Elmo: Source Routed Multicast for Public Clouds

Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Fellow, IEEE, Nick Feamster, Ori Rottenstreich, and Mukesh Hira

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 data-center 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 (max. 325 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.

Index Terms—Multicast, source routing, bitmap encoding, P4, PISA.

I. INTRODUCTION

To quote the Oracle Cloud team [1], we believe that the lack of multicast support among public cloud providers [2], [3] is a huge “missed opportunity.” First, IP multicast is in widespread use in enterprise datacenters to support virtualized workloads (e.g., VXLAN and NVGRE) [4], [5] and by financial services to support stock tickers and trading workloads [6], [7]. 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 [8], [9], replicated state machines [10], [11], publish-subscribe systems [12], database replication [13], messaging middleware [14], [15], data-analytics platforms [16], and parameter sharing in distributed machine learning [17], [18]. 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 [19], [20]—indeed, many commercial offerings exist in this space [21]. This leads to inefficiencies for both tenants and providers alike (§V-B.1); performing packet replication on end-hosts instead of the network not only inflates CPU load in datacenters, but also prevents tenants from sustaining 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 [4]. 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 [22], [23]. These group-table sizes are a challenge even in small enterprise datacenters [24], let alone at the scale of public clouds that host hundreds of thousands of tenants [25]. On the control-plane side, IP multicast has historically suffered from having a “chatty” control plane [26], [27], which is a cause for concern [1] in public clouds with a shared network where tenants introduce significant churn in the multicast state (e.g., due to virtual machine allocation [28], [29] and migration [30], [31]). Protocols like IGMP and PIM trigger many control messages during churn and periodically query the entire broadcast domain, while the SDN-based solutions [32], [33] 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 [34] and the unique characteristics of datacenters (namely, hypervisor switches [35], symmetric topology, and short paths [36], [37]), 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 [34].
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 [38], [39], 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 [40], Li and Freedman [33]). Or, they expect unorthodox switching capabilities that are infeasible to implement in today’s datacenters, and yet, only work for smaller networks (BIER [41]) or with small group sizes (SGM [42]). Or, they have high traffic and end-host overheads, and therefore, cannot support multicast at line rate [19], [43]. 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 [4]; 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 §VII.

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 (§III), the prevailing topology in today’s datacenters [36], [37]. 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 (§IV). Our evaluation shows that our encoding facilitates a feasible implementation in today’s multi-tenant datacenters (§V). 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 (§V-B.1) and host telemetry (§V-B.2).

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.

II. 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.

a) 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 [44] 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 [45] (using fault-tolerant distributed directory systems [36]). The controller relies on a high-level language (like P4 [46]) 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 [47]) 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 (§III-B).

Definition 1: (p-rule) A p-rule contains a set of output ports along with zero or more switch identifiers, encoded as headers in a packet (Figure 2).

Definition 2: (s-rule) An s-rule is a group-table entry in a network switch, consisting of an IP address and list of ports.

¹Today’s datacenter controllers are capable of executing these steps in sub-second timescales [45] and can handle concurrent and consistent updates to tens of thousands of switches [48].
switch forwards the packet based on a default packet header (as short). As such, if a packet header contains no matching rules, the network simply parses the header to look for a matching destination IP address (multicast group) and forwards the packet, a physical switch (or network switch) running inside a switch, e.g., a maximum of five hops in the Facebook Fabric topology [37].

A. 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 switch identifier to forward the packet, a physical switch running inside a switch, e.g., a maximum of five hops in the Facebook Fabric topology [37].

III. 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).

A. 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. We formally represent a p-rule as follows: d-bits(u-bits(optional)|M(optional)|p-rules); for example, d-bits = 11|u-bits = 00|M = 1:1|p-rules = 10; for the upstream rule for L0, and d-bits = 10; for the downstream rule for spine switches S0 and S1.

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 H0 and H1, 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 [34]. Alternative encoding strategies use destination group members, encoded as bit strings [41]; Bloom filters, representing link memberships [51]; or simply a list of IP addresses [42] to 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 §VII).

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 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:

1. **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 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, $10:\{P_3\}$.

2. **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 [52], or HULA [53]) 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, §III-C). In Figure 3b, for the Elmo packet originating from host $H_a$, leaf $L_0$ will use the upstream $p$-rule ($01:[P_2]$) to forward the packet to the neighboring host $H_b$ as well as multipathing it to the logical spine $P_0$ ($S_0$, $S_1$).

3. **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).

4. **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 §IV. 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).

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 [34]), 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.\(^3\) 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) [52]. The algorithm to identify sets of switches with similar bitmaps is described in §III-B.

We encode the set of switches as a simple list of switch identifiers, as shown in Figure 2b. Alternate encodings, such as Bloom filters [55], 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 [33], 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., $R = 2$, §III-B), 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 $L_4$ and $L_5$ with $L_7$.

