switches (*e.g.*, loops [117], multiple lookups on a single table [31], division operators [68], run-length encoding [65], and more), and are infeasible to implement and process multicast traffic at line rate. BIER, for example, requires flow tables to return all entries (wildcard) matching the bit strings—a prohibitively expensive data structure compared to traditional TCAM-based match-action tables in emerging programmable data planes. SGM looks up all the IP addresses in the routing table to find their respective next hops, requiring an arbitrary number of routing table lookups, thus, breaking the line-rate invariant. Moreover, these approaches either support small group sizes or run on small networks only. Contrary to these approaches, Elmo is designed to operate at line rate using modern programmable data planes (like Barefoot Tofino [29] and Cavium XPliant [36]) and supports hundreds of groups per tenant in a cloud datacenter.

7 CONCLUSION

We presented Elmo, a system to scale native multicast to support the needs of modern multi-tenant datacenters. By compactly encoding multicast forwarding rules inside packets themselves, and having programmable switches process these rules at line, Elmo reduces the need to install corresponding group-table entries in network switches. In simulations based on a production trace, we show that even when all tenants collectively create millions of multicast groups in a datacenter with 27,000 hosts, Elmo processes 95–99% of groups without any group table entries in the network, while keeping traffic overheads between 5–34% of the ideal for 64-byte and 1,500-byte packets. Furthermore, Elmo is inexpensive to implement in programmable switches today and supports applications that use multicast without modification. Our design also opens up opportunities around deployment, reliability, security, and monitoring of native multicast in public clouds, as well as techniques that are of broader applicability than multicast.

Path to deployment. Elmo runs over a tunneling protocol (*i.e.*, VXLAN) that allow cloud providers to deploy Elmo incrementally in their datacenters by configuring existing switches to refer to their group tables when encountering an Elmo packet (we tested this use case with a legacy switch). While Elmo retains its scalability benefits within clusters that have migrated to Elmo-capable switches, the group-table sizes on legacy switches will continue to be a scalability bottleneck. For multi-datacenter multicast groups, the source hypervisor switch in Elmo can send a unicast packet to a hypervisor in the target datacenter, which will then multicast it using the group's p- and s-rules for that datacenter.

Reliability and security. Elmo supports the same best-effort delivery semantics of native multicast. For reliability, multicast protocols like PGM [107] and SRM [52] may be layered on top of Elmo to support applications that require reliable delivery. Furthermore, as Elmo runs inside multi-tenant datacenters, where each packet is first received by a hypervisor switch, cloud providers can enforce multicast security policies on these switches [98], dropping malicious packets (*e.g.*, DDoS [70]) before they even reach the network.

Monitoring. Debugging multicast traffic has been an issue, with difficulties troubleshooting copies of a multicast packet and the lack of tools (like traceroute and ping). However, recent advances in network telemetry (*e.g.*, INT [74]) can simplify monitoring and debugging of multicast systems like Elmo, by collecting telemetry data within a multicast packet itself, which analytics servers can then use for debugging (*e.g.*, finding anomalies in routing configurations).

Applications beyond multicast. We believe our approach of using the programmable parser to offset limitations of the match-action pipeline is of general interest to the community, beyond source-routing use cases. Furthermore, source tagging of packets could be helpful to drive other aspects of packet handling (*e.g.*, packet scheduling and monitoring)—beyond routing—inside the network.

ACKNOWLEDGMENTS

We thank our shepherd Justine Sherry, Sujata Banerjee, Amin Vahdat, Jon Crowcroft, Radhika Mysore, Manya Ghobadi, Isaac Keslassy, Udi Wieder, Mihai Budiu, Mike Freedman, Wyatt Lloyd, Ihsan Qazi, Daniel Firestone, Praveen Tammana, Rob Harrison, Sean Choi, and the anonymous SIGCOMM reviewers for their valuable feedback that helped improve the quality of this paper. We also thank Nick McKeown and Ben Pfaff for their invaluable support at various stages of this project. This research was supported by National Science Foundation (NSF) Awards CNS-1704077 and AitF-1535948.

