Optimizing Multicast Flows in High-Bandwidth Reconfigurable Datacenter Networks

Long Luo[a], Klaus-Tycho Foerster[b], Stefan Schmid[c] and Hongfang Yu[a]

[a]University of Electronic Science and Technology of China, Chengdu, P.R. China
[b]TU Dortmund, Germany
[c]TU Berlin, Germany, University of Vienna, Austria, and Fraunhofer SIT, Germany

ARTICLE INFO

Keywords:
Computer networks
Multicast communication
Reconfigurable architectures
Scheduling algorithms

ABSTRACT

Modern cloud applications has led to a huge increase in multicast flows, which is becoming one of the primary communication patterns in nowadays datacenter networks. Emerging datacenter technologies enable interesting new opportunities to support such multicast traffic more effectively and flexibly in the physical layer: novel circuit switches offer high-bandwidth and reconfigurable inter-rack multicasting capabilities. However, not much is known today about the algorithmic challenges introduced by this new technology, especially in optimizing the completion times for multicast flows.

This paper presents SplitCast, a preemptive multicast scheduling approach that fully exploits emerging high-bandwidth physical-layer multicasting capabilities to reduce flow times. SplitCast dynamically reconfigures the circuit switches to adapt to the multicast traffic, accounting for reconfiguration delays. In particular, SplitCast relies on simple single-hop routing and leverages transfer flexibilities by supporting *splittable* multicast so that a transfer can already be delivered to just a subset of receivers when the circuit capacity is insufficient. Moreover, SplitCast supports two common forwarding models, the all-stop and the not-all-stop, during circuit reconfiguration. We conduct extensive simulation to evaluate the performance of SplitCast, and the results show that SplitCast can cut down flow times significantly compared to state-of-the-art solutions.

1. Introduction

With the vast popularity of data-centric applications, it is expected that datacenter traffic will continue to grow explosively in the coming decades, pushing today's datacenter designs to their limits. Accordingly, we currently witness great efforts to design innovative datacenter topologies, offering high throughput at low cost [1, 2, 3, 4, 5, 6, 7, 8, 9], or even allowing to adjust networks adaptively, in a demand-aware manner [10, 11, 12].

A particularly fast-growing communication pattern in data centers today are *multicasts*. Data-centric applications increasingly rely on *one-to-many communications* [13, 14, 15, 16], including distributed machine learning [17], applications related to financial services and trading workloads [18, 19], virtual machine provisioning [20], pub/sub systems [21], etc. [22, 23]. In many of these applications, multicasts are frequent and high-volume and are becoming a bottleneck [13, 24, 25, 14, 26]. For example, in distributed machine learning frameworks, the learning model has to be frequently communicated among all computation nodes during the long-term training process [17, 27]. Despite the wide use of multicast, there is currently no support among cloud providers for efficient multicasting: a missed opportunity [24, 13, 28].

Our paper is motivated by emerging optical technologies which can potentially improve the performance of transferring multicast flows by supporting in-network multicast on the physical layer. In particular, recent advancements for reconfigurable circuit switches (RCS) allow to set up high-bandwidth port-to-multiport circuit connections which support adaptive and demand-aware multicasting among top-of-rack (ToR) switches [29, 30, 31, 25, 26, 14, 12]. See the example in Fig. 1(a) for an illustration: in this example, the RCS can be used to first create a circuit connection to directly multicast data from ToR 1 to ToRs 3 and 4, after which the RCS is reconfigured to support multicast from ToR 2 to ToRs 1, 3, and 4 in a demand-aware manner (see Fig. 1(b)). In general, the possibility to simultaneously fan out to multiple receiver racks over a single port-to-multiport circuit connection can greatly improve the delivery of multicast flows.

However, while this technology is an interesting *enabler*, we currently lack a good understanding of how to algorithmically *exploit* this technology to optimize multicast communications. While interesting first approaches such as Blast [25] and Creek [26] are emerging, these solutions are still limiting in that they cannot fully use the high capacity of the circuit switch as they only transfer data as long as a circuit connection can be created to match *all* the receivers of a flow.

Motivating Example. Consider the situation shown in Fig. 2, where there are three multicast flows f_1, f_2 and f_3, each of unit size. In this example, each node denotes one port of the circuit switch and connects a ToR. Using the state-of-the-art solution Blast [25] (single-hop) or Creek [26] (multi-hop), the three flows have to be delivered one by one: the output of a circuit (right) can only receive traffic from a single input (left) on the other side of the circuit, while any two of these three flows compete for one outport (f_1 contends with f_2, f_3 at outport R_3, R_2, respectively, and f_2 contends with f_3 at outport R_1). Hence, a multicast flow can only be sched-

ORCID(s): 0000-0003-1028-8178 (L. Luo); 0000-0003-4635-4480 (K. Foerster); 0000-0002-7798-1711 (S. Schmid); 0000-0002-5219-1780 (H. Yu)

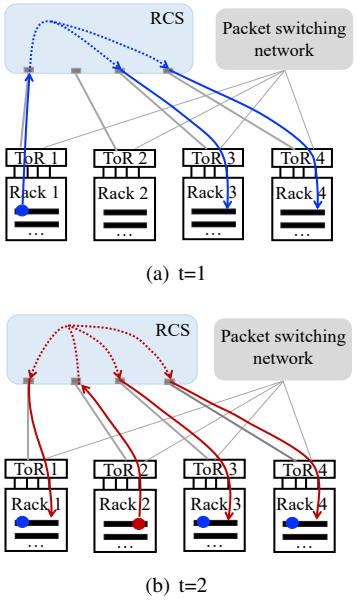


Figure 1: A reconfigurable datacenter network based on a reconfigurable circuit switch (RCS) allows to enhance the conventional packet switching network with multicast capabilities: e.g. at time $t = 1$, RCS supports multicasting ToR 1 to ToRs 3 and 4, while at time $t = 2$, from ToR 2 to ToRs 1, 3 and 4.

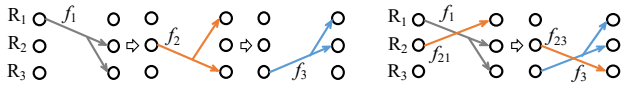


Figure 2: A splittable multicast decreases the average flow time by 22.2%, compared to a non-splittable multicast.

uled at a time when a port-to-multiport circuit connection is set up between the corresponding ToRs matching its sender and receivers as shown in Fig. 2(a). The average flow time (*i.e.,* average of the multicast flow duration) is $\frac{1+3+5}{3} = 3$ units of time if reconfiguring circuit connections takes one unit of time.

However, if just matching the multicasts, we observe that there are some *free* ports: an unused optimization opportunity we want to exploit in this work. For example, R_1, R_2, R_3 in Fig. 2(a) are unused when f_1, f_2, f_3 are delivered, respectively. In principle, all these ports can be fully used if one splits and schedules f_2 in two steps, each transferring data to one of its receivers, as shown in Fig. 2(b). This reduces the average flow time to $\frac{1+3+3}{3} = 2.3$ units of time, decreasing the average flow time by 22.2% compared to the non-splittable solution in Fig. 2(a).

Problem. This simple example motivates us, in this paper, to study the opportunities to improve the multicast performance by supporting *splittable multicasting*: multicast data which can be sent just to a subset of receivers at a time by allowing circuit switch to strategically build port-to-multiport circuit connections between ToRs. Essentially, a circuit connecting an input port to multiple outports forms a hyperedge and creating port-to-multiport circuit connections for multicast data produces a hypergraph matching problem [32]. We are particularly interested in the *splittable multicast matching problem*: a sender port/rack is allowed to match to just a subset of the receiver ports/racks of a multicast flow at a time.

More specifically, we consider the non-negligible time delay for reconfiguring the circuit switch and multicast demands with different flow sizes, where our objective is to design a scheduler which reconfigures the circuit switch in a demand-aware manner, in order to minimize *multicast flow times*. In our model, a created port-to-multiport circuit connection allows one ToR to simultaneously transmit to multiple ToRs at bandwidth b_c. Inter-rack multicast flows arrive in an online manner, and a scheduled flow transmits from server to the associated ToR first via a link with bandwidth $b_s \leq b_c$.

Our Contributions. This paper presents SplitCast, an efficient scheduler for multicast flows in reconfigurable datacenter networks, which leverages splitting opportunities to improve performance of multicast transfers. By dynamically adjusting the network topology and supporting splittable multicasting, SplitCast fully exploits the available infrastructure, reducing flow times and improving throughput. SplitCast is also simple in the sense that it relies on segregated routing: similar to reconfigurable unicast solutions, routing is either 1-hop along the reconfigurable circuit switch or along the (static) packet switching network, but never a combination.

We provide insights into the algorithmic complexity of a simplified problem variant, scheduling bounds, and present a general problem formulation. We also show how to tackle the challenges of splittable flows, how to account for reconfiguration delays, and how to incorporate different circuit switch reconfiguration models. Our extensive simulations indicate that SplitCast can significantly outperform state-of-the-art approaches: not only are the flow times reduced by up to 9× but also the throughput increased by up to 3×.

Organization. Our paper is organized as follows. We first review related work in §2 and theoretically analyze a fundamental version of the multicast matching problem and greedy scheduling in §3. We next provide a formulation and motivating example for the generalized scheduling problem, and discuss the challenges of splittable multicast scheduling in §4. In §5, we present our solution, SplitCast, in detail. After reporting the simulation results in §6, , we conclude this work in §7.

2. Related Work

Multicast communication in datacenter networks
Multicast is a typical communication pattern of many datacenter applications, such as the data dissemination of publish-subscribe services [21], distributed caching infrastructures updates [24] and state machine replication [33]. Multicast

communications are growing explosively in scale due to the proliferation of applications based on big data processing frameworks [34, 35, 36]. In order to improve the performance of multicast transfers, many works turn to IP multicast and focus on issues like scalability and reliability [37, 38, 13] of the deployment of IP multicast in the datacenter context. Recent work also investigated the benefits of P4 in multicast [39] and of different topologies, e.g., multi-root networks [40]. However, all these above proposals consider multicast over static networks, and hence do not investigate the benefits of reconfigurability.

Reconfigurable datacenter networks
One of the recent technology innovations in networking research is the possibility of providing reconfigurable high-bandwidth inter-rack connections at runtime, with reconfigurable circuit switches. Importantly, such technology also supports high-performance inter-rack multicasting. This capability of data multicast is enabled by circuit switching technologies [14, §I], e.g., optical circuit switches (OCS) [25, 26], free-space optics (FSO) [29, 10], and 60 GHZ wireless links [41, 42, 43].

