Orca: Server-assisted Multicast for Datacenter Networks

Khaled Diab, Parham Yassini, and Mohamed Hefeeda, *Simon Fraser University*

https://www.usenix.org/conference/nsdi22/presentation/diab-orca

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Orca: Server-assisted Multicast for Datacenter Networks

Khaled Diab Parham Yassini Mohamed Hefeeda
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada

Abstract

Group communications appear in various large-scale datacenter applications. These applications, however, do not currently benefit from multicast, despite its potential substantial savings in network and processing resources. This is because current multicast systems do not scale and they impose considerable state and communication overheads. We propose a new architecture, called Orca, that addresses the challenges of multicast in datacenter networks. Orca divides the state and tasks of the data plane among switches and servers, and it partially offloads the management of multicast sessions to servers. Orca significantly reduces the state at switches, minimizes the bandwidth overhead, incurs small and constant processing overhead, and does not limit the size of multicast sessions. We implemented Orca in a testbed to demonstrate its performance in terms of throughput, consumption of server resources, packet latency, and the impact of server failures. We also implemented a sample multicast application in our testbed, and showed that Orca can substantially reduce its communication time, through optimizing the data transfer between nodes using multicast instead of unicast. In addition, we simulated a datacenter consisting of 27,648 hosts and handling 1M multicast sessions, and we compared Orca versus the state-of-art system in the literature. Our results show that Orca reduces the switch state by up to two orders of magnitude, the communication overhead by up to 19X, and the control overhead by up to 14X, compared to the state-of-art.

1 Introduction

Many modern datacenter applications require group communications in the form of one-to-many or many-to-many patterns. Examples of these applications include distributed databases, telemetry systems, consensus protocols, and machine learning systems. Multicast can efficiently support these communication patterns. For example, in distributed databases, multicast can be used to distribute and replicate data among servers [69]. For telemetry systems, multicast is suitable for sending updates and monitoring data to collector nodes [46, 67]. In addition, multicast can be used for state machine replication tasks in the Paxos consensus protocol and its variations [21, 39, 45, 53]. Furthermore, multicast can improve the performance of iterative algorithms that distribute data from a server to multiple working nodes. Examples of such algorithms appear in training machine learning models [28], text mining [48], and recommendation systems [36].

In addition to the above applications, an efficient multicast primitive would benefit various systems that naturally perform group communication. For example, publish-subscribe systems [37, 58, 68] typically send each message to a group of receivers. These systems are the substrate for many applications such as activity trackers, log aggregators, and stream processing frameworks. Moreover, in the emerging serverless platforms [2, 60], a common pattern is that a worker communicates with multiple other workers to enroll them in a single burst computation [7, 8], which can efficiently be realized using multicast.

Despite its potential significant bandwidth savings, multicast faces multiple challenges that slow down its deployment by major cloud providers [62]. First, to forward packets on links belonging to the multicast tree, multicast forwarding requires *maintaining state at all switches* for each session, which imposes substantial memory overheads on switches. Second, *updating and refreshing this state* upon changes generates a storm of messages, which reduces the scalability of switches as they are required to process numerous control packets. Finally, since multicast trees in datacenters could potentially span many switches and servers, *encoding these trees into labels* could impose substantial processing, communication and/or bandwidth overheads.

As a result, there has been a lack of efficient and scalable multicast systems that support large numbers of sessions. For example, in practice, switch vendors are forced to limit the number of IP multicast sessions per switch [55], because of the inefficiencies introduced by the group management [50] and tree construction [16] protocols of IP multicast. This is not cost-effective for cloud providers. In addition, while

current datacenter multicast approaches, e.g., [5, 6, 32, 43], improve upon the basic IP multicast, they also do not scale well and impose substantial overheads on the network, as we show in this paper. To partially mitigate the lack of efficient multicast systems, many datacenter applications had to rely on (inefficient) application-layer protocols [51]. For example, Apache Spark [40] implements its own primitives [9] such as Cornet [36] and HTTP-based multicast.

This paper presents a new architecture, called Orca, to realize efficient multicast forwarding that can support millions of concurrent multicast sessions in datacenter networks. The idea of Orca is to *offload* some of the state maintained at network switches to end servers. To achieve this idea, Orca computes *fixed-size and compact* labels and attaches them to packets of multicast sessions. These labels effectively enable shifting some of the data plane tasks to servers. As a result, Orca significantly reduces the state at switches, minimizes the bandwidth overhead, incurs small and constant processing overhead, does not limit the size of multicast sessions, and eliminates redundant traffic. Realizing the proposed *server-assisted* multicast approach, however, faces multiple challenges at the control and data planes that Orca addresses. At the control plane, the proposed architecture needs to calculate optimized labels, manage state at servers, and handle failures. At the data plane, it requires packet processing algorithms at switches and servers that sustain the line-rate performance and minimize the latency and resource consumption.

This paper makes the following contributions.

- We introduce the idea of *server-assisted (or offloaded) multicast* for scalable multicast services in datacenters.
- We design a hierarchical control plane that efficiently manages multicast sessions and their dynamics, handles network failures, and does not impose high control overheads (§3.3 and §3.4).
- We present a scalable data plane algorithm to process multicast packets within high-speed datacenter networks, without introducing redundant traffic or requiring switches to maintain large states (§3.5).
- We design and implement APIs to transparently integrate multicast into datacenter applications (§4).
- We implement the proposed multicast approach and evaluate its performance in a testbed using programmable switches to demonstrate its practicality (§5). Our results show that the proposed approach can support high-speed traffic, uses small CPU resources at servers, and imposes small and predictable packet delays.
- We show the potential significant gains achieved by using multicast instead of unicast in datacenter applications. We implemented a sample application using Orca and the unicast approach used in current systems such as Apache Spark [40]. For this application that has only 12 receivers, our results show that Orca can reduce the communication time by almost an order of magnitude; larger savings are expected for applications with more receivers. In addition, since an Orca sender transmits only one copy per packet regardless of the number of receivers in the session, the required CPU resources are significantly reduced, compared to using unicast.
- We compare Orca against the closest system in the literature, Elmo [5], in large-scale simulations (§6). Our results show that Orca reduces the switch state by up to two orders of magnitude, the communication overhead by up to 19X, and the control overhead by up to 14X compared to Elmo in large-scale datacenter networks.

2 Related Work

Internet Multicast. IP multicast is not practical for datacenter networks because of its limited scalability for both the control and data planes [11, 52]. Specifically, its group management and tree construction protocols, e.g., IGMP [50] and PIM [16], need to maintain state at routers belonging to each multicast session. Moreover, to refresh this state, these protocols generate control messages that routers need to process. These overheads limit the number of multicast sessions, and they could delay a receiver joining a session for up to 23 seconds [66], which is not practical for datacenters. Furthermore, current multicast approaches designed for ISP networks, e.g., [1, 43], introduce significant communication overheads, and thus they are not suitable for datacenter networks.

Datacenter Multicast. Multiple approaches, e.g., [5, 32, 35], attempted to address the challenges of IP multicast. Li et al. [35] propose a multi-class Bloom filter (MBF) to support multicast in datacenter networks. For every interface, MBF uses a Bloom filter to store whether packets of a session should be duplicated on that interface. MBF may introduce redundant traffic due to the probabilistic nature of Bloom filters. Li and Freedman [32] partition the IP address space and aggregate addresses for different sessions. However, this approach consumes the limited flow table resources in switches and limits the number of supported multicast sessions. Elmo [5] encodes links of a multicast tree into rules to be attached to packets and maintained at switches. Elmo is the state-of-art multicast system for datacenters, and we compare against it.

Other works, e.g., [9, 10, 25], enabled multicast in circuit-switched datacenter networks. For example, Republic [9] and Blast [25] realize multicast by using additional optical circuit switches. Orca is designed to be deployed in the common packet-switched networks. Application-layer multicast approaches could also be used in datacenters, by concurrently sending unicast flows to multiple receivers. This, however, results in inefficient bandwidth utilization [47, 49, 51], and increases the CPU load on the sender.

Server-assisted Data Planes. While Orca is the first server-assisted multicast for datacenters, to the best of our knowledge,

it is not the first work to utilize server resources to implement parts of the data plane. For examples, Katta et al. [17] propose an OpenFlow [26] rule caching system using both switch TCAM and server memory, which is managed by a controller. In contrast, we design Orca to reduce the state maintained at switches instead of improving how the large number of rules are stored. A recent work [4] offloads the state of network functions to the server memory using RDMA. Unlike this system, Orca has small header sizes and its agents maintain small state instead of large network function state. Moreover, Orca simplifies how state is fetched, managed, and replicated.

3 Orca: Server-assisted Multicast

We start this section by specifying the design goals of Orca. Then, we present an overview of Orca describing its main components and how they work together. This is followed by the details of each component. In the **Appendix**, we describe various overheads, extensions, and limitations of Orca.