REFERENCES

[1] Akka: Build Powerful Reactive, Concurrent, and Distributed Applications more Easily – Using UDP. https://doc.akka.io/docs/akka/2.5.4/java/io-udp.html. Accessed on 01/25/2019.
[2] Amazon Web Services. https://aws.amazon.com. Accessed on 01/25/2019.
[3] Arstechnica: Amazon cloud has 1 million users. https://arstechnica.com/information-technology/2016/04/amazon-cloud-has-1-million-users-and-is-near-10-billion-in-annual-sales. Accessed on 01/25/2019.
[4] Cloud Networking: IP Broadcasting and Multicasting in Amazon EC2. https://blogs.oracle.com/ravello/cloud-networking-ip-broadcasting-multicasting-amazon-ec2. Accessed on 01/25/2019.
[5] Deploying Secure Multicast Market Data Services for Financial Services Environments. https://www.juniper.net/documentation/en_US/release-independent/nce/information-products/pathway-pages/nce/nce-161-deploying-secure-multicast-for-finserv.html. Accessed on 01/25/2019.
[6] Elmo P4 Programs. https://github.com/Elmo-MCast/p4-programs. Accessed on 06/16/2019.
[7] Galera Cluster. http://galeracluster.com. Accessed on 01/25/2019.
[8] Google Cloud Platform. https://cloud.google.com. Accessed on 01/25/2019.
[9] JGroups: A Toolkit for Reliable Messaging. http://www.jgroups.org/overview.html. Accessed on 01/25/2019.
[10] Jgroups: Reducing Chattiness. http://www.jgroups.org/manual/html/user-advanced.html#d0e3130. Accessed on 01/25/2019.
[11] Microsoft Azure. https://azure.microsoft.com. Accessed on 01/25/2019.

[12] Multicast in the Data Center Overview. https://datatracker.ietf.org/doc/html/draft-mcbride-armd-mcast-overview-02. Accessed on 01/25/2019.

[13] Netronome: Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/. Accessed on 01/25/2019.

[14] OpenStack: Open source software for creating private and public clouds. https://www.openstack.org. Accessed on 01/25/2019.

[15] Trading Floor Architecture. https://www.cisco.com/c/en/us/td/docs/solutions/Verticals/Trading_Floor_Architecture-E.html. Accessed on 01/25/2019.

[16] VMware vSphere: The Efficient and Secure Platform for Your Hybrid Cloud. https://www.vmware.com/products/vsphere.html. Accessed on 01/25/2019.

[17] VXLAN scalability challenges. https://blog.ipspace.net/2013/04/vxlan-scalability-challenges.html. Accessed on 01/25/2019.

[18] Adrichem, N. L. M. v., Asten, B. J. v., and Kuipers, F. A. Fast Recovery in Software-Defined Networks. In *IEEE EWSDN* (2014).

[19] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V. T., Matus, F., Pan, R., Yadav, N., and Varghese, G. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM* (2014).

[20] Amazon. Overlay Multicast in Amazon Virtual Private Cloud. https://aws.amazon.com/articles/overlay-multicast-in-amazon-virtual-private-cloud/. Accessed on 01/25/2019.

[21] Amazon Web Services (AWS). Frequently Asked Questions. https://aws.amazon.com/vpc/faqs/. Accessed on 01/25/2019.

[22] Andreyev, A. Introducing Data Center Fabric, The Next-Generation Facebook Data Center Network. https://code.facebook.com/posts/360346274145943/. Accessed on 01/25/2019.

[23] Arista. 10Gb Ethernet – The Foundation for Low-Latency, Real-Time Financial Services Applications and Other, Latency-Sensitive Applications. https://www.arista.com/assets/data/pdf/Whitepapers/Arista_Solarflare_Low_Latency_10GbE__1_.pdf. Accessed on 01/25/2019.

[24] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. A View of Cloud Computing. *Communications of the ACM (CACM) 53*, 4 (Apr. 2010), 50–58.

[25] Balakrishnan, M., Birman, K., Phanishayee, A., and Pleisch, S. Ricochet: Lateral Error Correction for Time-critical Multicast. In *USENIX NSDI* (2007).

[26] Balakrishnan, M., Pleisch, S., and Birman, K. Slingshot: Time-CriticalMulticast for Clustered Applications. In *IEEE NCA* (2005).

