When Creek Meets River: Exploiting High-Bandwidth Circuit Switch in Scheduling Multicast Data

Xiaoye Steven Sun
Rice University

T. S. Eugene Ng
Rice University

Abstract—**Data multicast is an important data traffic pattern in today's data center running big data oriented applications. The physical layer multicast capability enabled by the emerging technologies used to build circuit switches exhibits huge benefit in transferring multicast data. This paper tackles the problem of scheduling multicast data transfer in high-bandwidth circuit switch. The scheduler aims at minimizing the average demand completion time to deliver the most benefit to the applications. Our algorithm exhibits up to 13.4× improvement comparing with the state-of-the-art solution.**

1. Introduction

Technologies dealing with big data have been significantly improving people's daily life and at the same time bring new challenges in building data center network systems supporting data transfer. Data multicast, or one-to-many data dissemination, is a prevalent traffic pattern in data center handling distributed big data processing. Specifically, data multicast widely exists in various applications ranging from big data analytics, such as iterative machine learning and database queries, to data center infrastructures, such as distributed storage system, virtual machine provisioning, and software maintenance. In these applications, multicast data could have a large volume (from tens of megabytes to several gigabytes) and data multicast happens frequently [1].

Hybrid data center network exhibits great advantages in transferring multicast flows. In a hybrid data center, the top-of-rack (ToR) switches are connected via a high-bandwdith circuit switch in addition to the traditional packet-switched network, as shown in Fig. 1. The circuit switch is able to build directed port-to-port [2], [3], [4] or port-to-multi-port [1], [5] circuit (P2MPC) connections between the ToRs. This capability drastically improves the performance in transferring multicast flows since packets can be delivered to multiple ToRs in a single transmission [1].

In order for the circuit switch to adapt to the current inter-rack traffic demands, a scheduler is a required component because, unlike packet switch, a circuit switch is not able to by itself decide the output ports for the input traffic unless the scheduler configures the circuits. More importantly, the scheduler plays a crucial role in achieving high-performance data multicast because the order and the concurrency in serving the multicast data greatly impact the transfer completion time. Driven by the urgent need of high-performance data multicast and the desirable physical-layer

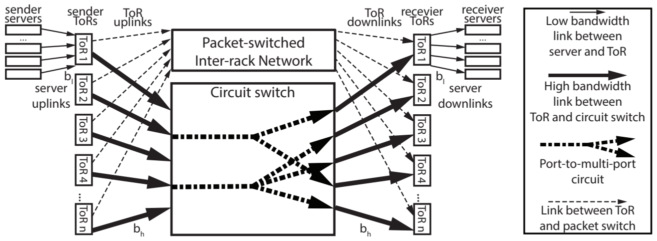

sender servers
sender ToRs
ToR uplinks
...
ToR 1
ToR 2
ToR 3
ToR 4
ToR n
b_l
server uplinks
b_h
Packet-switched Inter-rack Network
Circuit switch
ToR downlinks
recevier ToRs
receiver servers
server downlinks
Low bandwidth link between server and ToR
High bandwidth link between ToR and circuit switch
Port-to-multi-port circuit
Link between ToR and packet switch

Figure 1. Hybrid data center illustration

multicast capability of the circuit switch, this paper deals with the problem of multicast data scheduling in the circuit switch. The design of the scheduling algorithm ought to address the following two challenges.

First, in hybrid data centers, the bandwidth of a circuit switch port is much larger than the bandwidth of a server NIC. This is driven by the following reasons. First, the inter-rack traffic is hungry for bandwidth because it is the aggregation of the traffic from/to tens of servers within the rack. This requires the circuit switch to provide a considerably larger bandwidth that can match the inter-rack traffic demand. Second, connecting a rack and the circuit switch with a high-bandwidth port is much more preferable than using a large number of low-bandwidth ports. Otherwise, it exacerbates the scalability problem of the circuit switch especially for large data centers having hundreds of racks each of which has tens of servers. This is because the number of circuit switch ports is proportional to the number of ToR ports connecting to it. Third, circuit switch is adopted for its capability in carrying high-bandwidth signal, so restricting the circuit switch port bandwidth contradicts the essential merit of using it. Thus, in a realistic hybrid data center, the traffic from a single server cannot fully take up the bandwidth of the circuit switch port. This means that in order to achieve high utilization of the circuit switch bandwidth, the scheduling algorithm must wisely share the bandwidth among the multicast traffic from multiple servers.

Second, improving the performance of applications should be the ultimate objective of the scheduling algorithm. Thus, to deliver the most benefit to applications creating network flows, the scheduling algorithms ought to be "application-aware". That is to say, the scheduling algorithm should consider the traffic demands from individual applications and optimize the time it takes to finish the traffic of each application [6], [7] rather than optimize for the aggregated demands no matter which application a demand belongs to [8], [9]. As previous works [6], [7] suggest, "average demand completion time" is the right metric to

This research was sponsored by the NSF under CNS-1422925, CNS-1305379 and CNS-1718980.