This work is in turn motivated by such novel circuit switching technologies and focuses on scheduling algorithms for multicast flows. The most related works are Blast [25] and Creek [26]. Both transfer data only when there exists a circuit connection matching all the receivers of a multicast flow. In contrast, we also allow a flow to transfer data when a circuit connection matching only a subset of its receivers.

We are also inspired by works leveraging the so-called not-all-stop switch model [44], where, similar to FSO, creating or tearing down some circuits does not interrupt the other circuits. Circuit switches leveraging not-all-stop model are available off-the-shelf and have also shown to be multicast feasible in an FSO setting [29]. While not-all-stop model has been considered to reduce flow completion time for coflows [44], and in standard unicast traffic as well [10, 45], we are not aware of other multicast scheduling work investigating this model

There also emerge further several works [46, 11, 47, 48, 49] on demand-aware reconfigurable datacenter networks. Schmid *et al.* [46, 11, 47] focus on designing topologies towards better network-wide performances, and Salman *et al.* [48] and Wang *et al.* [49] focus on learning the topology. They consider unicast flows and are all orthogonal to this work. We refer to [50] for a recent survey.

Frame-based multicast scheduling
Different from the frame-based multicast scheduling problem in input-queued switches that takes flow *rates* as demands, we focus on *size-based* traffic demands. The former aims to decompose a rate matrix into multiple switch configurations (permutation matrices) under switch constraints, while we determine not only switch configurations but also how long every computed configuration should maintain. We also consider the switch reconfiguration delay of the practical system while the frame-based multicast scheduling does not. Moreover, our objective is to minimize the flow completion time, while the frame-based multicast scheduling aims at finding permutation matrices that satisfy a given rate matrix. We refer to the work of Sundararajan et al. [51] for an overview on frame-based multicast scheduling.

Reconfigurable Wide Area Networks (WANs)
WANs have also benefited from programmable physical layers, by e.g. leveraging reconfigurable optical add/drop multiplexers (ROADMs) [52]. Various networking research directions were investigated, such as scheduling and completion times [53, 54, 55], robustness [56] and attacks [57], abstractions and optical topology programming [58, 59], connectivity as a service [60], CDN-ISP-interplay [61], and variable bandwidth links [62, 63]. Benefits of multicast were studied by Luo et al. [39], in the context of bulk transfers.

Bibliographical note: A preliminary version of this article appeared in the proceedings of IEEE INFOCOM 2020 [64]. This version was carefully revised and extended. In particular, this journal version mainly includes additional theoretical scheduling bounds, an extension to not-all-stop model of circuit connection, a new algorithm designed for this new model, and an extensive evaluation of its performance benefits.

3. First Insights into the Algorithmic Problem

We first present insights into the fundamental *static* problem underlying the design of datacenter topologies optimized for multicasts. In the following, we consider the multicast matching and the splittable multicast matching problem introduced by the question: how to configure the RCS such that the resulting matching maximizes the number of multicast transfers that can be served simultaneously.

3.1. The Multicast Matching Problem

The multicast matching problem is essentially a hypergraph matching problem [32], this connection has already been observed in [25], where the authors also provide some intuition that "*the multicast matching problem has a similar complexity*". However, the complexity is not proven formally. We first formalize their intuition and then prove it to be correct already for a simple version multicast matching: Given a set of $k \in \mathbb{N}$ sources and receivers and a set of hyperedges, each of the same weight (value/cost) with one source and one or two receivers, can we admit $q \in \mathbb{N}$ such edges into a matching s.t. each node is incident to at most one hyperedge?

Theorem 1. *Multicast matching is NP-hard, already in the unit weight case with at most $k = 2$ receivers per transfer.*

PROOF. We use a variant of the 3-dimensional matching problem for our proof, i.e., given a set $M \subseteq X \times Y \times Z$, where X, Y, Z are disjoint sets of q elements each: Does M contain a subset (matching) M' s.t. $|M'| = q$ and no two elements in M' share a coordinate? This problem has been shown to be NP-hard [65]:

Observe that we can cast (in polynomial time) each such instance as a multicast matching problem: To this end, we translate the hyperedges in M into transfers by setting X

as the sources and Y, Z as the receiver nodes, potentially padding the number of source/receiver nodes to be of equal number. Now, if q such transfers can be admitted simultaneously, it directly translates to a set M' of q elements and vice versa. Hence, we have shown our multicast matching problem to be NP-hard, as it can be used to solve each instance of the NP-hard 3-dimensional matching problem, using only polynomial-time overhead.

We note in this context that Sundararajan et al.[51] showed the problem of deciding the feasibility of a given rate vector in this context to be NP-hard. We note that there are subtle differences between the models, see our discussion in §2. Moreover, we show hardness already for $k = 2$, i.e., more specific, whereas the former only considers the general case.

3.2. Upper Bounds for Online Non-Preemptive Scheduling

In this paper, we are particularly interested in the *online* problem where requests arrive over time in an online fashion and need to be served quickly, requiring fast algorithms. Thus, a scheduling algorithm has to evolve and compute dynamic matchings. To this end, we consider greedy scheduling algorithms, and revisit the approach by Jia et al. [54].

In the online setting, multicast transfer requests arrive at the system in an arbitrary order, and each transfer is associated with one source and multiple (*i.e.*, k) receivers. We assume that the size of every transfer is integer and all links have unit capacity.

Definition 1. At any time slot t, we have a collection of available multicast flows represented by edges in $G(t)$. We go through the flows in any arbitrary order and schedule flows if the port constraints are not violated. This is the Greedy Scheduling proposed by Jia et al.[54].

Jia et al. [54] showed the competitive ratio of greedy scheduling to be $2 + 1 = 3$ in a unicast setting with $k = 1$ receiver. We extend their results to arbitrary k, deriving a bound of $2 + k$:

Theorem 2. *If each multicast flow has k receivers, the greedy scheduling algorithm is* $(2 + k)$*-competitive.*

PROOF. For each flow j, we use r_j, T_j and T_j^* respectively, to denote its arrival time, the time when it is scheduled in the greedy algorithm, and the time when it is scheduled in the optimal offline algorithm. We can use T and T^* to denote the makespan of the greedy algorithm and that of the offline optimal solution. When flow j has a size of v_j, we obtain

$$r_j + v_j \leq T_j^* \leq T^* \tag{1}$$

For the greedy algorithm, a worst case situation arises when a flow j needs to be scheduled when all the flows at its sender node s_j and at its receivers nodes $\boldsymbol{d}_j$ have finished. Thus, T_j can be upper bounded by

$$T_j \leq r_j + \sum_{i \in F(s_j)} v_i + \sum_{d_j \in \boldsymbol{d}_j, i \in F(d_j)} v_i + v_j \tag{2}$$

On the other hand, we know that flow j and the flows in $F(s_j)$ share the sender node s_j and thus the optimal solution has to use at least $\sum_{i \in F(s_j)} v_i + v_j$ time to schedule them, i.e.,

$$\sum_{i \in F(s_j)} v_i + v_j \leq T^* \tag{3}$$

Similarly, at the receiver side, the optimal solution requires at least time $\max_{d \in \boldsymbol{d}_j}(\sum_{i \in F(d_j)} v_i) + v_j$ for scheduling, i.e.,

$$\forall d_j \in \boldsymbol{d}_j, \sum_{i \in F(d_j)} v_i + v_j \leq T^* \tag{4}$$

Putting equation (1), (3) and (4) together into (2), we have

$$\begin{aligned} T_j &\leq r_j + \sum_{i \in F(s_j)} v_i + \sum_{d_j \in \boldsymbol{d}_j, i \in F(d_j)} v_i + v_j \\ &\leq T^* + \sum_{i \in F(s_j)} v_i + \sum_{d_j \in \boldsymbol{d}_j, i \in F(d_j)} v_i \\ &\leq T^* + T^* + kT^* \\ &= (2 + k)T^* \end{aligned} \tag{5}$$

The first inequality is derived from equation (1), and the second one is derived from (3) and (4).

4. Problem, Approach and Challenges

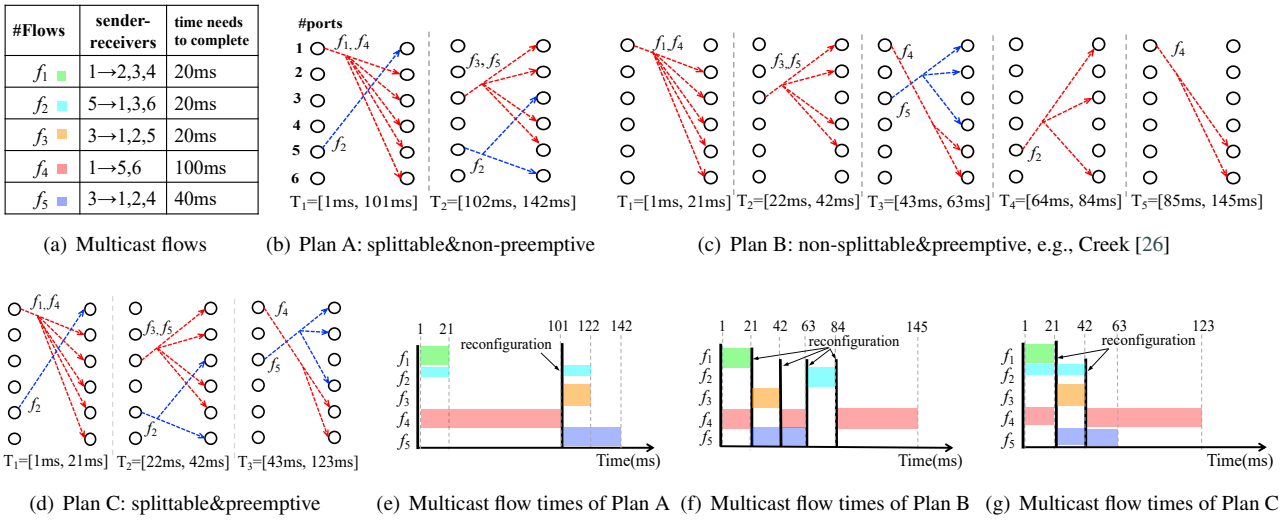
Given the first insights in §3, we now formulate the general multicast scheduling problem considered in this paper.

4.1. Problem Setting

We consider a hybrid datacenter including a reconfigurable circuit switching network and a packet switching network, recall Fig. 1. We focus on the former that is enabled by the high-bandwidth circuit switch, as in Blast [25] and Creek [26], and consider multicast flows that arrive over time and can be of arbitrary size. Our main goal is to minimize *flow time*, but we will also consider a throughput maximization objective.