3.1 Design Goals

The objective of this paper is to design a multicast architecture for datacenter networks that achieves the following goals:

- ***Reduce State at Switches***. Maintaining large state at network switches not only consumes their scarce memory resources, but it also increases the number and frequency of exchanged update messages to handle network failures and session dynamics. This forces switches to process many control messages while forwarding data packets, which may slow down the data plane [66].
- ***Minimize Communication Overhead***. We aim at minimizing the header size per packet to reduce the communication (or bandwidth) overhead, which is critical to decreasing the total transmission time. We note that some of the existing multicast systems, e.g., [5], attach labels that can be as large as the packet payload.
- ***Support Large-scale Multicast Sessions***. As datacenter applications become complex, the numbers of multicast sessions and receivers per session are expected to grow at high rates [70]. Existing systems, e.g., [32], do not efficiently scale to support the growing demands and high dynamics of recent datacenter applications.
- ***Avoid Redundant Traffic***. Switches should forward packets *only* on links belonging to the multicast tree. This is because redundant traffic wastes network resources and overloads switches. Many of the existing multicast systems, e.g., [5, 35], cannot avoid sending redundant traffic without imposing a substantial amount of communication overheads by using large label sizes and/or increasing the state size maintained at switches.

Simultaneously realizing these goals is challenging as they are inter-dependent and pose conflicting trade-offs. For example, although attaching a large label to packets reduces switch state, it significantly increases the communication overhead and packet processing at switches. Our approach to concurrently achieve these design goals is to attach a *small* and *fixed-size* label to packets of every multicast session. This substantially minimizes the communication overhead and reduces packet processing on switches. In addition, we carefully calculate and process labels to eliminate redundant traffic. Furthermore, to reduce state at switches, we make servers assist in forwarding the packets. As a result, switches will be able to support large-scale multicast sessions.

3.2 Overview

Orca is designed for multi-rooted Clos topologies that are widely deployed in datacenter networks. We use the leaf-spine topology throughout the paper, but the same principles apply for other tree-based topologies. In the leaf-spine topology, the top layer consists of core switches that connect different leaf-spine planes. Spine switches connect leaf switches to other leaf switches and to core switches. Every leaf switch connects a rack of servers to the datacenter network. Each server runs a hypervisor switch and hosts multiple virtual machines (VMs).

In traditional IP multicast, network switches need to maintain state about each multicast session, which imposes significant overheads on the switches. In contrast, the proposed approach *carefully offloads* most of the work needed to manage multicast sessions to end hosts in the datacenter, which enables efficient and scalable multicast–a long standing problem. In addition, unlike IP multicast, Orca uses labels to direct the forwarding of multicast packets through the network. Each label consists of different components, each of which encodes a specific datacenter layer (i.e., leaf, spine, or core). However, as the size of a multicast tree grows, simple stacking of label components would lead to large, *variable-size*, labels and thus significant communication and processing overheads.

The proposed architecture is based on three key insights that enable us to design *small* and *fixed-size* labels and achieve scalability. First, a large portion of the label overhead comes from encoding the tree downstream links belonging to leaf switches, and that this overhead increases for multicast trees with large numbers of receivers. Second, labels belonging to leaf downstream links are not needed until the packet reaches a leaf switch. Third, servers in datacenters already host hypervisor switches to process various packet types. Based on these insights, we logically divide the data plane at the leaf layer: between each leaf switch and the servers connected to it. Then, we offload handling of the leaf downstream labels to some of the servers. To process the labels, these servers run an *Orca agent*, which can run on SmartNICs or CPU cores by integrating it with an existing hypervisor switch or running it as a standalone process.

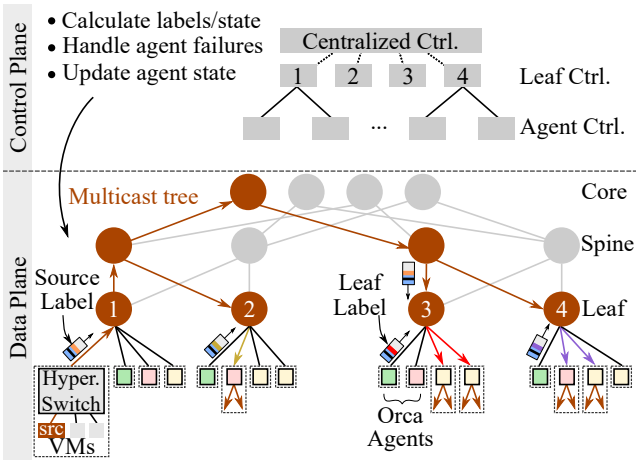


Figure 1: The proposed multicast architecture.

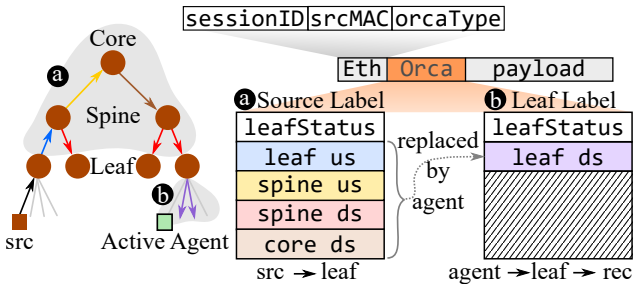


Figure 2: Structure of the Orca header. The color of each label component matches the corresponding link in the network.

As illustrated in Figure 1, Orca is composed of two components: (i) Hierarchical Control Plane and (ii) Server-assisted Data Plane. The Control Plane is composed of three controllers: Centralized, Leaf, and Agent. The Centralized controller creates labels to represent multicast trees and decides the state that needs to be maintained at network switches; details are presented in §3.3. A leaf controller is deployed on each leaf switch, while agent controllers are deployed on VMs within racks. All three components of the Hierarchical Control Plane collaborate to handle network and agent failures as well as to manage the dynamic nature of multicast sessions, as described in §3.4. The Data Plane, presented in §3.5, instructs switches and Orca agents how to process packets.

At a high-level, a multicast session is created and managed as follows, refer to Figure 1. The tree spanning the source and receivers has one path from the source VM to any core switch, then it reaches the receivers by branching to spine and leaf switches. The tree is then given to the centralized controller, which creates a fixed-size label (referred to as *source label*) to represent a part of the tree. It is important to notice that although the multicast tree can be large and spans many parts of the datacenter network, Orca optimizes the source label and keeps its size small and constant, as described in §3.3. The source label is sent to the source of the session, which attaches it to each packet. The packet is then sent upstream to spine and core switches, which forward it based on various components of the source label in that packet. Then, the packet is sent downstream from the spine and core switches, using other components of the source label, to the leaf switches that have receivers of the session in their racks. Each leaf switch sends the packet to an active Orca agent within its rack. The agent replaces the source label with another label (called *leaf label*) and sends it back to the leaf switch. The leaf label contains the information needed by the leaf switch to forward the packet to the end receivers within the rack.

3.3 Calculating Labels

Labels play a critical role in the proposed multicast architecture, and they need to be carefully designed to ensure proper functioning of the multicast system as well as minimize the communication and processing overheads.

The centralized controller computes a fixed-size source label that consists of *four components* and a single `leafStatus` bit. Figure 2 illustrates the header format of an Orca data packet and the label structure. The four label components encode tree links belonging to leaf upstream (us), spine upstream, spine downstream (ds), and core downstream links. When a data packet reaches an active agent, the source label is replaced with the corresponding leaf label to forward packets to the multicast receivers.

The centralized controller calls the CALCULATELABELS algorithm (pseudo code is given in Algorithm 1) to calculate a source label to be attached to packets of the multicast session, a set of leaf labels to be maintained at the agents and state to be maintained at spine switches (if needed). No session state is needed at core or leaf switches. The algorithm takes as input the multicast tree $\mathbb{T}$. It groups tree links of each network layer and encodes their IDs independently in a fixed-size label component.

The algorithm first creates a bitmap of size (in bits):

$$1 + \max(L_d, \lceil \log(L_u) \rceil + P_d + F + \lceil \log(P_u) \rceil + C_d),$$

where L_d, L_u, P_d, P_u, and C_d are the maximum numbers of downstream and upstream ports per leaf switch, downstream and upstream ports per spine switch, and downstream ports per core switch. F is the size of a filter encoding the spine downstream links. This bitmap accommodates the leaf labels that will be inserted by agents. A typical datacenter switch has 48 ports [5]. Thus, the size of Orca label is 19 bytes in most practical cases.

The first bit in an Orca source label is the `leafStatus` bit, which indicates whether an agent has replaced a source label with a leaf label. The remaining bits are used to encode links of the multicast tree based on four cases as follows.

Case 1: Leaf and Spine Upstream. For the leaf and spine upstream links, the CALCULATELABELS algorithm maps the two link IDs to outgoing ports in the leaf and spine

Algorithm 1 The CALCULATELABELS algorithm.