[27] Ballardie, T., Francis, P., and Crowcroft, J. Core Based Trees (CBT). In *ACM SIGCOMM* (1993).

[28] Banerjee, S., Bhattacharjee, B., and Kommareddy, C. Scalable Application Layer Multicast. In *ACM SIGCOMM* (2002).

[29] Barefoot. Barefoot Tofino: World's fastest P4-programmable Ethernet switch ASICs. https://barefootnetworks.com/products/brief-tofino/. Accessed on 01/25/2019.

[30] Bloom, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM (CACM) 13*, 7 (July 1970), 422–426.

[31] Boivie, R., Feldman, N., and Metz, C. Small group multicast: A new solution for multicasting on the internet. *IEEE Internet Computing 4*, 3 (2000), 75–79.

[32] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming protocol-independent packet processors. *ACM Computer Communication Review (CCR) 44*, 3 (July 2014), 87–95.

[33] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM* (2013).

[34] Budiu, M., and Dodd, C. The $P4_{16}$ Programming Language. *ACM SIGOPS Operating Systems Review 51*, 1 (Sept. 2017), 5–14.

[35] Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. SplitStream: High-bandwidth Multicast in Cooperative Environments. In *ACM SOSP* (2003).

[36] Cavium. XPliant® Ethernet Switch Product Family. https://www.cavium.com/xpliant-ethernet-switch-product-family.html. Accessed on 01/25/2019.

[37] Choi, S., Long, X., Shahbaz, M., Booth, S., Keep, A., Marshall, J., and Kim, C. The case for a flexible low-level backend for software data planes. In *ACM APNet* (2017).

[38] Cisco. Cisco Nexus 5000 Series - hardware multicast snooping group-limit. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus5000/sw/command/reference/multicast/n5k-mcast-cr/n5k-igmpsnp_cmds_h.html. Accessed on 01/25/2019.

[39] Cisco. IP Multicast Best Practices for Enterprise Customers. https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/multicast-enterprise/whitepaper_c11-474791.pdf. Accessed on 01/25/2019.

[40] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In *USENIX NSDI* (2005).

[41] Consul. Frequently Asked Questions. https://www.consul.io/docs/faq.html#q-does-consul-rely-on-udp-broadcast-or-multicast-. Accessed on 01/25/2019.

[42] Costa, L. H. M. K., Fdida, S., and Duarte, O. Hop by Hop Multicast Routing Protocol. In *ACM SIGCOMM* (2001).

[43] Crowcroft, J., and Paliwoda, K. A Multicast Transport Protocol. In *ACM SIGCOMM* (1988).

[44] Cui, W., and Qian, C. Dual-structure Data Center Multicast using Software Defined Networking. *arXiv preprint arXiv:1403.8065* (2014).

[45] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *USENIX NSDI* (2008).

[46] Dainese, A. VXLAN on VMware NSX: VTEP, Proxy, Unicast/Multicast/Hybrid Mode. http://www.routereflector.com/2015/02/vxlan-on-vmware-nsx-vtep-proxy-unicastmulticasthybrid-mode/. Accessed on 01/25/2019.

[47] Das, A., Gupta, I., and Motivala, A. SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol. In *IEEE DSN* (2002).

[48] Deering, S., Estrin, D., Farinacci, D., Jacobson, V., Liu, C.-G., and Wei, L. An Architecture for Wide-area Multicast Routing. In *ACM SIGCOMM* (1994).

[49] Diot, C., Levine, B. N., Lyles, B., Kassem, H., and Balensiefen, D. Deployment Issues for the IP Multicast Service and Architecture. *IEEE Network 14*, 1 (2000), 78–88.

[50] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., and Carle, G. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM IMC* (2015).

[51] Firestone, D. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *USENIX NSDI* (2017).

[52] Floyd, S., Jacobson, V., McCanne, S., Liu, C.-G., and Zhang, L. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *ACM SIGCOMM* (1995).

[53] Garg, N. *Learning Apache Kafka, Second Edition*, 2nd ed. Packt Publishing, 2015.

[54] Gibb, G. Data-center Parse Graph. https://github.com/grg/parser-gen/blob/master/examples/headers-datacenter.txt. Accessed on 01/25/2019.