In our reconfigurable network model, every ToR is directly connected to the circuit switch via an exclusive port. The bandwidth of a circuit switch port is generalized to b_c, which is typically multiple times the bandwidth b_s of the link between a server and a ToR. For example, b_c can be 10Gbps, 40Gbps, 100Gbps and beyond while b_s is 10Gbp [25, 26, 14]. The circuit switch can constantly reconfigure high-speed port-to-multiport circuit connections between the ToRs, but comes with a reconfiguration delay and stops transmitting data during the reconfiguration period. Any two circuit connections can share neither sender port nor receiver ports [8, 14, 25, 26].

A multicast flow f is characterized by a tuple $(s_f, \boldsymbol{d}_f, v_f, t_f^{\text{arr}})$, where s_f, $\boldsymbol{d}_f$, v_f, and t_f^{arr} denote the sender rack, the *set* of receiver racks, the data volume, and the release time, respectively. By taking advantage of the application



(a) Multicast flows (b) Plan A: splittable&non-preemptive (c) Plan B: non-splittable&preemptive, e.g., Creek [26]

(d) Plan C: splittable&preemptive (e) Multicast flow times of Plan A (f) Multicast flow times of Plan B (g) Multicast flow times of Plan C

Figure 3: Preemptive and splittable scheduling can significantly speed up the flow time, compared to the solutions that are just based on splittable or preemptive.

knowledge, the multicast group membership and traffic volume is available upon flow arrival [66, 25]. We assume segregated routing model in this work, where a flow routed using either a single-hop circuit switching or multihop packet switching, but not a combination of both [10, 47]. Such a single-hop segregated routing avoids moving traffic between the circuit and the packet switching network.

The question is to decide which circuit connections to create and which flows to transfer over these connections, optimizing the objective and accounting for the circuit switch constraints.

4.2. Scheduling Approach: Splittable and Preemptive Multicast

Flows are scheduled epoch by epoch, where in each of the epochs the circuit connections and the set of serving flows are fixed. At the beginning of an epoch, we determine the circuit connections to be set up as well as the flows to be served in this epoch. We also have to determine the epoch duration in order to properly trade off reconfiguration overhead against sub-optimal configurations. Overall, we adopt splittable and preemptive data transfers.

We briefly explain the key idea of our solution and its advantages over splittable-only and preemptive-only solutions in Fig. 3, respectively. Using splittable transfers, the scheduling algorithm may transfer data to just a subset of the receivers of a multicast flow in an epoch. For example, for a flow f_2, the circuit switch may transfer data to its receiver 1 in the first epoch and its receivers 3, 6 in the second epoch by building port-to-multiport connections between the source ToR and the destination ToRs corresponding to the selected subset of receivers, respectively, as shown in Fig. 3(b).

Additionally, a circuit connection can be shared by multiple matched flows to fully use the high-bandwidth capacities of circuit switches. As the bandwidth (b_c) of per circuit switch port is assumed to be twice the fan-in rate (b_s) of each flow, f_1 and f_4 are simultaneously transferred over a circuit connecting inport 1 and outports 2 to 6 in the first epoch in Fig. 3(b). Therefore, f_1 also reaches the non-receiver ToRs via outports 5, 6. However, it is necessary to prevent the non-receiver ToRs from further forwarding f_1 to non-destination racks for storage efficiency. To this end, we install ToR forwarding rules only for the flows with this rack destination. Hence, the ToRs connecting outports 5, 6 will directly discard the packets of f_1 due to no matching rule.

Using preemptive transfers, the scheduling algorithm may reconfigure the circuit connections and reallocate the connections to the most critical flows before the completion of serving flows. For example, the circuit switch can be reconfigured with a delay of 1ms and flow f_4 is preempted by f_3 and f_5 after the completion of f_1 shown in Fig. 3(c). With preemption, the average flow time (see Fig. 3(f)) is sped up 1.43× over the splittable but non-preemptive plan shown in Fig. 3(e). Moreover, by combining splittable and preemptive scheduling (Fig. 3(d)), the average flow time (Fig. 3(g)) can be sped up 1.74× and 1.22× over the only splittable (Fig. 3(b)) and preemptive plan (Fig. 3(c)), respectively.

4.3. Mathematical Formulations

We now present our formulation for the splittable multicast problem in an epoch, using the notations shown in Table 1. We determine the circuit connections, the flows to-be-scheduled and the length of a concerned epoch t. We first point out the constraints of building circuit connections and of scheduling flows and then formalize the objectives.

Constraints. Constraint (6) expresses that the output of a circuit connection can only receive traffic from a single input on the other side of the circuit. Constraint (7) constricts the transmission rate of flows from a rack i to the circuit should not exceed the circuit port bandwidth b_c. Constraint (8) states that a flow f with a set of receiver racks $\boldsymbol{d}_f$ could transfer data to its receiver rack $d \in \boldsymbol{d}_f$ via the circuit

Table 1
Key notations used problem formulations

Network model	
n	the number of all racks connecting to circuit switch
b_s	bandwidth of per server NIC port
b_c	bandwidth of per circuit switch port, $b_c \geq b_s$
δ	reconfiguration time of switch circuit
Multicast flow f	
s_f	the sender rack
$a_{f,i}$	indicator whether rack i is the sender rack
$\boldsymbol{d}_f$	the set of receiver racks
v_f	the (remaining) flow size
t_f^{arr}	the arrival time
Internal and decision variables for epoch t	
$x_{i,j}^t$	binary: indicate whether there is a circuit connection destining to rack j from rack i
w_f^t	binary: indicate whether flow f is to be scheduled
$w_{f,d}^t$	binary: indicate whether flow f transfers data to its receiver d
θ^t	the time duration of the concerned epoch

switch only if there is a circuit connection originating from the sender rack s_f to d. Constraint (9) together with (10) express that a flow f is to be scheduled as long as at least one of its receivers is to be served.

$$\forall j \,:\, \sum_i x_{i,j}^t \leq 1 \tag{6}$$

$$\forall i \,:\, \sum_f b_s a_{f,i} w_f^t \leq b_c \tag{7}$$

$$\forall f, d \in \boldsymbol{d}_f \,:\, w_{f,d}^t \leq x_{s_f,d}^t \tag{8}$$

$$\forall f \,:\, w_f^t \leq \sum_{d \in \boldsymbol{d}_f} w_{f,d}^t \tag{9}$$

$$\forall f, d \in \boldsymbol{d}_f \,:\, w_f^t \geq w_{f,d}^t \tag{10}$$

Objectives. A most fundamental objective is to maximize the average throughput of every concerned epoch t, which can be formulated as

$$\max g(\boldsymbol{w}^t, \theta^t) = \frac{\sum_f \sum_{d \in \boldsymbol{d}_f} \min(v_{f,d}^t, b_s \theta^t) w_{f,d}^t}{(\theta^t + \delta)} \tag{11}$$

where $v_{f,d}^t$ denotes the size of data that needs to be transferred to the receiver d of flow f before epoch t. It is easy to compute the total size of data transferred over the circuit switch in epoch t through $\sum_f \sum_{d \in \boldsymbol{d}_f} \min(v_{f,d}^t, b_s \theta^t) w_{f,d}^t$.

A second important objective is to minimize the flow times. As we are just planning epoch-by-epoch, we cannot know the exact flow times of unfinished flows. However, we could know the lower bound of the minimum flow times and optimize it via

$$\min h(\boldsymbol{w}^t, \theta^t) = \sum_f \sum_{d \in \boldsymbol{d}_f} (I(v_{f,d}^t > b_s \theta^t w_{f,d}^t)(\theta^t + \delta) + I(v_{f,d}^t < b_s \theta^t w_{f,d}^t) \frac{v_{f,d}^t}{b_s} + t^{\text{start}} - t_f^{\text{arr}}) \tag{12}$$

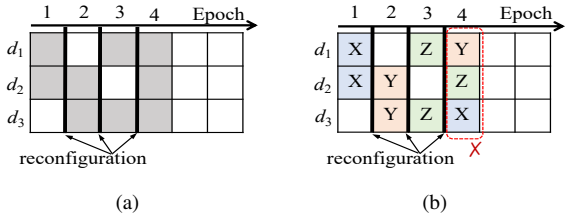


Figure 4: The challenge of receiver asynchronization: All the receivers of a splittable flow have not yet received all requested data (see (b)) despite the equivalent volume of requested data (see (a)).

where $I(v_{f,d}^t > b_s \theta^t w_{f,d}^t)(\theta^t + \delta)$ is the time experienced by the receiver d of flow f if it has not finished by the end of epoch t, and $I(v_{f,d}^t < b_s \theta^t w_{f,d}^t) \frac{v_{f,d}^t}{b_s}$ otherwise. t^{start} is the start time of epoch t and t_f^{arr} is the arrival time of flow f.

We can see that the above formulations result in Integer Linear Programs (ILPs) for the given epoch length θ^t. However, the length θ^t of every concerned epoch t is also subject to optimization for the multicast flow scheduling problem in reconfigurable datacenter networks, which makes the objective function become non-linear and introduces additional challenges (see the next subsection).

4.4. Challenges

In this work, we would like to exploit the flexibility of adapting the epoch lengths dynamically and also consider an online problem that needs to schedule multicast flows over time. This means that we not only determine the circuit connections $\boldsymbol{x}^t$ and the flow scheduling decisions $\boldsymbol{w}^t$ as formulations in §4.3 but also the epoch duration θ^t for every concerned epoch t to optimize the multicast flows in reconfigurable datacenter networks. However, we can see that the circuit connections $\boldsymbol{x}^t$, the flow scheduling decisions $\boldsymbol{w}^t$ and the epoch duration θ^t actually interact with each other and they together determine the network throughput and the sum of flow times. This renders it challenging to find an optimal solution in practice. In addition to the above challenges in the optimization formulations, there is also a synchronization issue introduced by splittable multicasting, as we discuss in more details in the following.

Let's consider an illustrative flow f_1 with three receivers d_1, d_2 and d_3 that request three units of data. Assuming that the circuit switch has unit capacity per port and can only build a circuit connecting two of the three receivers in the first three epochs with unit time length, due to one port that has been taken by other circuit connections. In particular, considering that a unit of data has been transferred to (d_1, d_2), (d_2, d_3), and (d_1, d_3) in the first, the second, and the third epoch, respectively, as shown in Fig. 4(a). When it comes to the fourth epoch, the circuit switch now can create a circuit connecting all the receivers. Seemingly, f_1 could be completed as d_1, d_2, d_3 will receive three units of data,

the total requested data size, at the end of the fourth epoch.

However, f_1 cannot been completed because two receivers did not receive the data they request in the last epoch. Assume that the requested data is {X, Y, Z} (each has a unit size) and that X, Y and Z are transferred in the first, the second and third epoch, respectively, as illustrated in Fig. 4(b). In the fourth epoch, as the circuit can only transfer either X or Y or Z, two receivers cannot receive their lastly requested data. We refer to this as receiver asynchronization problem.