Input: $\mathbb{T}$: multicast tree
Output: L: computed source label sent to the source VM
Output: $\mathbb{S}$: state sent to a subset of the spine switches
Output: $\mathbb{F}$: set of leaf labels, each is sent to an agent

```
1:  function CALCULATELABELS(𝕋)
2:      ⟨L, 𝕊⟩ = CALCULATESOURCELABEL(𝕋)
3:      𝔽 = CALCULATELEAFLABELS(𝕋)
4:      return ⟨L, 𝕊, 𝔽⟩
5:  function CALCULATESOURCELABEL(𝕋)
6:      size=1 + max(L_d, ⌈log(L_u)⌉ + P_d + F + ⌈log(P_u)⌉ + C_d)
7:      L = BitString(size)
8:      L.append(0) // Set leafStatus to 0 (source label)
9:      Case 1: Leaf and Spine Upstream.
10:     L.append(𝕋.leaf_us_link().port_num)
11:     L.append(𝕋.spine_us_link().port_num)
12:     Case 2: Spine Downstream.
13:     // Common downstream ports across spine switches
14:     ℂ = FINDCOMMONPORTS(𝕋.spine_switches())
15:     L.append(MAPTOBITSTRING(ℂ, P_d))
16:     // Call Algorithm 2
17:     ⟨D, 𝕊⟩ = ENCSPINEDSLINKS(𝕋, ℂ, F)
18:     L.append(D)
19:     Case 3: Core Downstream.
20:     core_links = 𝕋.core_ds_links()
21:     L.append(MAPTOBITSTRING(core_links, C_d))
22:     return ⟨L, 𝕊⟩
23: function CALCULATELEAFLABELS(𝕋)
24:     𝔽 = {}
25:     Case 4: Leaf Downstream.
26:     for (leaf ∈ 𝕋.leaf_switches()) do
27:         // Each bit set to 1 represents a session receiver
28:         lbl = MAPTOBITSTRING(leaf.ds_links(), L_d))
29:         𝔽 = 𝔽 ∪ lbl
30:     return 𝔽
```

switches and encodes these port numbers as two labels of sizes $\lceil \log(L_u) \rceil$ and $\lceil \log(P_u) \rceil$ bits, respectively.

Case 2: Spine Downstream. Since the multicast tree may include more than one spine switch, reserving a bit per spine downstream link significantly increases the label size. Instead, we trade off large label sizes, which impose overhead on every single multicast packet, with a small state maintained at a subset of the spine switches. Specifically, we encode the spine downstream links using two label components with a total size of $P_d + F$ bits. The first label component encodes the common downstream ports across all spine switches belonging to the multicast tree using P_d bits. For example, if a tree has three spine switches and the first two outgoing ports belong to the tree for each of the three spine switches, then the calculated label is `1100...0`. We refer to this set of common ports as $\mathbb{C}$.

The second label component uses a probabilistic set membership data structure (a.k.a *filter*) to encode the remaining spine downstream links in a label D of size F bits. Since these filters trade off membership accuracy for space efficiency, they may result in false positives, which occur when some spine downstream links that do not belong to the multicast tree are incorrectly included in the computed filter. False positives result in redundant traffic. To address this issue, we calculate a state alongside the label. This state can have zero or more entries, and each entry takes the form $\langle sID, linkID \rangle$, where sID is the ID of the spine switch that should maintain this state and $linkID$ is the ID of the downstream link identified as a false positive during the encoding. The filter supports two functions: (i) $D = encode(l)$ to encode an input item l (link ID in our case) into a bit string D of size F bits using a hash function, and (ii) $check(l, D)$ to check whether a given item l belongs to D using the same hash function. Our link encoding algorithm can use any filter, e.g., Bloom [54] and Cuckoo [27] filters, that can support: (1) adding an item to an existing filter, (2) testing whether an item exists (potentially with false positives), and (3) avoiding false negatives. A false negative happens when a link in the multicast tree is not represented in the filter.

The CALCULATELABELS algorithm calls the ENCSPINEDSLINKS function, the pseudo code is shown in Algorithm 2 in the Appendix. This function encodes spine downstream links of the multicast tree into a label D and calculates the state $\mathbb{S}$ to be maintained by spine switches. To calculate $\mathbb{S}$, we need to identify false positive links belonging to spine switches. We refer to the subset of the spine downstream links that *may* be false positives as *candidates*. There are two conditions for a spine downstream link to be a false positive candidate. First, it has to be attached to a spine switch that belongs to the multicast tree, as packets of that session do not reach other spine switches. Second, it should not belong to the spine downstream links of the multicast tree. Otherwise, it is not a false positive.

The ENCSPINEDSLINKS function has three steps. First, it encodes every link l in the set of spine downstream links using the *encode* function. Then, it computes the false positive candidates based on the two conditions mentioned earlier. Finally, it calculates the state that needs to be maintained at spine switches by checking all false positive candidates stored in *cands* and adding only the links that collide with the spine downstream links encoded in D and not belonging to $\mathbb{C}$.

Case 3: Core Downstream. The CALCULATELABELS algorithm maps IDs of core downstream tree links to a bitmap of size C_d bits, where C_d is the maximum number of downstream ports in the core layer. The label bits identify the outgoing ports at the core switch belonging to the multicast tree. Thus, a bit at location i in the label is set to 1 if the core switch should duplicate packets on the i^{th} port.

Case 4: Leaf Downstream. For every leaf switch belonging to the multicast tree, the CALCULATELABELS algorithm

calculates a leaf label that encodes all link IDs to reach the receivers of the session within the rack managed by that leaf switch. Each leaf label simply maps the link IDs into a bitmap of size L_d bits.

3.4 Handling Session Dynamics and Failures

Orca employs simple, but effective, mechanisms to manage the dynamic nature of multicast sessions, and to mitigate network and agent failures. We only assume the continuous availability of the top-level, centralized controller of Orca, which can be achieved through mechanisms usually used for such datacenter functions, e.g., [42].

Session Dynamics. Multicast receivers can join and leave any time during the sessions by calling corresponding APIs that communicate with the centralized controller (§4).

When a joining/leaving event is received, the centralized controller runs a simple method (Case 4 in §3.3) to update leaf labels at the agents. The controller then sends the updated leaf labels to the corresponding leaf controllers. The message also includes a unique sequence number. Each leaf controller relays the new leaf label to all active and standby agents within its rack. An agent updates its memory with the new label if the received sequence number is larger than the largest sequence number it has processed so far. Agents then send confirmation messages to upstream controllers indicating that the new changes were processed successfully.

Orca Agent Failures. In each rack, we maintain N Orca agents active and M as standby, where N, M are configurable parameters. All agents within a rack maintain the same leaf label per multicast session. The leaf switch in the same rack distributes the labeling workload among the N active agents, in a round robin manner. This adds more reliability and reduces the labeling load on individual agents.

All agents, active and standby, send heartbeat packets to the leaf controller at a fixed rate. If the leaf controller does not receive any heartbeats from an agent within a timeout period T, the agent is assumed failed. T is in the same order of the RTT within a single rack, which is often a few milliseconds [19]. If the failed agent was active, the leaf controller replaces it by one of the standby agents, otherwise the controller just removes the failed agent from the standby set.

We note that Orca agents deployed in a rack operate independently of agents deployed in other racks. Thus, our approach *localizes* failures within each rack, which reduces the control overhead and increases the control plane responsiveness. In other words, a leaf controller handles *only* the failures of its downstream agents. In addition, heartbeats provide responsiveness and simplicity, which is sufficient in our system as all agents maintain the same state. The state is updated across all agents when there is a change in the multicast tree, which is detected by the centralized controller.

Network Failures. The centralized controller detects network (link and switch) failures using existing systems such as [12]. Once a failure is detected, the controller re-calculates new source and leaf labels for the impacted sessions (§3.3). It also computes a new state at switches (if needed). To mitigate losses during network or agent failures, applications can use reliable transport protocols, e.g., [9].

3.5 Server-assisted Data Plane Forwarding

The data plane in Orca consists of leaf, spine and core switches, as well as agents deployed at servers. The data plane components process received packets as described below.

Leaf Switch. For a packet received on a downstream port, the leaf switch data plane processes that packet based on the `leafStatus` bit. If this bit is zero, i.e., a packet from the source, the data plane reads the first $\log(\lceil L_u \rceil)$ bits after the `leafStatus` bit as a leaf upstream label component, and forwards the packet based on the upstream port number encoded in that component. If the `leafStatus` is set to 1, this means the active agent has inserted a leaf label into the packet header. Thus, the data plane uses the leaf label component of size L_d bits to forward/duplicate the packet to corresponding servers. Specifically, a bit set at location i instructs the data plane to duplicate the packet on its i^{th} port.

If a packet is received on an upstream port, the data plane forwards the packet on a port connected to one of the active agents, which is set and updated by the leaf controller.

Spine Switch. For a packet received on a downstream port, the data plane processes both the upstream and downstream label components. First, the packet is forwarded to a core switch by reading the spine upstream label, which encodes the outgoing port number. Second, since the packet may be forwarded/duplicated on the spine downstream links, the data plane runs the PROCSPINEDSLABEL algorithm to process the two spine downstream labels (pseudo code is shown in Algorithm 3 in the Appendix). This algorithm is executed for packets received on upstream ports as well. The algorithm first identifies the common links $\mathbb{C}$ by reading the first label. If a link is set to one in the label, the switch duplicates the packet on that link. Then, the algorithm uses the second label D and state $State$ maintained by the spine switch to decide which of the other downstream links belong to the tree.