[55] Gibb, G. Enterprise Parse Graph. https://github.com/grg/parser-gen/blob/master/examples/headers-enterprise.txt. Accessed on 01/25/2019.

[56] Gibb, G., Varghese, G., Horowitz, M., and McKeown, N. Design Principles for Packet Parsers. In *ACM/IEEE ANCS* (2013).

[57] Google. Cloud APIs. https://cloud.google.com/apis/. Accessed on 01/25/2019.

[58] Google. Cloud Pub/Sub. https://cloud.google.com/pubsub/. Accessed on 01/25/2019.

[59] Google Cloud Platform. Frequently Asked Questions. https://cloud.google.com/vpc/docs/vpc. Accessed on 01/25/2019.

[60] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM* (2009).

[61] Gulati, A., Ahmad, I., and Waldspurger, C. A. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *USENIX FAST* (2009).

[62] Handley, M. J., and Widmer, J. TCP-Friendly Multicast Congestion Control (TFMCC): Protocol Specification. RFC 4654, Aug. 2006.

[63] Hintjens, P. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[64] Huang, D. Y., Yocum, K., and Snoeren, A. C. High-fidelity Switch Models for Software-defined Network Emulation. In *ACM HotSDN* (2013).

[65] Huang, K., and Su, X. Scalable Datacenter Multicast using In-packet Bitmaps. *Springer Distributed and Parallel Databases 36*, 3 (2018), 445–460.

[66] Iyer, A., Kumar, P., and Mann, V. Avalanche: Data Center Multicast using Software Defined Networking. In *IEEE COMSNETS* (2014).

[67] Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and O'Toole, Jr., J. W. Overcast: Reliable Multicasting with on Overlay Network. In *USENIX OSDI* (2000).

[68] Jia, W.-K. A Scalable Multicast Source Routing Architecture for Data Center Networks. *IEEE JSAC 32*, 1 (2013), 116–123.

[69] Jokela, P., Zahemszky, A., Esteve Rothenberg, C., Arianfar, S., and Nikander, P. LIPSIN: Line Speed Publish/Subscribe Inter-networking. In *ACM SIGCOMM* (2009).

[70] Judge, P., and Ammar, M. Security issues and solutions in multicast content distribution: A survey. *IEEE Network 17*, 1 (2003), 30–36.

[71] Juniper. Understanding VXLANs. https://www.juniper.net/documentation/en_US/junos/topics/topic-map/sdn-vxlan.html. Accessed on 01/25/2019.

[72] Katta, N., Ghag, A., Hira, M., Keslassy, I., Bergman, A., Kim, C., and Rexford, J. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *ACM CoNEXT* (2017).

[73] Katta, N., Hira, M., Kim, C., Sivaraman, A., and Rexford, J. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM SOSR* (2016).

[74] Kim, C., Bhide, P., Doe, E., Holbrook, H., Ghanwani, A., Daly, D., Hira, M., and Davie, B. In-band Network Telemetry (INT). *P4 Consortium* (2018).

[75] Komolafe, O. IP Multicast in Virtualized Data Centers: Challenges and Opportunities. In *IFIP/IEEE IM* (2017).

[76] Kuźniar, M., Perešíni, P., Kostić, D., and Canini, M. Methodology, Measurement and Analysis of Flow Table Update Characteristics in Hardware OpenFlow Switches. *Computer Networks 136* (2018), 22–36.