When scheduling splittable flows, one should be careful to handle receiver asynchronization to ensure that every multicast receiver obtains all requested data, which is challenging in general. A naive solution is to partition the receivers of a flow into multiple non-neighboring subsets, consider each subset of receivers together with the sender as a subflow and independently scheduled these subflows without allowing further splitting during the transmission. However, such a premature fixed partition cannot adapt to the traffic dynamics, and it is difficult to determine an optimal partition in advance [67].

5. SplitCast: Efficient Optical Multicast Scheduling over Reconfigurable Networks

Given the motivation and challenges discussed above, we now present details of our scheduler, SplitCast. SplitCast solves the multicast scheduling problem in reconfigurable datacenter networks, in an online and efficient manner.

Overview. In a nutshell, SplitCast works in an epoch by epoch manner. At the beginning of an epoch, SplitCast creates circuit connections, chooses flows to be transferred and determines the time duration of this epoch. SplitCast chooses the to-be-served flows and creates circuit connections in a hierarchical fashion, see the pseudo-code in Alg. 1. It first picks the flows where all receivers can be served without splitting and creates the circuit connections that match them (line 1, Alg. 1), and then determines the epoch duration according to the picked flows (line 2, Alg. 1). Subsequently, SplitCast searches for more flows where subsets of their receivers can be served by employing or extending the created circuit connections (line 3, Alg. 1).

5.1. Creating Circuit Connections and Scheduling Flows

In our algorithm, a circuit configuration is modeled as a directed hypergraph H, where each node denotes a rack and each directed hyperedge denotes a circuit connection, as in [25, 26]. The creation of circuit connections is modeled as adding directed hyperedges (without sharing tail node and head nodes) to the hypergraph H. Initially, the hyperedge set is empty.

Given a set F of multicast flows, we consider flows in a shortest remaining processing time first (SRPT) manner to determine a subset of flows where all their receivers can be matched by creating circuit connections under the circuit switch constraints. We consider flows and create circuit connections in two rounds. In the first round, we only consider the multicast flows with all receivers (line 9-12, Alg. 1). In the second round, we consider the subflows of multicast flows (line 13-16, Alg. 1), where subflows are products of splitting flows, and a subflow includes a subset of receivers, as explained later.

The scheduler considers a flow to be servable if the following two conditions are satisfied: i) the number of to-be-served flows that use the hyperedge originating from its sender is less than $\frac{b_c}{b_s}$ (line 2, Alg. 2), ii) all the receivers of this flow are included in the hyperedge originating from its sender (line 13, Alg. 2). Once such a flow is found, a directed hyperedge is added from the sender to all the receivers if no hyperedge originates from the sender (line 19, Alg. 2). Otherwise, the directed hyperedge originating from the sender is extended to include the unconnected receivers (line 21, Alg. 2).

5.2. Calculating the Epoch Length

We determine the epoch length according to the created circuit connections (via the hypergraph H) and the to-be-served unsplittable flows found in the last step. Given a set of to-be-scheduled flows, the network throughput function $g(\boldsymbol{w}^t, \theta^t)$ and flow time function $h(\boldsymbol{w}^t, \theta^t)$ can be proven to have unique extreme values [26] because the flow scheduling decisions $\boldsymbol{w}^t$ are already known. Actually, the optimal epoch length is related to the completion times of the flows to be served. Thus, we enumerate all possible epoch lengths and pick the one that achieves the optimal value of (11) or (12). Taking the objective function of maximizing throughput $g(\boldsymbol{w}^t, \theta^t)$ as an example, we can compute a possible epoch length value $\frac{v^t_{f,d}}{b_s}$ for every scheduling flow f with $w^t_{f,d} = 1$. Then, it is easy to compute the corresponding throughput under every possible epoch length value and choose the one that yields the maximum throughput. Optimizing the objective function (12) is similar. Given the determined length θ_t of an epoch t, every to-be-served flow can send up to $\theta_t b_s$ of data in this epoch. Accordingly, we can update the completion status (*i.e.*, completion time and sent size) of every scheduling flow and know whether a flow can be completed by the end of this epoch.

5.3. Scheduling Splittable Flows

In order to fully use the remaining circuit capacity, we further schedule flows for which subsets of receivers can still be served (line 15, Alg. 2). Recall the example flow f_1 in Fig. 4, where the switch could only send a unit of data to d_1 and d_2 in the first epoch. At the end of this epoch, our solution will decrease the remaining size of f_1 by one and create a subflow f_{11} with the unserved receiver d_3: the flow size equals to the amount of untransmitted data in this epoch. The algorithm works similarly in the second and the third epoch: it decreases the remaining size of f_1 by one and creates a subflow f_{12} with the unserved receiver d_1 and f_{13} with the unserved receiver d_2, respectively. In the following epochs, subflows are scheduled independently with other flows, which is easy to operate. In addition, if a newly

Algorithm 1 Splittable Multicast Scheduling Algorithm

Input: A set F of flows to be scheduled in an epoch;
Output: A hypergraph H of the circuit configuration, a set F^{serve} of to-be-served flows and the epoch duration θ;
1: $(H, F^{\text{serve}}) \leftarrow$ NONSPLITSCHEDULE(F);
2: $\theta \leftarrow$ CALCULATEEPOCHLENGTH(F^{serve}, δ);
3: $(H, F^{\text{serve}}_{\text{split}}) \leftarrow$ SPLITSCHEDULE($H, F \setminus F^{\text{serve}}, \theta$);
4: $F^{\text{serve}} \leftarrow F^{\text{serve}} \cup F^{\text{serve}}_{\text{split}}$;

5: **procedure** NONSPLITSCHEDULE(F)
6: $F^{\text{serve}} \leftarrow \emptyset$;
7: F.order($policy$ = SRPT);
8: Initialize a hypergraph H to include the nodes of all racks and an empty hyperedge set;
9: **for** $f \in F$ **do** ▷ Check every multicast f
10: $schedule \leftarrow$ CREATECIRCUIT(H, f, *False*);
11: Add f to F^{serve} if $schedule$ is *True*;
12: **end for**
13: **for** $(f_s, f) \in F$ **do** ▷ Check every subflow f_s of each multicast f
14: $schedule \leftarrow$ CREATECIRCUIT(H, f_s, *False*);
15: Add f_s to the to-be-served subflow list of f and add f to F^{serve} if $schedule$ is *True*;
16: **end for**
17: **return** (H, F^{serve})
18: **end procedure**

19: **procedure** SPLITSCHEDULE(H, F', θ)
20: $F^{\text{serve}}_{\text{split}} \leftarrow \emptyset$;
21: F'.order($policy$ = SRPT);
22: **for** $f \in F'$ **do**
23: **try** CONSOLIDATESUBFLOW(SF_f); ▷ SF_f stores all unfinished subflows of f
24: **for** $f_s \in SF_f$ **do**
25: $schedule \leftarrow$ CREATECIRCUIT(H, f_s, *True*);
26: **if** $schedule$ **then**
27: Add f_s to the to-be-served subflow list of f;
28: $F^{\text{serve}}_{\text{split}}$.add($f$);
29: Create a subflow f'_s if f has unserved receivers;
30: **try** MERGESUBFLOW(SF_f, f'_s);
31: **end if**
32: **end for**
33: **end for**
34: **return** $(H, F^{\text{serve}}_{\text{split}})$
35: **end procedure**

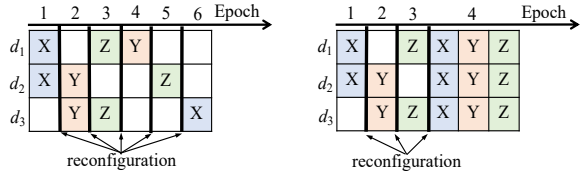
created subflow has the same receivers as other unfinished subflows, it is natural to merge them into a larger one: the resulting flow size equals to the sum of the sizes of the merged subflows (line 30, Alg. 1).

5.4. Complexity Analysis and Discussion

We first analyze the complexity of our algorithm and then discuss an opportunity for further performance improvement. The computation time of Alg. 1 depends on the time to schedule non-splittable flows, the time to compute the epoch length, and the time to schedule the splittable flows. They have a time complexity of $\mathcal{O}(g \log g + gn + gn^2)$, $\mathcal{O}(g)$, $\mathcal{O}(g \log g + gn^2)$, respectively, where g is the number of

Algorithm 2 Create Circuit Connections

procedure CREATECIRCUIT(H, f, $split$)
  if not H.hasfreeCapacity(s_f) **then**
    return H, *False*
  end if
  $r^{list}_{cover} \leftarrow \emptyset$, $r^{list}_{outlier} \leftarrow \emptyset$;
  for $d \in \boldsymbol{d}_f$ **do**
    if H.predecessor(d) = s_f **then**
      r^{list}_{cover}.add(d);
    else if H.inDegree(d) = 0 **then**
      $r^{list}_{outlier}$.add(d);
    end if
  end for
  if not $split$ and len($r^{list}_{cover} + r^{list}_{outlier}$)< $|\boldsymbol{d}_f|$ **then**
    return *False*
  else if $split$ and len($r^{list}_{cover} + r^{list}_{outlier}$)= 0 **then**
    return *False*
  end if
  if H.outDegree(s_f)= 0 **then**
    H.addHyperedge(s_f, $r^{list}_{outlier}$);
  else
    H.extendHyperedge(s_f, $r^{list}_{outlier}$);
  end if
  H.decreaseCapacity(s_f);
  $\boldsymbol{d}^{\text{unserved}}_f \leftarrow \boldsymbol{d}_f \setminus \left(r^{list}_{cover} \cup r^{list}_{outlier}\right)$;
  return *True*
end procedure



(a) Possible scheduling solution A (b) Possible scheduling solution B

Figure 5: Subflow scheduling

flows considered by the algorithm and where n is the number of racks. In total, the computation complexity of our scheduling algorithm is therefore $\mathcal{O}(g(\log g + n^2))$.

While our algorithm performs well as we will see in the evaluation section, it offers an interesting opportunity to further improve the transfer performance through subflow consolidation when scheduling splittable flows. Recall that the multicast flow f_1 in Fig. 4 has three subflows, f_{11}, f_{12} and f_{13}, at the end of the third epoch. If the fourth epoch has only one unit of time, it is impractical to simultaneously finish f_{11}, f_{12} and f_{13} over a circuit connection between the sender and all receivers due to the receiver asynchronization problem. The circuit switch has to schedule these subflows one by one and reconfigures the circuit connections for scheduling each of them, as shown in Fig. 5(a). However, if the fourth epoch lasts for three units of time, we could consolidate these subflows as a larger one and finish them over the circuit connection to all their receivers without more reconfigurations, as shown in Fig. 5(b). In our

algorithm, we use a greedy subflow consolidation method (line 23, Alg. 1). For every considered multicast flow, we first determine which of its receivers could be matched, if one more hyperedge is added subject to the capacity constraints. Then, we greedily consolidate the subflows which can be matched by this hyperedge, and we stop before the remaining sizes of the consolidated subflows exceeds θb_s, θ is the time length computed for current epoch.