For each link $l \notin \mathbb{C}$, the algorithm decides to not forward the packet on l if it is not encoded in D. This is because filters in Orca do not produce false negatives. When $l.id$ exists in the label, the algorithm needs to check the maintained state $State$ as $l.id$ may be a false positive. Recall that the state contains the false positive links computed by the control plane. The algorithm duplicates the packet only if $l.id$ does not exists in $State[sID]$.

Core Switch. The data plane reads the core downstream label component (of size C_d bits) to forward/duplicate the packet to downstream spine switches. Similar to the leaf downstream label, this label encodes which downstream ports the incoming packet should be forwarded on.

Orca Agent. For incoming packets from a leaf switch, the agent data plane checks the `leafStatus` bit. If it is set to 0 (i.e., it has no leaf label), the agent reads the corresponding leaf label from the leaf label map and inserts it into the packet header and sets the `leafStatus` bit to 1. If the `leafStatus` bit is 1, this packet is destined to receiver VMs.

Overheads and Limitations of Orca. We describe Orca overheads and limitations in the Appendix. In summary, Orca agents require small processing resources at servers as the computation performed on packets is simple. In addition, Orca adds a small latency to packets at the leaf layer only. Furthermore, deploying Orca in graph-based datacenter networks requires changes in some of its components.

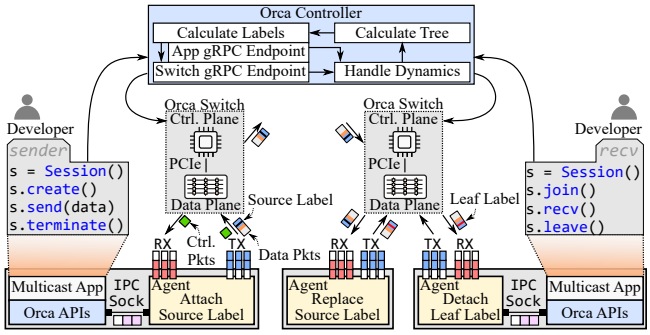


Figure 3: Implementation of Orca components.

4 Implementation and Orca APIs

We briefly describe the implementation of Orca components which are illustrated in Figure 3.

Orca APIs for Multicast Applications. We implemented two sets of interfaces for multicast applications. The first one is between the agent and applications to provide `send` and `recv` functionalities seamlessly to the application. These APIs use Unix domain sockets to communicate with the agent. When using `send` at the source, the agent gets data from the sockets, attaches Orca label and transmits the packets to the leaf switch. Receivers use `recv` to instruct the agent to relay available data to the application. The second set of APIs is between applications and the centralized controller to `create`, `join` and `leave` multicast sessions. We implemented the communication and data encoding/decoding using gRPC [64] and Protocol Buffers.

Orca Agents. We implemented the agent using BESS [23,61] in about 640 lines of C++, where packet processing is done completely in the user space using DPDK [63]. The agent leverages Receive Side Scaling (RSS) to receive packets on different RX queues, each is assigned to a single core.

Orca Hierarchical Controller. The centralized and leaf controllers are implemented in about 3K lines of Golang. The current implementation of the leaf controller communicates with the data plane through raw or Unix domain sockets, but it can easily support other interfaces. For instance, in our testbed, the leaf controller process is deployed to the workstation hosting a NetFPGA, and it uses PCIe to exchange control packets with the NetFPGA, which is done through the RIFFA framework [18]. Communications between the centralized and leaf controllers are done using gRPC [64].

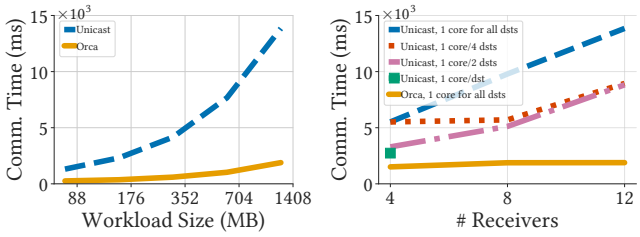
Switch Data Plane. Since Orca's data plane processing is simple, it can easily be implemented in different programmable switches. We implemented the data plane of Orca in NetFPGA SUME [29] and tested it on multiple of them. We used the open source project in [56], and implemented a Verilog module to decide the outgoing ports. We measured the number of clock cycles and resource usage of our implementation using Xilinx tools. Our implementation of core and leaf switches maps the corresponding bitmaps to outgoing ports, which is done in one clock cycle. We implemented the spine switch algorithm in three clock cycles to identify common ports and check the Bloom filter using a bitwise-AND between the label and hashed link IDs stored at the switch, read state from memory, and decide the outgoing ports. In terms of resource usage, our algorithm utilizes a tiny percentage of the available hardware resources. It uses 0.12% and 0.16% of the available lookup tables (LUTs) and registers, respectively.

5 Evaluation of Orca in Testbed

We evaluate Orca in a testbed to demonstrate its potential benefits to applications and asses the performance of its data plane and control plane components. The testbed has three NetFPGA SUME switches [29] representing a spine and two leaf switches, each of which has four 10GbE ports. The testbed also has five workstations to act as Orca agents and multicast senders and receivers. We configure our testbed to only have one active agent per rack to stress our labeling algorithm at servers. Each workstation is equipped with a dual-port Intel 82599ES 10GbE NIC. Each leaf switch is connected to two workstations, and the spine switch is connected to one workstation. We generate traffic at line rate from a multicast source and transmit it to leaf switches through the spine switch.

5.1 Benefits of Orca

We implemented a sample multicast application that has the same behavior of the iterative machine learning algorithms implemented in Spark [40]. In these algorithms, the data to be processed is often written to files, and a server iteratively sends them to all receivers for processing. In our application, a server reads a file and transmits it in chunks. In every iteration, after a file is sent, the server awaits acknowledgment of file reception from the receivers. The next round starts only after receiving all acknowledgments. This emulates the aggregation phase in distributed data processing frameworks [44], which indicates all workers have updated their model parameters. In our implementation, we set the payload size to the maximum



(a) Communication time (b) Required CPU resources

Figure 4: Benefits of Orca.

UDP message length. We run each instance of the client application inside a separate Docker container on the receiver machines.

Performance Metrics. We compare Orca versus the current unicast approach used in systems such as Apache Spark [40]. In particular, we demonstrate the potential benefits of Orca in terms of the communication time, impact of available processing capacity at the sender on the communication time, and total transmitted traffic.

The running time of datacenter applications consists of communication and computation times. The communication time of an application is the total time spent on sending data and receiving corresponding acknowledgments without including the computation times. We measure the communication time of Orca versus unicast, as this is the main aspect being optimized by Orca and it does not control or modify the computations. In addition, we aim at showing that Orca can present the same packet to the application layer much faster compared to the current unicast approaches.

Depending on the application and its total computation time, Orca can reduce the running times of a variety of applications. For example, the authors in [36] reported that the communication time of data-intensive tasks using unicast can be larger than the computation time, especially as the number of workers increases. Thus, optimizing data transfer is critical for these applications.

Workloads. The number and size of the transmitted files are similar to the ones used in the distributed latent Dirichlet allocation (LDA) algorithm [9, 30]. LDA identifies topics in the input documents and maps each document to a set of topics. The vocabulary training set is the data transmitted to the worker nodes. To calculate the workload size, we run the algorithm on a synthetic dataset containing 16,923 documents and 100 topics using the tool in [71]. To evaluate Orca using realistic workloads, we create five different workloads with sizes of 88MB, 176MB, 352MB, 704MB, and 1.4GB.

Results. We conduct experiments using concurrent 4, 8 and 12 receivers and the five different workloads mentioned above.

Figure 4a shows the communication time for Orca and unicast for different workloads when the number of receivers is 12. The sender in the multicast session uses one CPU core to transmit the traffic. These results show that Orca can significantly reduce the communication time for all considered workloads. In addition, the figure shows that, unlike the case for Orca, the communication time for unicast grows in a super-linear manner with the workload size. This is because the unicast sender needs more time to transmit packets to each of the concurrent 12 receivers, whereas Orca transmits only a single packet for all receivers. Packet transmission at high rates also requires processing cycles.

To analyze the impact of the available processing capacity at the sender on the communication time, we allocate a varying number of CPU cores to transmit the traffic of the multicast session in the case of unicast. For Orca, only one CPU core is used. In Figure 4b, we plot the communication time for the largest workload (1.4GB) for Orca and unicast, as we vary the number of available CPU cores. The figure shows that Orca has a fairly stable communication time as the number of receivers increases, despite using only one CPU core to transmit all packets of the session. In contrast, unicast needs more CPU cores to send the traffic to different receivers to reduce the communication time. In our testbed, allocating a single core per receiver for unicast could not sustain the high packet rate at the sender for 8 and 12 receivers.

Next, we measure the total transmitted traffic from the sender for the largest workload as well as the label overhead of Orca. When using Orca, the total outgoing traffic is only 1.51 GB, compared to 18.01 GB when using unicast. This means the sender in the unicast model would need to transmit 12X more traffic, which not only consumes more bandwidth, but also requires more processing and memory resources to transmit much more packets. The total label overhead of Orca is 7.69 MB which represents only 0.51% of the transmitted multicast traffic.