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.

\(^3\)We can configure the logical-to-physical port mapping in network switches such that each logical port connects to the same downstream switch; hence, allowing these switches to share a single bitmap [54].

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).

B. 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).

a) 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 ($§V$).

A network switch has space for at most $F_{max}$ $s$-rules (typically thousands to a few tens of thousands [22], [23], [33]), 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 ($§V$) shows that it is still effective with negligible traffic overhead over ideal multicast.

b) 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 [56]. 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) [56]. 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 fall back 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).

For example, in Figure 3a, with $\#p$-rules ($H_{max}$) = 2, max switches per $p$-rule ($K_{max}$) = 2, $\#s$-rules ($F_{max}$) = 1, and $R = 0$, the leaf switches ($L_0$ and $L_0$) are assigned the same bitmap, and leaf $L_7$ is mapped to the default bitmap, 01.

C. 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 [57].

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 [45] 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 §V-A.3.

IV. 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 [35] and Barefoot Tofino [58].

$^4$Example P4 programs for network and hypervisor switches based on the datacenter topology in Figure 3a are available on GitHub [59].

<table>
<thead>
<tr>
<th>Algorithm 1 Clustering Algorithm for Each Layer of a Group</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constants:</strong> $R$, $H_{max}$, $K_{max}$, $F_{max}$</td>
</tr>
<tr>
<td><strong>Inputs:</strong> Set of all switches $S$, Bitmaps $B = b_i \forall i \in S$</td>
</tr>
<tr>
<td><strong>Outputs:</strong> $p$-rules, $s$-rules, and default-$p$-rule</td>
</tr>
<tr>
<td>1: $p$-rules $\leftarrow \emptyset$, $s$-rules $\leftarrow \emptyset$, default-$p$-rule $\leftarrow \emptyset$</td>
</tr>
<tr>
<td>2: unassigned $\leftarrow B$, $K = K_{max}$</td>
</tr>
<tr>
<td>3: while unassigned $\not\in \emptyset$ and $</td>
</tr>
<tr>
<td>4: bitmaps $\leftarrow$ approx-min-k-union($K$, unassigned)</td>
</tr>
<tr>
<td>5: output-bm $\leftarrow$ Bitwise OR of all $b_i \in$ bitmaps</td>
</tr>
<tr>
<td>6: if $\text{dist}(b_i, \text{output-bm}) \leq R \forall b_i \in$ bitmaps then</td>
</tr>
<tr>
<td>7: $p$-rules $\leftarrow p$-rules $\cup$ bitmaps</td>
</tr>
<tr>
<td>8: unassigned $\leftarrow$ unassigned $\setminus$ bitmaps</td>
</tr>
<tr>
<td>9: else</td>
</tr>
<tr>
<td>10: $K \leftarrow K - 1$</td>
</tr>
<tr>
<td>11: for all $b_i \in$ unassigned do</td>
</tr>
<tr>
<td>12: if $\text{switch i has} \</td>
</tr>
<tr>
<td>13: $s$-rules $\leftarrow s$-rules $\cup {b_i}$</td>
</tr>
<tr>
<td>14: else</td>
</tr>
<tr>
<td>15: default-$p$-rule $\leftarrow$ default-$p$-rule $</td>
</tr>
<tr>
<td>return $p$-rules, $s$-rules, default-$p$-rule</td>
</tr>
</tbody>
</table>

switches entail multiple challenges in efficiently parsing, matching, and acting on $p$-rules.

A. 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 (MATs) to perform this matching is prohibitively expensive, resulting in inefficient use of switch resources (i.e., flow-table entries and MAT stages). Instead, we present an efficient implementation by exploiting the match-and-set capabilities of parsers in modern programmable data planes.

a) 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 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 [34], [60]. 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 the last rule (i.e., “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 [34] the size is 512 bytes. We show in §V-A, how Elmo’s encoding scheme easily fits enough p-rules using a header-size budget of 325 bytes (mean 114 bytes) 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.

b) 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 [34], using a bitmap_port_select primitive\(^5\); 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-capsulation. Otherwise, the egress pipeline invalidates all p-rules up to the p-rules(s) of the next-hop switch before forwarding the packet.

B. 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 [35]) 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 (§V-C). Not doing so decreases throughput linearly with increasing number of p-rules.

V. EVALUATION

In this section, we evaluate the data- and control-plane scalability requirements of Elmo, as well as end-host and hardware resource overheads.

- **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 (§V-A).
- **Applications run unmodified.** and benefit from reduced CPU and bandwidth utilization for multicast workloads (§V-B).
- **End-host resource requirements:** Elmo adds negligible overheads to hypervisor switches (§V-C).
- **Hardware resource requirements:** Elmo is inexpensive to implement in programmable switching ASICs (§VI).