5.5. Scheduling under not-all-stop model

So far in our model and algorithms, as well as in prior related work [25, 26] in our setting, a circuit reconfiguration implied that *all* circuits are offline for some period. However, this hardware limitation is not always present and more fine-grained reconfigurations may be possible, for example in free-space optics [29, 10, 45], where the laser paths can be adjusted independently. Moreover, there is also various off-the-shelf circuit switch hardware which does not block all circuits during reconfigurations [44].

Using the terminology of Huang et al. [44], we will denote this model as the *not-all-stop* model, where we might refer to our standard model as the *all-stop* model, if not clear from the context. not-all-stop model allows for localized reconfigurations, not disturbing the other ongoing flows, which can be very useful to further reduce flow completion time [44, 10, 45]. More formally, during reconfiguration in not-all-stop model, communication are blocked only on reconfigured circuits that are newly established or removed, while the circuits which are not being reconfigured are can still be used for sending traffic during the reconfiguration. The benefits of not-all-stop model in improving bandwidth utilization and further reducing flow time is apparent, compared to all-stop model. We next propose how to leverage not-all-stop model in our setting, in order to further reduce flow completion time.

Algorithm Design. We slightly modify Alg. 1, to adapt it to not-all-stop model. The main idea of our variant scheduling algorithm is to replace the epoch calculation method in Alg. 1, Line 2 with a new one shown in Alg. 3.

Once there is a new flow coming or some existing flow being completed, we opportunistically reschedule flows on available free ports. When some existing flow completes, we could tear down the circuit used by this finished flow if no other existing flow uses this circuit. If there is no new flow that arrives at this time, the current epoch length extends to the completion time of this finished flow minus the start time of current epoch, as shown in Line 15, Alg. 3. On the other hand, if there are new flows arriving before finishing the existing flows, Alg. 3 runs Line 7-13 to compute the epoch length.

The key idea is that if the new flow is short, we decide to reconfigure the circuit when it arrives. Otherwise, we wait to reconfigure circuits after some existing flow finishes. By the end of each epoch, reconfiguration delay is only added to flows using circuits of ports affected by reconfiguration.

Algorithm 3 Calculate Epoch Length θ (Not-All-Stop Model)

1: $t_0 \leftarrow$ The start time of current epoch;
2: $f \leftarrow$ The upcoming flow with the earliest arrival time;
3: $t'(f) \leftarrow$ The arrival time of f;
4: $t''(f) \leftarrow$ The expected completion time of f without reconfiguration;
5: $t_1 \leftarrow$ The earliest completion time among scheduable (unsplittable) flows;
6: **if** $t'(f) \leq t_1$ **then**
7: $\quad tt = t_1 - t'(f)$;
8: $\quad tt' = t''(f) - t'(f) + 2 \times \delta$;
9: $\quad$ **if** $tt' \leq tt$ **then**
10: $\quad\quad \theta = t'(f) - t_0$;
11: $\quad$ **else**
12: $\quad\quad \theta = t_1 - t_0$;
13: $\quad$ **end if**
14: **else**
15: $\quad \theta = t_1 - t_0$;
16: **end if**

6. Evaluation

In order to study the performance of SplitCast in different scenarios and in order to compare our results to the state-of-the-art, we conducted extensive simulations. In the following, we first introduce our setup in §6.1, and then discuss our results in §6.2. We find that SplitCast can significantly improve the performance of multicast transfers across multiple networks and workloads.

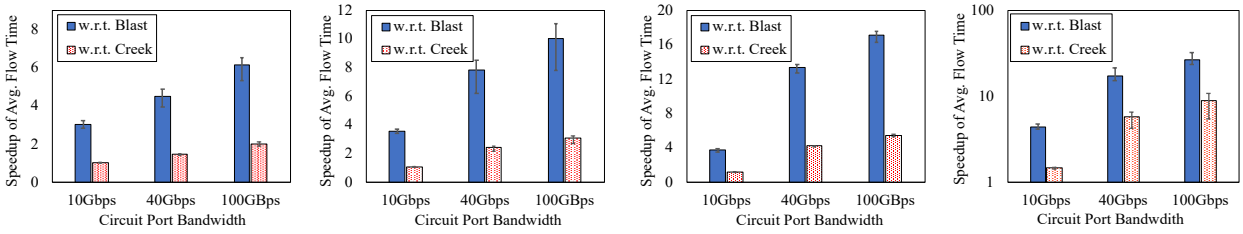
6.1. Setup

To be fair, our evaluation setup largely follows the most related work Blast [25] and Creek [26].

Network topologies: We run simulations over four typical datacenter sizes, with 32, 64, 128, and 256 racks, respectively. According to recent circuit switch designs, we choose the bandwidth per circuit switch port from 10Gbps, 40Gbps, and 100Gbps and let the circuit reconfiguration time range from 0.1ms to 100ms, similar to the evaluation settings in Creek [26, §5.1]. The fanout of per circuit port is 8 by default. The bandwidth between the server and the ToR switch is 10Gbps.

Workloads: We use synthetic non-uniform multicast traffic resulting from real datacenter applications similar to related work [26, 14]. The distribution of data sizes (in GB) per receiver follows a beta distribution B(0.7, 1.7). The sender and the receivers of every multicast are randomly chosen from the racks in the network. To simulate different traffic, we change the number of receiver racks in our experiments and the input load increases as the number of receivers increases. The number of receiver racks follows a uniform distribution $U[2, n\gamma]$, where n is the total number of racks in a network and γ is chosen from $[10\%, 20\%, 30\%]$.

The total simulation time T of every experiment ranges from 5,000ms to 10,000ms and flows uniformly arrive between 1ms and $\frac{T}{10}$ms.



(a) Network with 32 racks. (b) Network with 64 racks (c) Network with 128 racks (d) Network with 256 racks

Figure 6: Impact of the circuit switch port bandwidth. (a)-(d) show the speedup of the average flow time of SplitCast to Blast and to Creek over all runs.

Compared approaches: We compare SplitCast against the state-of-the-art approaches for scheduling multicast demands using a reconfigurable circuit switch.

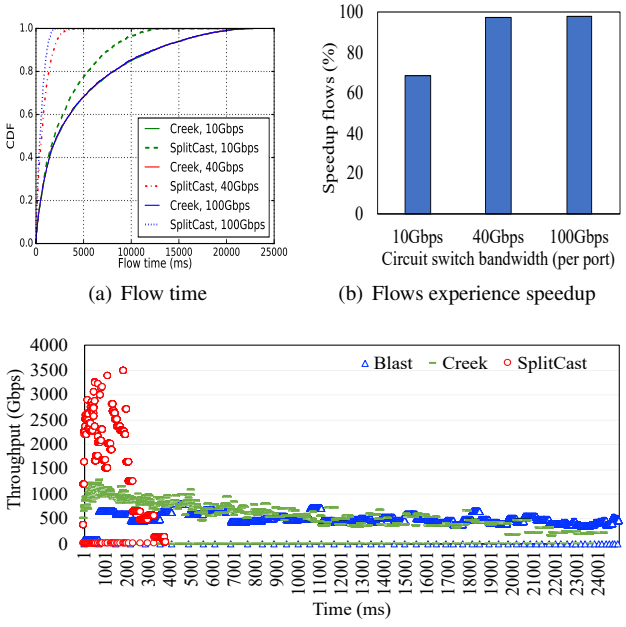
- Blast [25] performs non-preemptive scheduling and iteratively schedules the flows in an decreasing order of a "score" defined by $\frac{size}{\#receivers}$. The latest flow from a set of flows that can be simultaneously transferred in an epoch determines the epoch duration.
- Creek [26] adopts preemptive scheduling, uses the SRPT policy to schedule flows, and chooses the epoch duration that can maximize the circuit switch utilization. We first let Creek use the 1-hop segregated routing model, but will later show how SplitCast compares when Creek may use a multi-hop routing model as well (see Fig. 10).

As in Blast [25] and Creek [26], we focus on the circuit switching network and hence the performance of flows delivered by it.

Performance metrics: We collect the following performance metrics: 1) flow time: the duration between the release time and the completion time of the latest receiver of a multicast flow, 2) throughput: the total transmission rate of all flows at any time, and 3) circuit utilization: the percentage of the total data size transferred to the total data size that can be delivered with full capacity, which reflects how much bandwidth has been utilized. We conduct at least 100 runs for every experiment setting over each network and report the average results below unless otherwise specified.

6.2. Evaluation Results

The impact of circuit port bandwidth. In this group of experiments, we evaluate the impact of circuit port bandwidth by varying it from 10Gbps to 100Gbps. The receiver fraction γ is fixed to 10% and the circuit reconfiguration time is 0.1ms in these experiments. Fig. 6 shows the speedup of the average flow time of SplitCast in comparison to Blast and Creek, in four topologies at different scales. Overall, the results in Fig. 6 show that SplitCast speeds up the average flow time over Blast and Creek and presents a similar trend, the speedup increases with an increasing circuit switch port bandwidth, in these four simulated topologies. Additionally, we can see that the speedup increases as the topology size



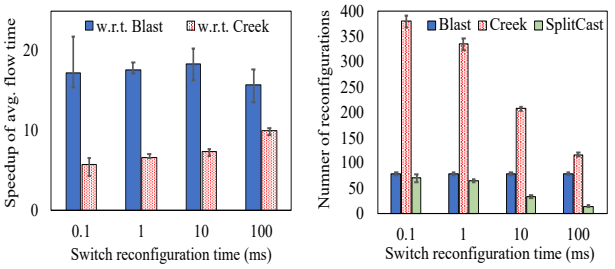
(a) Flow time (b) Flows experience speedup

(c) Throughput over time

Figure 7: (a)-(c) show the CDF of the flow time, the percentage of flows experiencing speedup, and the throughput over time in the 256 rack topology.

grows. In particular, the experiment results for the topology with 256 racks in Fig. 6(d) present the speedup of the average flow time of SplitCast in log scale, from which we can see that SplitCast outperforms Blast and Creek in reducing the average flow time by up to a factor of 27× and 9×, respectively. Thanks to the preemptive scheduling, both Creek and SplitCast outperform Blast. However, Creek cannot beat SplitCast as Creek only transfers data when the circuit connections can match all the receivers of a multicast flow, even through a subset of receivers could be matched. In contrast, SplitCast allows data to be transferred to partially matched receivers, fully using the circuit capacity and achieving the shortest average flow time.