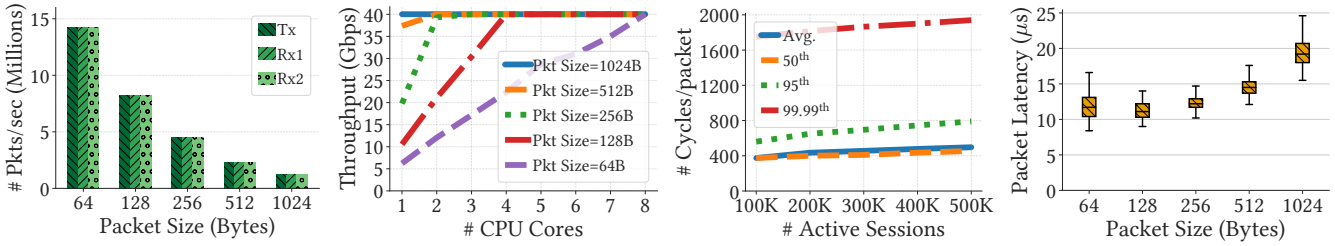
Although current multicast approaches may yield similar benefits to applications, they cannot scale well to support a large number of multicast sessions. Therefore, we compare the scalability of Orca versus the state-of-art multicast system using large-scale simulations in §6.

***Summary**: For a sample application with 4–12 receivers, Orca achieves substantial savings in communication time, required processing resources at the sender, and bandwidth, compared to the current unicast approach.*

5.2 Data Plane Performance

Throughput of Spine Switches. We report the throughput of the spine switch; we omit the results of leaf and core switches as they run simple forwarding algorithms.

We transmit labelled packets of many concurrent multicast sessions at the maximum link speed (i.e., 10 Gbps) from the source to the spine switch. The labels instruct the spine switch to duplicate packets to two leaf switches. We run this experiment five times for every packet size and compute the average across them. We compare the incoming packet rate against the outgoing packet rates observed at the two leaf



(a) Spine switch throughput (b) Agent throughput (c) CPU usage per packet (d) Packet latency

Figure 5: Data plane performance of Orca.

switches in Figure 5a. Our results show that the packet rates are the same (i.e., no packet losses). We also measure the achieved throughput at the two interfaces, and confirm that the spine switch can sustain the 10 Gbps for all packet sizes.

Agent Scalability. We stress and evaluate the scalability of the agent data plane. In this setup, we deploy two NICs (i.e., 4x10GbE ports) at the agent workstation and direct labeled traffic at rate of 40Gbps from the other two workstations. The labels have the `leafStatus` set to zero, which indicates that the agent needs to label them using a corresponding leaf label. We measure the throughput of the packets after being processed by the agent.

Figure 5b shows the throughput versus the number of allocated cores for the data plane, which shows that the agent scales well to support high rates. We measure the smallest packet size at which the agent can sustain the 40Gbps traffic using a single core. Our results show that the agent can sustain this rate using 1 core for packets of size 560 bytes or larger. For enterprise datacenters, the average packet size is reported to be 850 bytes [41]. Furthermore, data-intensive jobs like Hadoop workloads often use 1500-byte packets [20]. That is, Orca agents require only a few cores per rack to support many multicast sessions at high rates. Major datacenters deploy 24–48 servers per rack [14, 65], and each typically has more than 16 cores. That is, even for applications that require small 64-byte packets and send at an aggregate rate of 40Gbps, an Orca agent would need up to 1–2% of the available CPU resources in a rack when SmartNICs are unavailable.

In addition, recall from §5.1 that Orca requires only one CPU core at the sender side regardless of the number of receivers, whereas the current unicast approach needs a proportional number of CPU cores to sustain the transmission rate especially as the number of receivers increases. Thus, the processing capacity needed to run Orca agents will likely be offset by the savings in the processing capacity needed to transmit the traffic in the unicast approach.

Agent CPU Usage. Recall that an active agent needs to look up a leaf label from its memory using the session ID. We measure the total number of CPU cycles needed by the agent to process packets (including labeling and memory lookup). We stress the agent by allocating leaf labels for 1M sessions at the agent. In this experiment, the sender randomly transmits traffic belonging to a subset of the total sessions, which we refer to as active sessions. We use large numbers of active sessions to stress the agent.

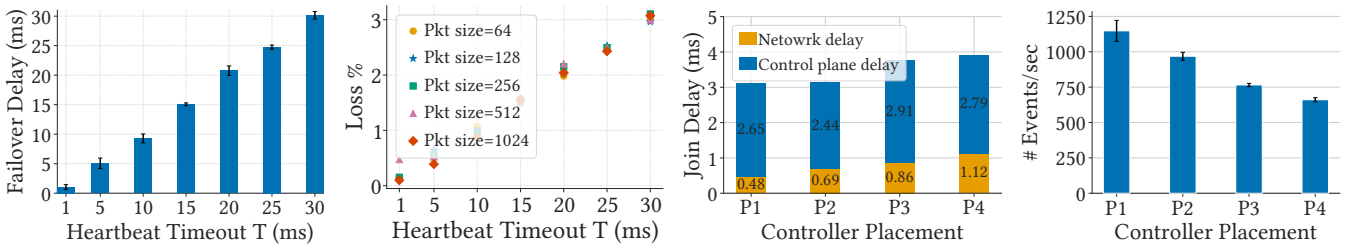
Figure 5c shows different statistics of the used number of CPU cycles per packet (measured by `rdtsc`) when the packet size is 1024 bytes. The results show the efficiency of the agent even without any code optimizations. For example, the agent running on a 3.8GHz CPU needs an average of 99 ns per packet when the number of active sessions is 100K per rack. We note that the number of CPU cycles is constant for different packet sizes, since the agent processes fixed-size labels. To put these numbers in context, existing, optimized, software switches such as OVS [24] and PISCES [15] use 409 and 426 cycles/packet, on average, to handle IP packets, respectively. The average for Orca is 375 cycles/packet when handling 100K active sessions.

Packet Latency and Jitter. We measure the packet latency and jitter of Orca at the leaf layer, which is defined as the total duration from when a packet is sent to the leaf switch to the time it is received by a multicast receiver connected to that switch. We emulate a dynamic traffic scenario, where the sender starts transmitting traffic at 1Gbps and increases the sending rate with 1Gbps steps every 20 seconds until it saturates the link, and holds this transmission rate for another 20 seconds.

Figure 5d shows the packet latency for different packet sizes. For 64-byte packets, the median latency is 11.3 μs. The packet latency slightly increases for large packet sizes because of the increased transmission time. Notice that in latency-sensitive applications, where latency for individual packets are important, smaller packets are more prevalent [57] where Orca has short packet latency.

We next measure the packet inter-arrival jitter, which is calculated as the difference between the current packet delay and previous packet delay. Our results show that Orca imposes negligible variance in packet latency. The average and maximum inter-arrival jitter values for 1024-byte packets are 1.135 μs and 3.3 μs, respectively.

***Summary**: The Orca data plane is scalable and can sustain high throughputs even with small packet sizes. Orca agents*



(a) Failover delay at a failure (b) Loss rate during a failure (c) Receiver join delay (d) Confirmation msgs. throughput

Figure 6: Control plane performance of Orca.

can process large numbers of concurrent sessions, use low CPU resources, and only add a small latency to packets.

5.3 Control Plane Performance

Responsiveness to Agent Failures. Orca localizes agent failures within the rack, thus we analyze failures for a single rack. We measure the performance while an active agent is handling traffic at 10Gbps. We manually crash the active agent and measure the following metrics at a receiver: failover delay, throughput, and data loss rate. We report the results for the worst-case scenario, where the rack has only one active agent. Failover delay is the time when the receiver does not receive traffic due to active agent failure and when it receives traffic again. Figure 6a shows the average failover delay for all packet sizes when we control the heartbeat timeout. The results confirm the fast response of the control plane in choosing a new active agent when the original agent fails. For instance, receivers can resume receiving traffic within 1.04 ms after an active agent fails when T is 1 ms.

We next measure the observed throughput at the receiver during a failure, where we crash the agent after two seconds. For 1024-byte packets and heartbeat timeout of 1ms, we observe a throughput drop by up to 0.012% only. In addition, for all packet sizes, the total throughput drop is less than 0.013% during a failure. Our results confirm that Orca quickly restores the transmission to full capacity after a failure. Finally, we plot the loss rate caused by an agent failure in Figure 6b. When the heartbeat timeout is 1 ms, Orca incurs a loss rate of 0.18%, on average, across all packet sizes. We note that such losses can be easily mitigated by using reliable multicast [9].