A. Scalability

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 multicast groups of varying sizes to each tenant. We simulate using a Facebook Fabric topology—a three-tier topology—with 12 pods [37]. 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 [52].)

**Tenant VMs and placement.** Mimicking the experiment setup from Li and Freedman [33]; 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 [62], running alongside the network controller [63], [64]. 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 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 [33]. 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).

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 [34])—leaving 400 bytes (mean) for other protocols, which in enterprises and datacenters consume about 90 bytes [65]. 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 size suffices 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 switches of having precisely the 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 and Freedman [33] with no limit on VMs of a tenant packed per rack (P = All). (Aside from these many group-table entries, Li et al’s scheme [33] 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 (§III-A), 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 (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. [33], 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 [33] 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, 6In 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.
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, but still being only 40\% over ideal multicast at \( p \) increases the number of switches that are mapped to the default memory on the efficiency of the encoding scheme. Doing so reduced the number of events per group is proportional to the group size, which follows the WVE distribution.

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 dispersion 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 [66] or Jellyfish [67]); 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.

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.

- **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 to leave. 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 [68]. As shown in Table I, 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 [33], 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 [38] respectively, implying that Elmo leaves enough spare control-traffic capacity for other protocols in datacenters to function.

- **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 [69]) for some groups to minimize transient loss while the network is reconfiguring (§III-C). 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 can handle batched updates of 80,000 requests per second 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 [45]. 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.
B. Evaluating End-to-End Applications

We ran two popular datacenter applications on top of Elmo: ZeroMQ [70] and sFlow [71]. 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 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 [35] 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.

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 [72] or GCP [73]), 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%).

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).

C. End-Host Microbenchmarks

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

Fig. 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.

Fig. 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.

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 [74] 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.

VI. HARDWARE RESOURCE USAGE

We study the hardware resource requirements of programmable switching ASICs to process p-rules, and synthesize Elmo against RMT [34]—a switch chip based on the 28 nm process technology. We found Elmo inexpensive to implement in such ASICs.

1) Header Usage With Varying Number of P-Rules: Figure 8 shows percentage header usage—for a chip that can parse a 512-byte packet header (e.g., RMT [34])—as we increase the number of p-rules. Each p-rule consists of four bytes for switch identifiers and a 48-bit bitmap. With 30 p-rules, we are still well within the range, consuming only 63.5% of header space (and 45% of parser’s TCAM) for p-rules with 190 bytes available for other protocols. We evaluated these results using the open-source compiler for P4’s behavioral model (i.e., bmv1 [75]).

7The area impact of Elmo will reduce further at lower technology nodes (e.g., 16 nm and 7 nm).
2) Enabling Bitmap-Based Output Port Selection: Data-plane languages (like P4 [76]) do not yet expose the output-port bit vector, network switches use for replicating packets [34], as a metadata field that a parser can set. We add support for specifying this bit vector using a new primitive action in P4. We call this new primitive `bitmap_port_select`. It takes a bitmap of size \( N \) as input and sets the output-port bit vector field that a queue manager then uses to generate copies of a packet, routed to each egress port. The function is executed by a match-action stage in the ingress pipeline before forwarding the packet to the queue manager. We evaluate the primitive using Synopsys 28/32 nm standard cell technology [77], synthesized with a 1 GHz clock. All configurations meet the timing requirements at 1 GHz.

A typical ASIC implements multicast using a group table that maps a group identifier to a bit vector [78]. We implement a multiplexer block representing the hardware requirements to pass this bit vector directly to the queue manager; and a shift register showing the hardware required to pass the bit vector from the parser, through all stages of the ingress pipeline, and up to the queue manager in an RMT-style switching ASIC [34], [58]. We compute the resource requirements for four switch-port counts: 32, 64, 128, 256.

We quantify the area cost and power requirements of adding the multiplexer and shift register blocks. We evaluate the area cost against a 200 mm\(^2\) baseline switching chip (assuming a lower-end chip and the smallest area given by [65]). Figure 9 shows that the added area cost for a multiplexer block is a nominal 0.0015\% (2,984 mm\(^2\)) for passing a 256-bit wide vector to the queue manager (and a corresponding 5.31 mW in power). Using a shift register (Figure 10) further increases the area usage by 0.05\% (114,150.12 mm\(^2\)) for 256-bit wide vectors for a 32-stage pipeline (and 171.11 mW in power). As another point of comparison for the reader, CONGA [52] and Banzai [61] consume an additional 2\% and 12\% area, respectively.

The circuit delay for the multiplexer is 120 ps (mean) for each of the four tested port counts. The circuit delay of a single stage in shift register is 150 ps (mean).