Fig. 7 reports more detailed results obtained from the topology with 256 racks. Fig. 7(a) shows that SplitCast reduces the flow time of the latest flow from around 20s to less than 5s when the circuit port bandwidth is 40Gbps and 100Gbps, and Fig. 7(b) shows that around 68%-98% of the



(a) Speedup of average flow time (b) Number of reconfigurations

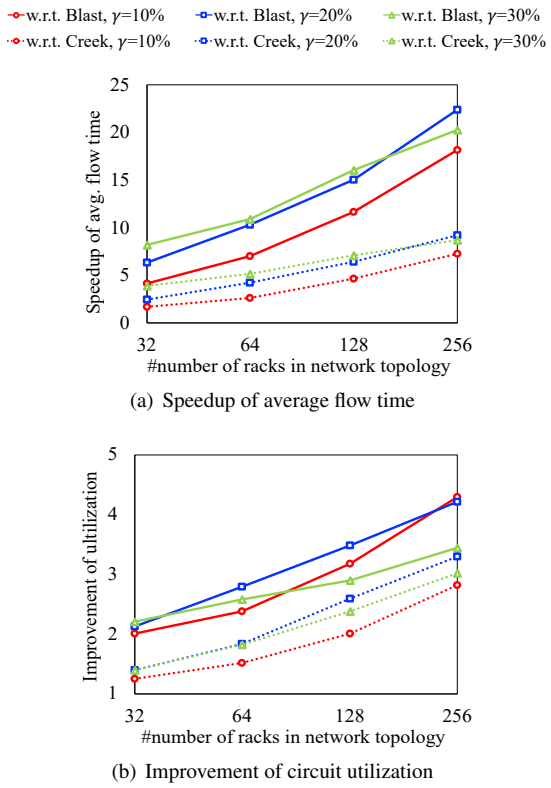
Figure 8: Impact of the reconfiguration time of circuit switch. (a-b) show the maximum-average-minimum speedups of average flow time and the number of reconfigurations of SplitCast in the 256-rack topology. The simulation results show that the improvement of SplitCast in flow time is quite stable for all common switch reconfiguration times.

flows experience speedups, compared to Creek. This is consistent with the results in Fig. 6. Fig. 7(a) also shows that Creek obtains very close flow times under different circuit port bandwidth (10Gbps, 40Gbps, 100Gbps), which indicates that it cannot efficiently use the high-bandwidth circuit capacity. Fig. 7(c) shows the throughput over time (averaged in every epoch) in one experiment simulated with 40Gbps circuit port bandwidth, where each zero point indicates a reconfiguration. SplitCast achieves the highest throughput in the early stage, Creek comes second and Blast follows. The high throughput of SplitCast is also related to its flow time improvements.

The impact of circuit switch reconfiguration time. As the circuit switch will stop transmitting data during circuit reconfiguration, we evaluate how different reconfiguration times impact scheduling approaches in this group of experiments. Fig. 8(a) shows that the speedup of the average flow time of SplitCast over Blast and Creek is stable as the reconfiguration time increases from 0.1ms to 100ms over the 256-rack topology. We omit the presentations of the similar results obtained from other three topologies due to the limited space. Additionally, Fig. 8(b) shows that SplitCast has the minimum number of reconfigurations, Blast comes second, and Creek is last. The very small number of reconfigurations of SplitCast also indicates lower operating cost, compared to Creek and Blast. Additionally, the average number of reconfigurations of SplitCast and that of Creek drop as the reconfiguration time increases. In contrast, the number of reconfigurations of Blast is unadapted to reconfiguration times. This is as expected because SplitCast and Creek consider the reconfiguration time when determining the epoch length, while Blast does not.

The impact of the number of receivers. In this part, we evaluate how the receiver scale impacts the performance of the scheduling approaches. To this end, we set the number of receivers of every multicast to be different percentages γ of the racks and vary γ from 10% to 30%.

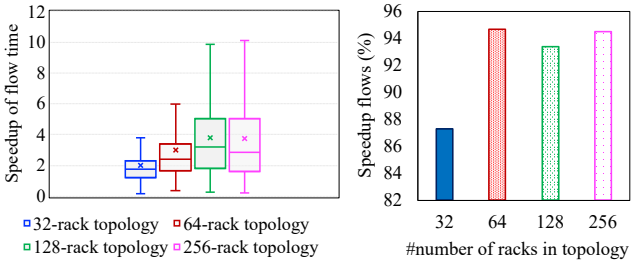
Fig. 9(a) shows the speedups of the average flow time of SplitCast over Blast and Creek. We can see that the speedup



(a) Speedup of average flow time

(b) Improvement of circuit utilization

Figure 9: Impact of the number of receivers. (a) and (b) show the speedup of the average flow time and the improvement of the switch utilization, respectively, in experiments where γ of racks are the receivers of multicast flows.

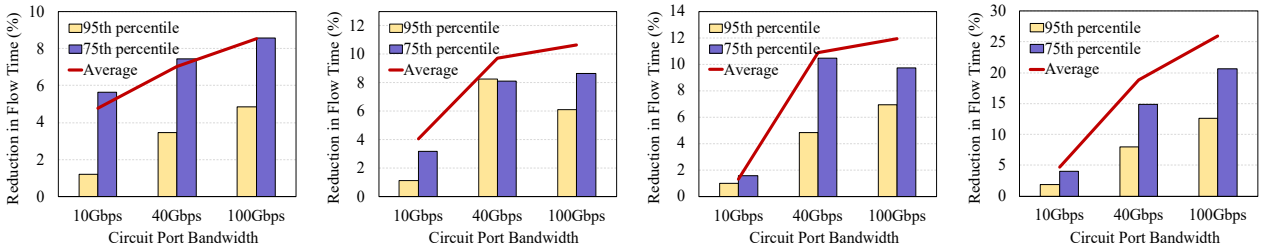


(a) Speedup of flow time (b) Flows experience speedup

Figure 10: SplitCast even outperforms Creek when Creek does m-hop circuit routing, on average speeding up the flow time 2× to 4×.

of the average flow time almost linearly increases with an increasing number of receivers and topology size. In addition, we collect the circuit utilization which is defined as a ratio: the total data size actually transferred, compared to the theoretical total data size that can be delivered with full capacity of the circuit switch for every epoch. We compare the average circuit utilization over all epochs. Fig. 9(b) shows that SplitCast improves the average circuit utilization up to 4.3× and 3× over Blast and Creek, respectively. The improvement of the average circuit utilization increases as the network size scales up.

Additionally, we count the number of subflows created



(a) Network with 32 racks (b) Network with 64 racks (c) Network with 128 racks (d) Network with 256 racks

Figure 11: Impact of circuit capacity. (a)-(d) show the average, 95th percentile, and 75th percentile flow time improvement of SplitCast under not-all-stop model, with $\gamma = 10\%$ of racks being receivers and a reconfiguration time of $1ms$.

by SplitCast for every multicast flow in these experiments. The experimental results show that SplitCast splits only 55% of multicast flows with three receivers on average and creates just one additional subflow for mutlicast flows with an average of three to nine receivers in 32-rack network. Also, SplitCast creates two more subflows for 30%-35% flows and at most four subflows for 80% flows in 256-rack network. The small number of subflows created indicates that the cost of maintaining the state of subflows is low or negligible.

SplitCast vs. Creek with multi-hopping: In the last group of experiments, we evaluate how our 1-hop solution SplitCast compares against the multi-hop version of Creek. Fig. 10 shows that SplitCast can still outperform Creek, speeding up the flow times for around 87% to 94% flows and achieving 2× to 4× speedups on average. These results indicate the great potential of splittable multicast for improving the performance of multicast transfers in reconfigurable datacenter networks. Also, it motivates us to exploit splittable multicast in multi-hop routing enabled reconfigurable networks in future work.

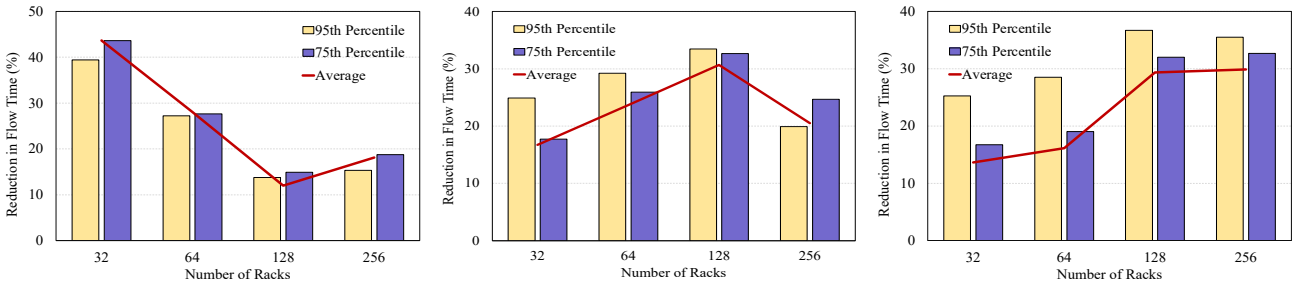
6.3. All-Stop vs. Not-All-Stop Model

We now use simulations to show the further benefits of our new scheduling approach in not-all-stop model, by comparing the performances of SplitCast in the all-stop model and not-all-stop model under a wide spectrum of experiments. To this end, we run simulations under different parameters, circuit port bandwidth, topology scale, reconfiguration time, and the number of receivers. We study the performance gain in terms of the 95th percentile, the 75th percentile, and average flow time obtained by using not-all-stop model. Note that we omit the comparison to Blast and Creek because our scheduling approach has already shown a significant performance improvement over Blast and Creek in the all-stop model. Therefore, we only investigate the further improvement brought by not-all-stop model over the all-stop model in this part of the experiments.

Impact of circuit port bandwidth. We first investigate the performance gain brought by not-all-stop model under different settings of circuit port bandwidth, in the same setting as the experiments in Fig. 6. Besides, the values of other parameters are also consistent with those in the experiments in Fig. 6. Fig. 11 shows the performance gain results in four topologies with different network sizes. As shown in Fig. 11, SplitCast in not-all-stop model reduces both 95th percentile, 75th percentile, and average flow time, and the gains roughly grow as the circuit port bandwidth increases. The results indicate that the performance gains have a positive correlation with the circuit port bandwidth. We can also see from the results that the performance gain increases as the topology size grows. For example, SplitCast in not-all-stop model can further reduce the average flow time by up to 25% in the network with 256 racks compared to the all-stop model. This results suggest that even when the reconfiguration delay is small, *e.g.,* 1ms, network operators who manage large-scale data centers can benefit from using reconfigurable circuit switches that support not-all-stop forwarding model to reap the maximum benefit of reconfigurable network technology in terms of accelerating the transfer of multicast traffic.