Receiver Joining Delay. We assess the performance of the proposed method for updating leaf labels when a new receiver joins. Recall that when a session changes, Orca sends new leaf labels to the corresponding agents. This impacts how quickly a joining receiver would receive traffic. In addition, network delays between the control plane components in datacenters might vary depending on the placement of the controllers and receivers. We emulate different controller placement setups in our testbed by adding synthetic delays at the network interface queues of the workstations (using `tc`) to stress our system. We consider four different placement setups (P1–P4) starting with no synthetic delay in P1 with mean RTT of 479 μs and adding 200 μs of delay every step till we reach a mean RTT of 1,120 μs (maximum 1,326 μs) in P4 setup. These RTT values follow what is reported in [19] where the 99th percentile RTT between two hosts is 1.34 ms. We note that even the lowest RTT in our setup (i.e., P1) is larger than the median RTT inside a rack in datacenter networks which is 268 μs [19].

We measure the receiver join delay, which is the time duration from when a receiver sends a join request for a session and the time the first packet of that session is received by the receiver. For each placement setup, the experiment is repeated for 30 join events. Figure 6c shows the average join delay as well as the contribution of network delays. We report that even in the worst-case scenario (P4), the average join delay is less than 4 ms. In total, the median and 99th-percentile delays are 3.12 ms and 6.53 ms, respectively. To put these numbers into perspective, note that inserting a new rule into an OpenFlow switch takes 1–3 ms, and rule modification delays vary from 2–18 ms [22].

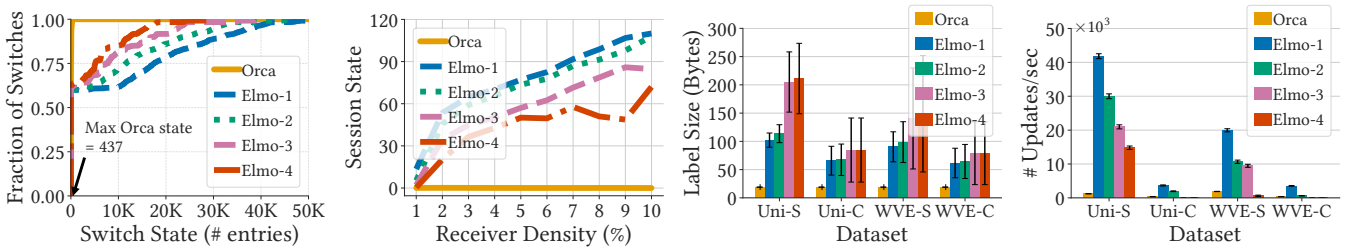
We next measure the throughput of processed confirmation messages at the centralized controller in Figure 6d, which represents the end-to-end performance. In the P1 setup, Orca handles an average of 1,147 msgs/sec (SD is 74). As increasing latency affects gRPC, the average throughput of the largest delay scenario is 662 msgs/sec (SD is 14).

***Summary**: Orca recovers quickly from failures and it supports dynamic multicast sessions. Furthermore, the failure detection mechanism in Orca is localized to individual racks.*

6 Orca versus State-of-Art

We analyze the performance and scalablity of Orca and compare it against the state-of-art system, Elmo [5], using large-scale simulations. We use the open-source code of Elmo.

Elmo employs three stages to encode a multicast tree. Elmo encodes switches of a tree as a union of multiple bitmaps representing outgoing ports. When the label size reaches a pre-configured value, Elmo installs forwarding state entries at switches without exceeding their capacities. Otherwise, Elmo calculates a default entry that may result in redundant traffic.



(a) Switch state CDF (b) Session state (c) Comm. Overhead (d) Message update rate

Figure 7: Performance of Orca versus Elmo.

6.1 Simulation Setup

Topology and VM Placement. We simulate a multi-rooted Clos topology consisting of 48 pods, each has 576 48-port leaf and spine switches. This results in a large datacenter network of 27,648 hosts. We use a setup similar to [5, 32]: There are 3,000 tenants, each has a number of VMs ranging from 10 to 5,000. The maximum number of VMs per server is 20. We use two VM placement strategies. The first one is a Clustered placement (denoted by C), which places at most 12 tenant VMs per rack. The second is a Scattered strategy (denoted by S), and it places at most one tenant VM per rack.

Multicast Sessions and Datasets. Multicast receivers per session are randomly chosen from all tenant VMs. The size of these sessions follows two different distributions similar to [5, 32]. The first distribution follows a workload from the IBM WebSphere Virtual Enterprise (WVE) [32], and the second one is a uniform distribution (Uni). The minimum and maximum session sizes for both distributions are 5 and 5,000 receivers, respectively.

We generate four datasets representing various workload characteristics and VM placement strategies. We denote a dataset using its session size distribution and VM placement strategy, e.g., a dataset with uniform session sizes and scattered strategy is referred to as Uni-S. We simulate 1M multicast sessions per dataset, where each tenant has sessions proportional to the number of its VMs.

Orca and Elmo Parameters. We set the filter size in Orca to 69 bits to compute byte-aligned label. For Elmo, we control two parameters to analyze different aspects of it, and set them according to [5]. The first one is the number of rules encoded in Elmo label, which is set to be either 10 or 30. The second parameter is the redundancy limit that controls the amount of redundant traffic caused by sharing a single rule in Elmo label. We set this parameter to be 0 (no redundant traffic) or 12. We refer to the Elmo four configurations as Elmo-1 (10, 0), Elmo-2 (10, 12), Elmo-3 (30, 0) and Elmo-4 (30, 12).

6.2 Data Plane Performance

Switch State. Figure 7a shows the CDF of the switch state for the Uni-S dataset. The results for other datasets are similar. The figure shows that Orca significantly reduces the state size compared to all considered configurations of Elmo. For example, in Orca, 99% of switches need to only maintain up to 253 entries in their memory, and no switch maintains more than 437 entries. In contrast, for Elmo-1, which calculates the smallest label sizes (i.e., 100 bytes on average), 99% of switches need to maintain up to 47.7K entries in their memory, with some switches need to maintain as many as 53.5K entries. Elmo could not reduce the state even when it doubles the label size. For example, in Elmo-4, 99% of switches need to maintain up to 24K entries in their memory (maximum is 30K entries). This is a significant improvement because it indicates that Orca requires much lower switch memory to support the same number of multicast sessions and much fewer control messages to update the switch state.

We next study the impact of session size on the required state to be maintained for that session in Figure 7b. The figure shows that Orca scales well, and it can reduce the session state by up to 55X compared to Elmo. For example, when a session has 2.5K receivers, Orca needs to maintain state at up to two switches only. Elmo-1, however, needs to maintain state at up to 110 switches.

These significant gains are achieved because, unlike Elmo, Orca does not require maintaining state at *any* leaf or core switch. In addition, the proposed spine labels can encode most of the spine downstream links while requiring small state at few spine switches.

***Summary**: Orca reduces state size by up to two orders of magnitude compared to Elmo, and can support a large number of concurrent multicast sessions.*

Communication Overhead. Figure 7c shows that Orca reduces the communication overhead by using a small and fixed-size label of size 19 bytes to forward traffic of 1M sessions. On the other hand, Elmo uses much larger labels. For example, in the Uni-S dataset, the average and maximum label sizes of Elmo-4 are 211 bytes and 368 bytes, respectively; SD is 62 bytes.

Elmo introduces variations in the label size for the *same* configuration across different datasets. This means that changes in VM placement strategy or shifts in traffic patterns introduce unpredictable forwarding performance in Elmo, as

its switches need to process labels with *varying* sizes. For example, changing the VM placement strategy from clustered to scattered in Elmo-4 increases the average label size by 148% because receivers in the scattered strategy span more racks compared to the clustered one. For the same configuration, a shift in traffic pattern from WVE to Uni increases the average label size by 42% as Elmo needs to encode more receivers using the same label size. This is because the WVE distribution is skewed, and thus, fewer sessions have large group sizes compared to the Uni distribution. In contrast, Orca has a *fixed-size* label of 19 bytes because it utilizes the key insights described in §3.2.

***Summary**: Orca reduces the communication overhead by up to 19X compared to Elmo while being robust against VM placement strategies and session sizes.*

Redundant Traffic. We define the redundant traffic per session as the ratio between the number of receivers that receive unwanted traffic to the total number of receivers in the session. *By design, Orca does not introduce any redundant traffic*. Elmo may introduce redundant traffic to reduce state size by controlling the redundancy limit parameter as it shares the same rule among multiple switches. We analyze the traffic redundancy of Elmo-2 and Elmo-4 for the Uni-S dataset. Other Elmo configurations have redundancy limit of 0 similar to Orca but they require maintaining much larger state at switches. Our results show that, for Elmo-2, 25% of the sessions have more than 67% redundant traffic, with a maximum value of 172%. For Elmo-4, with much larger labels, the maximum redundant traffic is 113%. That is, the traffic could erroneously be sent to more destinations not participating in the multicast session than the actual receivers.