VII. RELATED WORK

Table 11 highlights various areas where related multicast approaches fall short compared to Elmo in the context of today’s cloud environments.

8They do so to support legacy multicast; otherwise, in private communications with switch vendors, we did not find any real tradeoff in passing the bit vector directly.
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.

d) Source-Routed Multicast: Elmo is not the first system to encode forwarding state inside packets. Previous work [51], [82] have tried to encode link identifiers inside packets using Bloom filters. BIER [41] encodes group members as bit strings, whereas SGM [42] 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 [41], multiple lookups on a single table [42], division operators [88], run-length encoding [89], 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 [58] and Cavium XPliant [90]) and supports hundreds of groups per tenant in a cloud datacenter.

VIII. 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.

a) 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.

b) Reliability and Security: Elmo supports the same best-effort delivery semantics of native multicast. For reliability, multicast protocols like PGM [91] and SRM [92] 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 [93], dropping malicious packets (e.g., DDoS [84]) before they even reach the network.

c) 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 [94]) 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).

d) 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.

REFERENCES


P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, Synopsys. Synopsys 32/28nm and 90nm Generic Libraries

IP Multicast Best Practices for Enterprise Customers

Google Cloud Platform

P4 Compiler for the Behavioral Model

Barefoot.

P. Bosshart

Amazon Web Services

D. Y. Huang, K. Yocum, and A. C. Snoeren, “High-fidelity switch
InMon Corporation’s sFlow: A


P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and

tion placement controller for enterprise data centers,” in Proc. 16th Int.

OpenStack: Open Source Software for Creating Private and

VMware vSphere: The Efficient and Secure Platform for Your

G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, “Design princi-

A. Valadarsky, G. Shahaf, M. Dinitz, and M. Schapira, “Xpander:
Towards optimal-performance datacenters,” in Proc. ACM CoNEXT,
2016, pp. 205–219

A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, “Jellyfish: Network-

D. Y. Huang, K. Yocum, and A. C. Snoeren, “High-fidelity switch
models for software-defined network emulation;” in Proc. 2nd ACM
SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN), 2013,
pp. 43–48

recovery in software-defined networks;” in Proc. 3rd Eur. Workshop Softw.

P. Hintijens, ZeroMQ: Messaging for Many Applications. Newton, MA,
USA: O’Reilly Media, 2013.

S. Panchen, N. McKee, and P. Phaal, InMon Corporation’s sFlow: A
Method for Monitoring Traffic in Switched and Routed Networks, RFC,


P. Emmerich, S. Galleinmuller, D. Raumer, F. Wohlfart, and G. Carle,
“MoonGen: A scriptable high-speed packet generator,” in Proc. ACM
IMC, 2015, pp. 275–287.

behavioral

P. Bosshart et al., “P4: Programming protocol-independent packet
pp. 87–95, Jul. 2014.

Synopsys, Synopsys 32/24m and 90m Generic Libraries. [Online]. Available: https://www.synopsys.com/community/university-
program/teaching-resources.html


S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and
L. Wei, “An architecture for wide-area multicast routing;” in Proc.

C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefer,
“Deployment issues for the IP multicast service and architecture,” IEEE

whitepaper_c11-474791.pdf

P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and

M. Balakrishnan, K. Birman, A. Phanishayee, and S. Pleisch, “Riccochet:
Lateral error correction for time-critical multicast;” in Proc. USENIX

P. Judge and M. Ammar, “Security issues and solutions in multicast

Jennifer Rexford (Fellow, IEEE) received the B.Eng. degree in electrical engineering from Prince-
ton University, and the Ph.D. degree in electrical engineering and computer science from the Uni-
versity of Michigan. She worked at AT&T Labs-
Research for eight years. She is currently the Gordon Y.S. Wu Professor of Engineering and the Chair of
Computer Science at Princeton University.

Nick Feamster received the S.B. and M.Eng.
degrees in electrical engineering and computer sci-
ence and the Ph.D. degree in computer science from
MIT. He was a Full Professor with the Computer
Science Department, Princeton University, where
he directed the Center for Information Technology
Policy (CITP). He is currently a Neubauer Professor of
Computer Science and the Director of the Center
for Data and Computing (CDAC), University of Chicago.

Ori Rottenstreich received the B.Sc. degree in
computer engineering and the Ph.D. degree in elec-
trical engineering from Technion - Israel Institute of
Technology, Haifa, Israel. He was a Post-Doctoral
Research Fellow at Princeton University. He is cur-
cently an Assistant Professor with the Department of
Computer Science and the Department of Electrical
Engineering, Technion.

Mukesh Hira received the master’s and Ph.D.
degrees in electrical engineering from Stanford Uni-
versity. He is currently a Software Engineering
Architect at VMware, Inc., with expertise in cloud
networking, data center networking, and distributed
systems.