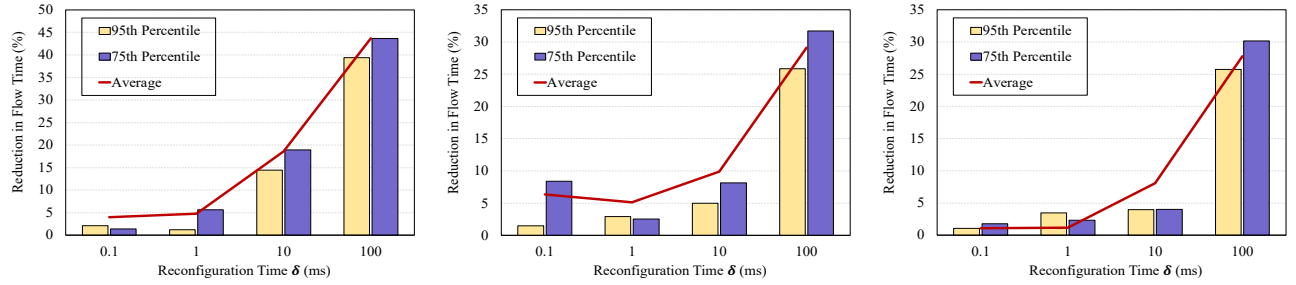
Impact of topology scales. We also investigate the performance gain brought by not-all-stop model under different topology scales. We use 10Gbps, 40Gbps, and 100Gbps circuit port bandwidth, 100ms reconfiguration time, and randomly choose 10% of racks as the receivers of each multicast flow. We observe from the results in Fig. 12(a)-Fig. 12(c) that the trend in performance gain is inconsistent, the gain in flow time reduction is generally higher in larger topologies at 40Gbps and 100Gbps circuit port bandwidth while that is reverse at 10Gbps. Nevertheless, we observe that the gain in flow time reduction is significant, more than 15% and up to (around) 40% in all cases. This results indicate that it is in particular better to use not-all-stop reconfigurable circuit switches in terms of reducing flow times if the reconfiguration delay is not negligible (*100ms*).

Impact of reconfiguration time δ. We also show the performance gain brought by not-all-stop model under different reconfiguration times, ranging from 0.1ms to 100ms, with different number of receivers per multicast flow in a topology with 32 racks. Fig. 13 shows consistent results with the plots in Fig. 12. As the results in Fig. 13 indicate, the performance gain sharply increases as the reconfiguration time becomes longer. This is as expected as not-all-stop model lets traffic flows utilize high-speed circuits with encountering fewer disruptions compared to the all-stop model. Moreover, we observe from Fig. 13(a)-Fig. 13(c)



(a) 10Gbps circuit port bandwidth (b) 40Gbps circuit port bandwidth (c) 100Gbps circuit port bandwidth

Figure 12: Improvement under different topology scales, at 10Gbps, 40Gbps, and 100Gbps circuit port bandwidth, 100ms reconfiguration time, 10% of racks being receivers.

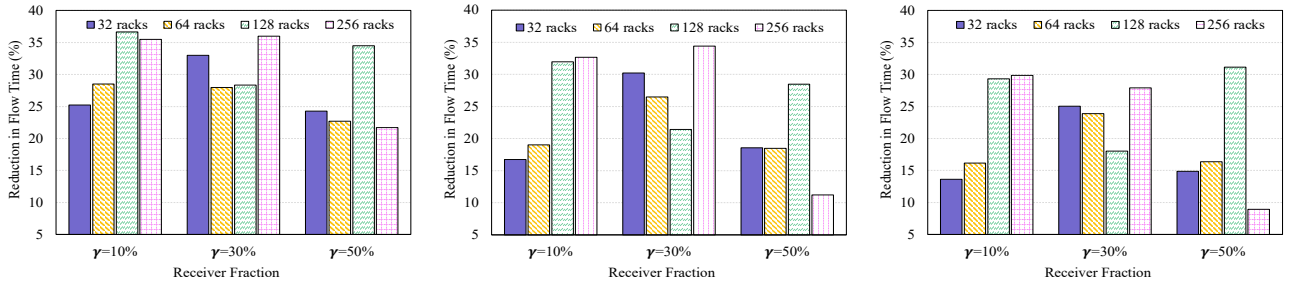


(a) 10% receivers per flow (b) 30% receivers per flow (c) 50% receivers per flow

Figure 13: Improvement under different reconfiguration downtimes, using 10%, 30%, and 50% receivers per flow respectively, in a 32-rack topology with 10Gbps circuit port bandwidth.

that the performance gains are very close for multicast flows with different number of receivers. The reason could be that not-all-stop model can fully use the capacity of unaffected ports during reconfiguration, no matter how many receivers the multicast flows have. Motivated by this results, in term of reducing flow time, we can again recommend network operators deploying reconfigurable circuit switch with not-all-stop forwarding capability when the reconfiguration downtime is long. This results also indicates that the forwarding model (*e.g.*, all-stop or not-all-stop) could be an optimization option for the deployment problem of reconfigurable circuit switches in data centers.

Impact of the number of receivers. At last, we study the performance gain brought by not-all-stop model to multicast flows with different numbers of receivers. The circuit port bandwidth is set to 100Gbps and the reconfiguration time is 100ms. To simulate different numbers of receivers, we vary the receiver fraction from 10% to 50% and rerun experiments under each setting in every evaluated topology. Fig. 14 shows the results. We see that the performance gain first increases and then decreases as the number of receivers increases for every simulated topology, but not-all-stop model retains significant gains all around. This results are a bit different from those in Fig. 13, where



(a) 95th Percentile (b) 75th Percentile (c) Average

Figure 14: Improvement of 95th-percentile, 75th-percentile, and avg. flow time under different numbers of receivers, at 100Gbps circuit port bandwidth and 100ms reconfiguration time.

the performance gain is relative stable when the number of receivers increases. The reason is that the experiments in Fig. 13 use 10Gbps circuit bandwidth ports, which is same as the bandwidth of per server NIC port. This means that each circuit can only be used by a single flow in each epoch and the performance of not-all-stop model is independent with the number of receivers of multicast flows. While, in this part of the experiments, we simulate 100Gbps circuit bandwidth ports, 10× the bandwidth of server NIC port, which means that we can let at most 10 flows share the high-bandwidth circuit in each epoch. Therefore, when the number of receivers increases, *by increasing γ from 10% to 30%*, the network load accordingly increase. SplitCast in not-all-stop model can improve the circuit utilization and achieve a higher reduction in flow times, compared to SplitCast in all-stop model. However, as traffic load continue to increase, *by increasing γ from 30% to 50%*, the network load becomes heavy, the gain of SplitCast in not-all-stop drops a bit because SplitCast can use the circuits well in both forwarding models.

Summary. We see that SplitCast can significantly benefit from our modifications to include not-all-stop model, showing reductions in the flow time of 10% to 30% in many settings, even up to 40% in some scenarios. As the here evaluated settings conform to our previous comparisons of SplitCast to Blast and Creek[1] in the all-stop model, we hence conclude that SplitCast in not-all-stop model widens the performance gap to prior work even further.

7. Conclusion

This paper studied the multicast scheduling problem in datacenter networks that are capable of high-bandwidth circuit switching and fast reconfigurations at runtime. We first discussed the unexploited potential of splittable multicast and analyzed the algorithmic complexity of splittable multicast matching. We proposed a scheduler, SplitCast, which relies on simple single-hop segregated routing and minimizes the flow times by leveraging the potential of splittable multicast, preemptive scheduling, and circuit switch reconfiguration. SplitCast employs a simple but fast algorithm that jointly optimizes both the flows and the circuit configuration schedules. Moreover, SplitCast can also leverage the benefits of both all-stop and not-all-stop models of circuit reconfiguration. Our extensive simulations on real-world topologies show that SplitCast significantly reduces the flow time and improves the throughput over the prior solutions.

We understand our work as a first step and believe that it opens several interesting avenues for future research. For example, it would be interesting to account for more specific application-level objectives, e.g., flow deadlines.

Acknowledgment

This project was funded by the National Key Research and Development Program of China (2019YFB1802800) and the National Natural Science Foundation of China (62102066). This work is being supported by Open Research Projects of Zhejiang Lab (NO. 2022QA0AB02). This project has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 864228, AdjustNet: Self-Adjusting Networks.

[1] Blast and Creek have no extensions for not-all-stop model.

References

[1] Mellette, W.M., McGuinness, R., Roy, A., et al. Rotornet: A scalable, low-complexity, optical datacenter network. In: SIGCOMM. ACM; 2017, p. 267–280.

[2] Mellette, W.M., Das, R., Guo, Y., et al. Expanding across time to deliver bandwidth efficiency and low latency. In: NSDI. 2020, p. 1–18.

[3] Cheng, Q., Bahadori, M., Glick, M., et al. Recent advances in optical technologies for data centers: a review. Optica 2018;5(11):1354–1370.

[4] Valadarsky, A., Shahaf, G., Dinitz, M., Schapira, M.. Xpander: Towards optimal-performance datacenters. In: CoNEXT. ACM; 2016,.

[5] Guo, C., Lu, G., Li, D., et al. BCube: a high performance, server-centric network architecture for modular data centers. SIGCOMM Computer Communication Review 2009;39(4):63–74.

[6] Singla, A., Hong, C.Y., Popa, L., Godfrey, P.B.. Jellyfish: Networking data centers randomly. In: NSDI. 2012, p. 225–238.

[7] Wu, D., Wang, W., et al. Say no to rack boundaries: Towards a reconfigurable pod-centric dcn architecture. In: SOSR. ACM; 2019, p. 112–118.

[8] Wu, D., Sun, X., Xia, Y., et al. Hyperoptics: A high throughput and low latency multicast architecture for datacenters. In: HotCloud. 2016, p. 1–6.

[9] Ballani, H., Costa, P., Behrendt, R., Cletheroe, D., Haller, I., Jozwik, K., et al. Sirius: A flat datacenter network with nanosecond optical switching. In: SIGCOMM. ACM; 2020, p. 782–797.

[10] Ghobadi, M., Mahajan, R., Phanishayee, A., et al. Projector: Agile reconfigurable data center interconnect. In: SIGCOMM. ACM; 2016, p. 216–229.