***Summary**: Orca does not introduce any redundant traffic, whereas Elmo may impose up to 172% redundant traffic.*

6.3 Control Plane Performance

Session Dynamics. We randomly generate 1,000 receiver joining/leaving events per second with joining probability of 0.5. Every event changes a multicast tree, and thus, the state maintained at switches may need to be refreshed. Refreshing the state requires the control plane to send update messages to the switches. We measure the total number of update messages per second sent by the controller for both Orca and Elmo. We report the results for all datasets in Figure 7d. For example, the Orca controller needs to send an average of 1,889 messages per second (SD is 45) for the WVE-S dataset. On the other hand, the Elmo controller needs to send 19.9K, 10K, 9.5K and 615 messages per second on average for Elmo-1, Elmo-2, Elmo-3 and Elmo-4, respectively. This is because Orca maintains state only at a small number of switches. Elmo-4 does not need to update many switches. It, however, imposes the largest label size among all Elmo configurations with high amount of traffic redundancy.

***Summary**: Orca reduces the rate of update messages by up to 10X compared to Elmo.*

Network Failures. Similar to session changes, a core or spine switch failure triggers Orca and Elmo to update state at switches if needed. For the WVE-S dataset, Orca needs to send 3,900 messages per core switch failure on average, while Elmo-1, Elmo-2, Elmo-3, and Elmo-4 needs to send an average of 56.2K, 31.2K, 27.6K, and 1.7K messages per core switch failure, respectively. For a spine switch failure, Elmo-1, Elmo-2, Elmo-3, and Elmo-4 send 34.7K, 20.6K, 18K, and 1.2K messages per failure, respectively, whereas Orca sends 4,890 messages per failure. Although Elmo-4 requires sending fewer messages per failure, it imposes significant overheads in terms of the label size and traffic redundancy.

***Summary**: Compared to Elmo, Orca reduces the control overhead for handling failures by up to 14X.*

Running Time. Orca calculates labels faster than Elmo. We report the running time of Orca and Elmo in the Appendix.

7 Conclusions and Future Work

We presented Orca, an efficient multicast architecture for datacenter networks. Orca splits the data plane operations between leaf switches and servers. That is, Orca offloads managing multicast sessions from leaf switches to servers. Orca has a scalable control plane that handles session dynamics and network failures. It also has a simple data plane that can sustain high rates and can easily be implemented in programmable switches. The server component in Orca can be implemented on SmartNICs, or on regular CPU cores if SmartNICs are not available. We implemented lightweight APIs to seamlessly integrate multicast into datacenter applications. We also implemented Orca in a testbed that contains programmable switches. We evaluated a sample multicast application in our testbed. Our results show that Orca can substantially reduce the communication time compared to unicast. In addition, we assessed the performance of Orca in terms of its throughput, resource usage, packet latency and the impact of failures. Moreover, we compared Orca versus the state-of-art multicast system, Elmo, using large-scale simulations. Compared to Elmo, Orca reduces the switch state by up to two orders of magnitude and the label size by up to 19X.

This work can be extended in multiple directions. For example, we plan to extend Orca to support various group communication primitives needed by modern datacenter applications.

Acknowledgments

We thank our shepherd, Sujata Banerjee, and the anonymous reviewers for their insightful and helpful comments. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Khaled Diab and Mohamed Hefeeda. Yeti: Stateless and generalized multicast forwarding. In *Proc. of USENIX NSDI'22*, Renton, WA, April 2022.

[2] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proc. of ACM SoCC'21*, pages 138–152, Seattle, WA, November 2021.

[3] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proc. of ACM SIGCOMM'20*, pages 681–693, Virtual Event, August 2020.

[4] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proc. of ACM SIGCOMM'20*, pages 90–106, Virtual Event, August 2020.

[5] Muhammad Shahbaz, Lalith Suresh, Jen Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source-routed multicast for public clouds. In *Proc. of ACM SIGCOMM'19*, pages 458–471, Beijing, China, August 2019.

[6] Toerless Eckert, Gregory Cauchie, and Michael Menth. Traffic Engineering for Bit Index Explicit Replication (BIER-TE). Internet-draft, Internet Engineering Task Force, July 2019. Work in Progress.

[7] Collin Lee and John Ousterhout. Granular Computing. In *Proc. of ACM HotOS'19*, pages 149–154, Bertinoro, Italy, May 2019.

[8] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. February 2019. arXiv: 1902.03383.

[9] Xiaoye Steven Sun, Yiting Xia, Simbarashe Dzinamarira, Xin Sunny Huang, Dingming Wu, and TS Eugene Ng. Republic: Data multicast meets hybrid rack-level interconnections in data center. In *Proc. of IEEE ICNP'18*, pages 77–87, Cambridge, United Kingdom, September 2018.

[10] Xiaoye Steven Sun and TS Eugene Ng. When creek meets river: Exploiting high-bandwidth circuit switch in scheduling multicast data. In *Proc. of IEEE ICNP'17*, pages 1–6, Toronto, Canada, October 2017.

[11] O. Komolafe. Ip multicast in virtualized data centers: Challenges and opportunities. In *Proc. of IFIP/IEEE IM'17*, pages 407–413, Lisbon, Portugal, May 2017.

[12] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive realtime datacenter fault detection and localization. In *Proc. of USENIX NSDI'17*, pages 595–612, Boston, MA, March 2017.

[13] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proc. of ACM CoNEXT'16*, pages 205–219, Irvine, CA, December 2016.

[14] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Proc. of IEEE MICRO'16*, pages 1–13, Taipei, Taiwan, October 2016.

[15] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proc. of ACM SIGCOMM'16*, pages 525–538, Florianopolis, Brazil, August 2016.

[16] Bill Fenner, Mark J. Handley, Hugh Holbrook, Isidor Kouvelas, Rishabh Parekh, Zhaohui (Jeffrey) Zhang, and Lianshu Zheng. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). RFC 7761.

[17] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. of ACM SOSR'16*, pages 1–12, Santa Clara, CA, March 2016.

[18] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 8(4), September 2015.

[19] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. of ACM SIGCOMM'15*, pages 139–152, London, United Kingdom, August 2015.

[20] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proc. of ACM SIGCOMM'15*, pages 123–137, London, United Kingdom, August 2015.

[21] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proc. of ACM SOSR'15*, pages 1–7, Santa Clara, CA, June 2015.

[22] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *Proc. of ACM SOSR'15*, pages 1–6, Santa Clara, CA, June 2015.

[23] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[24] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *Proc. of USENIX NSDI'15*, pages 117–130, Oakland, CA, May 2015.

[25] Yiting Xia, TS Eugene Ng, and Xiaoye Steven Sun. Blast: Accelerating high-performance data analytics applications by optical multicast. In *Proc. of IEEE INFOCOM'15*, pages 1930–1938, Hong Kong, China, April 2015.

[26] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, January 2015.

[27] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proc. of ACM CoNEXT'14*, pages 75–88, Sydney, Australia, December 2014.

[28] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proc. of USENIX OSDI'14*, page 583–598, Broomfield, CO, October 2014.

[29] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, September 2014.

[30] Zhuhua Cai, Zekai J Gao, Shangyu Luo, Luis L Perez, Zografoula Vagena, and Christopher Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proc. of ACM SIGMOD'14*, pages 1371–1382, Snowbird, UT, June 2014.

[31] Dan Li, Mingwei Xu, Ying Liu, Xia Xie, Yong Cui, Jingyi Wang, and Guihai Chen. Reliable multicast in data center networks. *IEEE Transactions on Computers*, 63(8):2011–2024, May 2014.

[32] Xiaozhou Li and Michael J Freedman. Scaling IP multicast on datacenter topologies. In *Proc. of ACM CoNEXT'13*, pages 61–72, Santa Barbara, CA, December 2013.

[33] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proc. of USENIX NSDI'12*, pages 253–266, San Jose, CA, April 2012.

[34] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proc. of USENIX NSDI'12*, pages 225–238, San Jose, CA, April 2012.

[35] Dan Li, Henggang Cui, Yan Hu, Yong Xia, and Xin Wang. Scalable data center multicast using multi-class bloom filter. In *Proc. of IEEE ICNP'11*, pages 266–275, Vancouver, Canada, October 2011.

[36] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proc. of ACM SIGCOMM'11*, page 98–109, Toronto, Canada, August 2011.

[37] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proc. of ACM Workshop on Networking Meets Databases (NetDB'11)*, pages 1–7, Athens, Greece, June 2011.

[38] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of ACM IMC'10*, pages 267–280, Melbourne, Australia, November 2010.

[39] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Proc. of IEEE/IFIP DSN'10*, pages 527–536, Chicago, IL, June 2010.

[40] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proc. of USENIX HotCloud'10*, pages 1–7, Boston, MA, June 2010.

[41] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[42] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proc. of ACM SIGCOMM'09*, page 51–62, Barcelona, Spain, October 2009.

[43] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. Lipsin: Line speed publish/subscribe inter-networking. In *Proc. of ACM SIGCOMM'09*, pages 195–206, Barcelona, Spain, August 2009.

[44] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[45] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[46] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[47] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. of ACM SOSP'03*, pages 282–297, Bolton Landing, NY, October 2003.

[48] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[49] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. *ACM SIGOPS Operating Systems Review*, 37(5):298–313, 2003.

[50] Bradley Cain, Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376.

[51] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM'02*, pages 205–217, Pittsburgh, PA, August 2002.

[52] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, January 2000.

[53] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[54] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[55] Multicast Command Reference for Cisco ASR 9000 Series Routers. https://bit.ly/3AaVGDQ. [Online; accessed February 2022].