[77] Lamport, L. The Part-time Parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (May 1998), 133–169.
[78] Lamport, L. Fast Paxos. *Springer Distributed Computing 19*, 2 (2006), 79–103.
[79] Lehman, L.-W. H., Garland, S. J., and Tennenhouse, D. L. Active reliable multicast. In *IEEE INFOCOM* (1998).
[80] Li, D., Cui, H., Hu, Y., Xia, Y., and Wang, X. Scalable Data Center Multicast using Multi-class Bloom Filter. In *IEEE ICNP* (2011).
[81] Li, M., Anderson, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX OSDI* (2014).
[82] Li, S., Maddah-Ali, M. A., and Avestimehr, A. S. Coded MapReduce. In *IEEE Communication, Control, and Computing* (2015).
[83] Li, X., and Freedman, M. J. Scaling IP Multicast on Datacenter Topologies. In *ACM CoNEXT* (2013).
[84] Lin, Y.-D., Lai, Y.-C., Teng, H.-Y., Liao, C.-C., and Kao, Y.-C. Scalable Multicasting with Multiple Shared Trees in Software Defined Networking. *Journal of Network and Computer Applications 78*, C (Jan. 2017), 125–133.
[85] Mahalingam, M., Sridhar, T., Bursell, M., Kreeger, L., Wright, C., Duda, K., Agarwal, P., and Dutt, D. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, Oct. 2015.
[86] Mai, L., Hong, C., and Costa, P. Optimizing Network Performance in Distributed Machine Learning. In *USENIX HotCloud* (2015).
[87] Massie, M. L., Chun, B. N., and Culler, D. E. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Elsevier Parallel Computing 30*, 7 (2004), 817–840.
[88] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM Computer Communication Review (CCR) 38*, 2 (Mar. 2008), 69–74.
[89] Microsoft Azure. Cloud Service Fundamentals – Telemetry Reporting. https://azure.microsoft.com/en-us/blog/cloud-service-fundamentals-telemetry-reporting/. Accessed on 01/25/2019.
[90] Microsoft Azure. Frequently Asked Questions. https://docs.microsoft.com/en-us/azure/virtual-network/virtual-networks-faq. Accessed on 01/25/2019.
[91] Network World. Multicast Group Capacity: Extreme Comes Out on Top. https://www.networkworld.com/article/2241579/virtualization/multicast-group-capacity--extreme-comes-out-on-top.html. Accessed on 01/25/2019.
[92] Niranjan Mysore, R., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., and Vahdat, A. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *ACM SIGCOMM* (2009).
[93] Open Config. Streaming Telemetry. http://blog.sflow.com/2016/06/streaming-telemetry.html. Accessed on 01/25/2019.
[94] P4 Language Consortium. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf. Accessed on 01/25/2019.
[95] P4 Language Consortium. P4Runtime Specification. https://s3-us-west-2.amazonaws.com/p4runtime/docs/v1.0.0-rc4/P4Runtime-Spec.pdf. Accessed on 01/25/2019.
[96] Panchen, S., McKee, N., and Phaal, P. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176, Sept. 2001.
[97] Pepelnjak, I. FIB update challenges in OpenFlow networks. https://blog.ipspace.net/2012/01/fib-update-challenges-in-openflow.html. Accessed on 01/25/2019.
[98] Pfaff, B., Pettit, J., Amidon, K., Casado, M., Koponen, T., and Shenker, S. Extending Networking into the Virtualization Layer. In *ACM HotNets* (2009).
[99] Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. The Design and Implementation of Open vSwitch. In *USENIX NSDI* (2015).
[100] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing Distributed Systems using Approximate Synchrony in Data Center Networks. In *USENIX NSDI* (2015).
[101] Ratnasamy, S., Ermolinskiy, A., and Shenker, S. Revisiting IP Multicast. In *ACM SIGCOMM* (2006).
[102] Roy, A., Zeng, H., Bagga, J., and Snoeren, A. C. Passive Realtime Datacenter Fault Detection and Localization. In *USENIX NSDI* (2017).
[103] Samadi, P., Gupta, V., Birand, B., Wang, H., Zussman, G., and Bergman, K. Accelerating Incast and Multicast Traffic Delivery for Data-intensive Applications Using Physical Layer Optics. In *ACM SIGCOMM* (2014).
[104] Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., and Rexford, J. PISCES: A Programmable, Protocol-Independent Software Switch. In *ACM SIGCOMM* (2016).
[105] Singla, A., Hong, C.-Y., Popa, L., and Godfrey, P. B. Jellyfish: Networking Data Centers Randomly. In *USENIX NSDI* (2012).
[106] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM* (2016).
[107] Speakman, T., Crowcroft, J., Gemmell, J., Farinacci, D., Lin, S., Leshchiner, D., Luby, M., Montgomery, T. L., Rizzo, L., Tweedly, A., Bhaskar, N., Edmonstone, R., Sumanasekera, R., and Vicisano, L. PGM Reliable Transport Protocol Specification. RFC 3208, Dec. 2001.
[108] Tang, C., Steinder, M., Spreitzer, M., and Pacifici, G. A Scalable Application Placement Controller for Enterprise Data Centers. In *ACM WWW* (2007).
[109] Valadarsky, A., Shahaf, G., Dinitz, M., and Schapira, M. Xpander: Towards Optimal-Performance Datacenters. In *ACM CoNEXT* (2016).
[110] Videla, A., and Williams, J. J. *RabbitMQ in Action: Distributed Messaging for Everyone*. Manning, 2012.
[111] Vigfusson, Y., Abu-Libdeh, H., Balakrishnan, M., Birman, K., Burgess, R., Chockler, G., Li, H., and Tock, Y. Dr. Multicast: Rx for Data Center Communication Scalability. In *ACM EuroSys* (2010).
[112] Vinterbo, S. A Note on the Hardness of the K-ambiguity Problem. Tech. rep., Technical report, Harvard Medical School, Boston, MA, USA, 2002.
[113] VMware. NSX Network Virtualization and Security Platform. https://www.vmware.com/products/nsx.html. Accessed on 01/25/2019.
[114] VMware. Recommended Configuration Maximums, NSX for vSphere 6.3 (Update 2). https://docs.vmware.com/en/VMware-NSX-for-vSphere/6.3/NSX%20for%20vSphere%20Recommended%20Configuration%20Maximums_63.pdf. Accessed on 01/25/2019.
[115] Weave Works. Multicasting in the Cloud: What You Need to Know. https://www.weave.works/blog/multicasting-cloud-need-know/. Accessed on 01/25/2019.
[116] Widmer, J., and Handley, M. Extending Equation-based Congestion Control to Multicast Applications. In *ACM SIGCOMM* (2001).
[117] Wijnands, I., Rosen, E. C., Dolganow, A., Przygienda, T., and Aldrin, S. Multicast Using Bit Index Explicit Replication (BIER). RFC 8279, Nov. 2017.