[11] Foerster, K.T., Pacut, M., Schmid, S.. On the complexity of non-segregated routing in reconfigurable data center architectures. SIGCOMM Comput Commun Rev 2019;49(2):2–8.

[12] Foerster, K.T., Schmid, S.. Survey of reconfigurable data center networks: Enablers, algorithms, complexity. SIGACT News 2019;50(2):62–79. doi:10.1145/3351452.3351464.

[13] Shahbaz, M., Suresh, L., Rexford, J., et al. Elmo: Source-routed multicast for cloud services. In: SIGCOMM. 2019, p. 458–471.

[14] Sun, X.S., Xia, Y., Dzinamarira, S., et al. Republic: Data multicast meets hybrid rack-level interconnections in data center. In: ICNP. IEEE; 2018, p. 77–87.

[15] AlSaeed, Z., Ahmad, I., Hussain, I.. Multicasting in software defined networks: A comprehensive survey. Journal of Network and Computer Applications 2018;104:61–77.

[16] Tseng, H.W., Yang, T.T., Chang, W.C., Lan, Y.X.. An efficient error prevention and recovery scheme for multicast traffic in data center networks. Journal of Network and Computer Applications 2018;114:38–47.

[17] Mai, L., Hong, C., Costa, P.. Optimizing network performance in distributed machine learning. In: HotCloud. USENIX; 2015, p. 1–7.

[18] Deploying secure multicast market data services for financial services environments. https://www.juniper.net/documentation/en_US/release-independent/nce/information-products/pathway-pages/nce/nce-161-deploying-secure-multicast-for-finserv.html; 2021. Accessed: 2021-01-21.

[19] Trading floor architecture. https://www.cisco.com/c/en/us/td/docs/solutions/Verticals/Trading_Floor_Architecture-E.html; 2008. Accessed: 2021-01-21.

[20] VMWARE, . Nsx network virtualization & security software. https://www.vmware.com/products/nsx.html; 2021. Accessed: 2021-01-21.

[21] GOOGLE, . Cloud pub/sub. https://cloud.google.com/pubsub/; 2021. Accessed: 2021-01-21.

[22] Dean, J., Ghemawat, S.. Mapreduce: simplified data processing on large clusters. Communications of the ACM 2008;51(1):107–113.

[23] Li, S., Maddah-Ali, M.A., Avestimehr, A.S.. Coded mapreduce. In: 2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton). IEEE; 2015, p. 964–971.

[24] Cloud Networking: IP Broadcasting and Multicasting in Amazon EC2. https://blogs.oracle.com/ravello/post/cloud-networking-ip-broadcasting-_.and-multicasting-in-amazon-ec2; 2014. Accessed: 2021-01-21.

[25] Xia, Y., Ng, T.E., Sun, X.S.. Blast: Accelerating high-performance data analytics applications by optical multicast. In: INFOCOM. IEEE; 2015, p. 1930–1938.

[26] Sun, X.S., Ng, T.E.. When creek meets river: Exploiting high-bandwidth circuit switch in scheduling multicast data. In: ICNP. IEEE; 2017, p. 1–6.

[27] Zhou, P., He, X., Luo, S., Yu, H., Sun, G.. Jpas: Job-progress-aware flow scheduling for deep learning clusters. Journal of Network and Computer Applications 2020;158:102590.

[28] Lin, Y.D., Lai, Y.C., Teng, H.Y., Liao, C.C., Kao, Y.C.. Scalable multicasting with multiple shared trees in software defined networking. Journal of Network and Computer Applications 2017;78:125–133.

[29] Bao, J., Dong, D., Zhao, B., Luo, Z., Wu, C., Gong, Z.. Flycast: Free-space optics accelerating multicast communications in physical layer. SIGCOMM Comput Commun Rev 2015;45(4):97–98.

[30] Wang, H., Xia, Y., Bergman, K., et al. Rethinking the physical layer of data center networks of the next decade: Using optics to enable efficient*-cast connectivity. SIGCOMM Comput Commun Rev 2013;43(3):52–58.

[31] Chen, L., Chen, K., Zhu, Z., et al. Enabling wide-spread communications on optical fabric with megaswitch. In: NSDI. 2017, p. 577–593.

[32] Lovász, L., Plummer, M.D.. Matching theory; vol. 367. American Mathematical Soc.; 2009.

[33] Ports, D.R., Li, J., Liu, V., et al. Designing distributed systems using approximate synchrony in data center networks. In: NSDI. 2015, p. 43–57.

[34] Apache spark. http://spark.apache.org/; 2021.

[35] Tensorflow. https://www.tensorflow.org/; 2021.

[36] Apache tez. https://tez.apache.org/; 2021.

[37] Vigfusson, Y., Abu-Libdeh, H., Balakrishnan, M., Birman, K., Burgess, R., Chockler, G., et al. Dr. multicast: Rx for data center communication scalability. In: Proceedings of the 5th European conference on Computer systems. ACM; 2010, p. 349–362.

[38] Li, X., Freedman, M.J.. Scaling IP multicast on datacenter topologies. In: CoNEXT. ACM; 2013, p. 61–72.

[39] Luo, S., Yu, H., Li, K., Xing, H.. Efficient file dissemination in data center networks with priority-based adaptive multicast. IEEE J Sel Areas Commun 2020;38(6):1161–1175.

[40] Fan, F., Hu, B., Yeung, K.L., Zhao, M.. Miniforest: Distributed and dynamic multicasting in datacenter networks. IEEE Trans Netw Serv Manag 2019;16(3):1268–1281.

[41] Zhou, X., Zhang, Z., Zhu, Y., et al. Mirror mirror on the ceiling: Flexible wireless links for data centers. SIGCOMM Comput Commun Rev 2012;42(4):443–454.

[42] Yu, Y.J., Chuang, C.C., Lin, H.P., Pang, A.C.. Efficient multicast delivery for wireless data center networks. In: 38th Annual IEEE Conference on Local Computer Networks. IEEE; 2013, p. 228–235.

[43] Shepard, C., Javed, A., Zhong, L.. Control channel design for many-antenna mu-mimo. In: MobiCom. ACM; 2015, p. 578–591.

[44] Huang, X.S., Sun, X.S., Ng, T.E.. Sunflow: Efficient optical circuit scheduling for coflows. In: CoNEXT. 2016, p. 297–311.

[45] Azimi, N.H., Qazi, Z.A., Gupta, H., Sekar, V., Das, S.R., Longtin, J.P., et al. Firefly: a reconfigurable wireless data center fabric using free-space optics. In: SIGCOMM. ACM; 2014, p. 319–330.

[46] Avin, C., Schmid, S.. Renets: Toward statically optimal self-adjusting networks. arXiv preprint arXiv:190403263 2019;:1–32.

[47] Fenz, T., Foerster, K., Schmid, S., Villedieu, A.. Efficient non-segregated routing for reconfigurable demand-aware networks. Comput Commun 2020;164:138–147.

[48] Salman, S., Streiffer, C., Chen, H., Benson, T., Kadav, A.. Deepconf: Automating data center network topologies management with machine learning. In: NetAI@SIGCOMM. ACM; 2018, p. 8–14.

[49] Wang, M., Cui, Y., Xiao, S., et al. Neural network meets dcn: Traffic-driven topology adaptation with deep learning. Proc of the ACM on Measurement and Analysis of Computing Systems 2018;2(2).

[50] Nance Hall, M., Foerster, K., Schmid, S., Durairajan, R.. A survey of reconfigurable optical networks. Opt Switch Netw 2021;41:100621.

[51] Sundararajan, J.K., Deb, S., Médard, M.. Extending the birkhoff-von neumann switching strategy for multicast - on the use of optical splitting in switches. IEEE J Sel Areas Commun 2007;25(S-6):36–50.

[52] Marom, D.M., Colbourne, P.D., D'Errico, A., Fontaine, N.K., Ikuma, Y., Proietti, R., et al. Survey of photonic switching architectures and technologies in support of spatially and spectrally flexible optical networking. J Opt Commun Netw 2017;9(1):1–26.

[53] Jin, X., Li, Y., Wei, D., et al. Optimizing bulk transfers with software-defined optical WAN. In: SIGCOMM. ACM; 2016,.

[54] Jia, S., Jin, X., et al. Competitive analysis for online scheduling in software-defined optical WAN. In: INFOCOM. IEEE; 2017, p. 1–9.

[55] Dinitz, M., Moseley, B.. Scheduling for weighted flow and completion times in reconfigurable networks. In: INFOCOM. IEEE; 2020,.

[56] Gossels, J., Choudhury, G., Rexford, J.. Robust network design for ip/optical backbones. J Opt Commun Netw 2019;11(8):478–490.

[57] Hall, M.N., Liu, G., Durairajan, R., Sekar, V.. Fighting fire with light: Tackling extreme terabit ddos using programmable optics. In: SPIN@SIGCOMM. ACM; 2020, p. 42–48.

[58] Foerster, K.T., Luo, L., Ghobadi, M.. Optflow: A flow-based abstraction for programmable topologies. In: SOSR. ACM; 2020,.

[59] Hall, M.N., Barford, P., Foerster, K., Ghobadi, M., Jensen, W., Durairajan, R.. Are wans ready for optical topology programming? In: OptSys@SIGCOMM. ACM; 2021, p. 28–33.

[60] Durairajan, R., Barford, P., Sommers, J., Willinger, W.. Greyfiber: A system for providing flexible access to wide-area connectivity. arXiv preprint arXiv:180705242 2018;.

[61] Zerwas, J., Poese, I., Schmid, S., Blenk, A.. On the benefits of joint optimization of reconfigurable cdn-isp infrastructure. IEEE Transactions on Network and Service Management 2021;:to appear.

[62] Singh, R., Ghobadi, M., Foerster, K.T., et al. Run, walk, crawl: Towards dynamic link capacities. In: HotNets. ACM; 2017,.

[63] Singh, R., Ghobadi, M., Foerster, K.T., et al. RADWAN: rate adaptive wide area network. In: SIGCOMM. ACM; 2018,.

[64] Luo, L., Foerster, K.T., Schmid, S., Yu, H.. Splitcast: Optimizing multicast flows in reconfigurable datacenter networks. In: INFOCOM. IEEE; 2020, p. 2559–2568.

[65] Garey, M.R., Johnson, D.S.. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman; 1979. ISBN 0-7167-1044-7.

[66] Zaharia, M., Chowdhury, M., et al. Spark: Cluster computing with working sets. In: HotCloud. USENIX; 2010,.

[67] Noormohammadpour, M., Raghavendra, C.S., Kandula, S., Rao, S.. QuickCast: Fast and efficient inter-datacenter transfers using forwarding tree cohorts. In: INFOCOM. IEEE; 2018, p. 225–233.