[56] NetFPGA SUME Reference Learning Switch Lite. https://bit.ly/2UrUFlx. [Online; accessed February 2022].

[57] 10Gb Ethernet: The Foundation for Low-Latency, Real-Time Financial Services Applications and Other, Latency-Sensitive Applications. https://bit.ly/33xtYST. [Online; accessed February 2022].

[58] Apache ActiveMQ. http://activemq.apache.org. [Online; accessed February 2022].

[59] Apache Hadoop. https://hadoop.apache.org/. [Online; accessed February 2022].

[60] Apache openwhisk. https://openwhisk.apache.org/. [Online; accessed February 2022].

[61] Bess (berkeley extensible software switch). https://github.com/NetSys/bess. [Online; accessed February 2022].

[62] Bringing Multicast to the Cloud. https://bit.ly/3jP7naY. [Online; accessed February 2022].

[63] Data plan development kit (DPDK). https://intel.ly/2GYxLAV. [Online; accessed February 2022].

[64] gRPC - An RPC library and framework. https://github.com/grpc/grpc. [Online; accessed February 2022].

[65] Introducing data center fabric, the next-generation Facebook data center network. https://bit.ly/3bWEDKG. [Online; accessed February 2022].

[66] Multicast group capacity: Extreme comes out on top. https://bit.ly/2H5sQ1n. [Online; accessed February 2022].

[67] Open Config, Streaming Telemetry. https://bit.ly/3kf7EEj. [Online; accessed February 2022].

[68] RabbitMQ. http://www.rabbitmq.com. [Online; accessed February 2022].

[69] Using reliable multicast for data distribution with opendds. https://bit.ly/3bWFefo. [Online; accessed February 2022].

[70] Why the world's largest hadoop installation may soon become the norm. https://tek.io/33gDCsU. [Online; accessed February 2022].

[71] Xiaoye Sun. LDA Data Generator. https://github.com/sunxiaoye0116/data_generator/tree/dev. [Online; accessed February 2022].

Appendix A Supplementary Materials

This appendix includes materials that complement the contents presented in the paper.

A.1 Encoding Spine Downstream Links

The pseudo code of the EncSpineDSLinks algorithm is shown in Algorithm 2. This algorithm encodes spine downstream links of the multicast tree into a label D and calculates the state $\mathbb{S}$ to be maintained by spine switches.

A.2 Processing Spine Downstream Labels

The pseudo code of the ProcSpineDSLabel algorithm is shown in Algorithm 3. The algorithm processes two spine downstream labels: the common links among spine switches in the tree (denoted by $\mathbb{C}$), and the filter that encodes the remaining spine downstream links (denoted by D).

A.3 Overheads of Orca

Multicast offers significant bandwidth savings compared to unicast, and thus, it can scale data-intensive tasks that dominate datacenter networks. The authors of [36] reported that the communication time of data-intensive tasks using unicast can be larger than the computation time, especially as the number of workers increases. Achieving the benefits of multicast has been a long-standing problem. Orca achieves the benefits of multicast at the expense of the small overheads described below.

Server Resources. Orca agents require processing resources at servers. However, the computation performed on packets (mostly replacing labels) is quite simple and the memory needed to store leaf labels is small. Thus, Orca agents can easily be implemented on SmartNICs, which are getting popular in datacenters [3]. In this case, no CPU cores are taken away from the servers. Orca agents can also run on regular CPU cores. In this case, the agents consume only a small fraction of the available computing resources in each rack, as shown in the evaluation section. We note that since Orca is a multicast paradigm, the sender in the session transmits only one copy of each packet regardless of the number of receivers in the session. In contract, in unicast, the sender needs to send a separate copy of each packet to every receiver, which for large-scale applications with many receivers and/or high data rates requires allocating additional CPU cores at the sender to sustain the needed data rate. That is, at the whole system level, the CPU resources used by Orca agents can be offset by the savings of CPU resources at the sender.

Packet Latency. Orca adds latency to packets at the leaf layer only, because the packets need to be sent to Orca agents for relabeling. This latency is in the order of one RTT within the rack, because of the simple processing done on packets by Orca agents. Most throughput-intensive datacenter applications, e.g., MapReduce [44], Hadoop [59], and Spark [40], can easily tolerate this small latency [33].

Algorithm 2 Encode spine downstream links.

Input: $\mathbb{T}$: multicast tree
Input: $\mathbb{C}$: common ports in spine downstream switches
Input: F: filter size in bits
Output: D: computed spine downstream label
Output: $\mathbb{S}$: state sent to a subset of the spine switches

```
1:  function EncSpineDSLinks(𝕋, ℂ, F)
2:      A Calculate a spine downstream label
3:      D = BitString(size=F)
4:      for (l ∈ 𝕋.spine_ds_links()) do
5:          if (l ∉ ℂ) then
6:              D = D ∪ encode(l.id)
7:      B Calculate false positive candidates
8:      cands = { }
9:      for (u ∈ 𝕋.spine_switches()) do
10:         for (l ∈ u.ds_links()) do
11:             if (l ∉ 𝕋.spine_ds_links()) then
12:                 cands = cands ∪ (u, l.dst)
13:     C Calculate spine switch state
14:     𝕊 = { }
15:     for (l ∈ cands) do
16:         if (check(l.id, D) and l ∉ ℂ) then //false positive
17:             𝕊 = 𝕊 ∪ {⟨l.src, l.id⟩} // add link to state
18:     return ⟨D, 𝕊⟩
```

Communications Overheads. Orca achieves substantial bandwidth savings compared to the commonly-used unicast model. Orca, on the other hand, attaches a small, fixed-sized label (19 bytes) to each packet in typical datacenters; Orca label is smaller than the IP header. In addition, Orca uses additional bandwidth between leaf switches and agents deployed on servers within the same rack. Prior studies, however, reported that links at leaf layer are under-utilized. For instance, the study in [38] has found that the datacenter edge is lightly utilized: 80% of the time, the utilization is less than 10% for cloud and enterprise datacenters. A recent study by Facebook [20] reported that links between leaf switches and servers have a 1-minute average utilization of less than 1%.

A.4 Extensions and Limitations of Orca

Multipath Routing. In Orca, the multicast tree has one path from the source VM to any core switch, then it reaches the receivers by branching to spine and leaf switches. Orca can support multipath routing to achieve reliability and load balancing as follows. The centralized controller can compute multiple trees, each has the same source and receivers of the session but consists of different links. For example, the centralized controller can choose a different core switch as the root for each tree. It then calculates a different source label

Algorithm 3 Process a spine downstream label.

Input: D: spine downstream label
Input: l: downstream link attached to the spine switch
Input: $\mathbb{C}$: set of common ports in spine downstream switches
Input: $State$: state maintained at the spine switch
Output: **true** if duplicating a pkt on link l, else **false**

```
    // Runs for every link attached to the spine switch
1:  function ProcSpineDSLabel(D, l, State)
        // Links belonging to ℂ
2:      If index(l.id) ∈ ℂ then return true
        // Checking the filter for index(l.id) ∉ ℂ
3:      If not check(l.id, D) then return false
        // sID is the session ID included in the packet header
4:      return l.id ∉ State[sID]
```

for each tree, and instructs the source VM to store the new labels and spine switches to maintain state (if needed). The source attaches different source labels to the packets to instruct switches to forward them on links of different trees. Leaf labels are identical for all trees as they have the same receivers. As in other multipath routing systems, packet reordering may occur in this case and would need to be handled by the application.

Reliability and Congestion Control in Multicast. Prior works, e.g., [9, 31], proposed various methods for reliable transmission and congestion control for datacenter multicast. These methods can be used on top of Orca. In addition, the Orca agent can reduce the number of control messages, e.g., ACK or NACK, since it can aggregate them per rack.

Incremental Deployment. Orca can run on legacy switches by encapsulating its labels in VLAN or VXLAN headers. The header identifier can be used to instruct switches to duplicate incoming packets.

Limitations of Orca. Deploying Orca in graph-based datacenter networks, e.g., Jellyfish [34] and Xpander [13], may require changes in some components of Orca. For example, although our server-assisted approach will work at the leaf layer in Jellyfish, Jellyfish's lack of structure does not allow Orca to use the same algorithms at other layers. A new label calculation algorithm would need to be designed to encode tree links without imposing assumptions on their layers.

A.5 Additional Simulation Results

We evaluate the running time of Orca and Elmo spent by the centralized controller when sessions change.

Running Time. Orca is simple and enables more updates per second to be processed by the control plane. For the Uni-S dataset, the average running time for Orca to calculate a session label is 0.34 ms (SD is 0.4 ms), while this average is up to 7.286 ms (SD is 9.8 ms) for Elmo-3. The average (SD) running times for Elmo-1, Elmo-2 and Elmo-4 are 6.1 ms (5.7 ms), 5.5 ms (5.6 ms) and 6.5 ms (8.6 ms), respectively. These times were measured on a workstation with a 2.3 GHz CPU.

***Summary**: Orca calculates labels 21X faster than Elmo.*