Appendices are supporting material that has not been peer reviewed.

A *P*-RULE LOOKUPS USING MATCH-ACTION STAGES IS EXPENSIVE – STRAWMAN

Lookups in network switches are typically done using match-action tables, after the parser. We could do the same for *p*-rules, but using match-action tables to lookup *p*-rules would result in inefficient use of switch resources. Unlike *s*-rules, *p*-rules are headers. Hence, to match on *p*-rules, we need a table that matches on all *p*-rule headers. In each flow rule, we only match the switch identifier with one *p*-rule, while wildcarding the rest. This is a constraint of match-action tables in switches that we cannot avoid. To match N *p*-rules, we need same number of flow-table entries.

The fundamental problem here is that instead of increasing the *depth*, *p*-rules increase the *width* of a table. Modern programmable switches can store millions of flow-table entries (depth). However, they are severely limited by the number of headers they can match on in a stage (width). For example, in case of RMT [33], a match-action stage consists of 106 1,000 x 112b SRAM blocks and 16 2,000 x 40b TCAM blocks. These blocks can combine together to build wider or deeper SRAMs and TCAMs to make larger tables. For example, to implement a table that matches on ten *p*-rules, each 11-bit wide, we need three TCAM blocks (as we need wildcards) to cover 110b for the match. This results in a table of 2,000 entries x 120b wide. And out of these 2,000 entries, we only use ten entries to match the respective *p*-rules. Thus, we end up using three TCAMs for ten *p*-rules while consuming only 0.5% of entries in the table, wasting 99.5% of the entries (which cannot be used by any other stage).

An alternative to using TCAMs for *p*-rule lookups is to eschew wildcard lookups and use SRAM blocks. In this case, a switch needs N stages to lookup N *p*-rules in a packet, where each stage only has a single rule. This too is prohibitively expensive. First, the number of stages in a switch is limited (RMT has 16 stages for the ingress

pipeline). Second, as with TCAMs, 99.9% of the SRAM entries go to waste, as each SRAM block is now used only for a single *p*-rule each (out of 1,000 available entries per block).