Notation	Definition
$g\in\mathbb{Z}^+$	total number of demands
$n\in\mathbb{Z}^+$	total number of racks
$b_l>0$	bandwidth of the server NIC port
$b_h\geq b_l$	bandwidth of the circuit switch port
$f\in\mathbb{Z}^+$	upper limit on the port-to-multi-port circuit (P2MPC) fanout
$\delta\geq 0$	circuit switch reconfiguration delay
$demand_k$	a multicast demand composed of a tuple of s_k, r_k and d_k
$s_k\in\{1,...,n\}$	sender rack index of Demand k
$r_{kj}\in\{0,1\}$	indicate if Rack j is a receiver of Demand k
$d_k\in\mathbb{Z}^+$	data size of Demand k
$C^t_{ij}\in\{0,1\}$	indicate if a port-to-multi-port circuit (P2MPC) is set up from Rack i to Rack j at Epoch t
$R^t_{ki}\in\{0,1\}$	indicate if Demand k uses Rack i as a relay at Epoch t
$D^t>0$	duration of Epoch t
$T_k\in\mathbb{Z}^+$	epoch index at which Demand k finishes

TABLE 1. NOTATIONS IN THE PAPER. LOWERCASE AND UPPERCASE REPRESENT KNOWN AND UNKNOWN VARIABLES RESPECTIVELY.

evaluate the effectiveness of multicast traffic scheduling on applications' performance. However, how to design a scheduling algorithm that optimizes the average completion time of multicast data demands is an open question. In addition to that, this algorithm should fully exploit the high-bandwidth circuit switch port even though it has much larger bandwidth than the NIC port of a single server.

This paper proposes a scheduling algorithm for multicast data transfer in a high-bandwidth circuit switch. The algorithm adopts multi-hopping and segmented transfer as the approaches to (1) fully utilize the high bandwidth, (2) overcome the fanout limit of P2MPCs and (3) effectively reduce the average completion time. We formulate the scheduling problem and show, using simulation, that our algorithm outperforms the state-of-the-art by up to 13.4×. Our on-going work is to realize an efficient algorithm implementation and to conduct an experimental study in a hardware testbed.

2. Problem and Approaches

This section formulates the scheduling problem and introduces the approaches used in the scheduling algorithm.

2.1. Network Model

We consider a data center (Fig. 1) with n ToRs (notations are in Tbl. 1). These n ToRs connect to a circuit switch with bandwidth b_h at each port (e.g. 40GbE or 100GbE) and a packet-switched network. The circuit switch is able to build directed port-to-multi-port circuits (P2MPCs) between an input port and multiple output ports. The P2MPC divides the physical layer signal to multiple beams with reduced power so the maximum fanout is limited to f whose value depends on the transmission power and the sensitivity of the transceiver. The connections in the circuit switch can be dynamically reconfigured with an overhead of reconfiguration delay δ. We assume that each ToR provides an exclusive port connecting to the circuit switch for multicast data transfer. Each of the ToRs is connected to multiple servers via links each with bandwidth b_l (e.g., 10GbE). We do not require that the ToRs have virtual output queues (VOQs) [10] buffering the outgoing packets. Instead, the scheduling algorithm is based on a more practical and general case where the multicast data is transferred server-to-server.

2.2. Data Multicast Demands

The scheduling algorithm takes a set of g data multicast demands as input. Each of the demands is identified by its

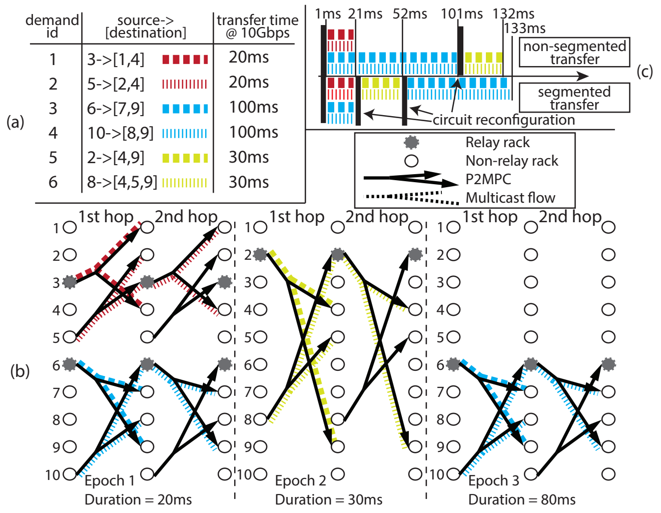


Figure 2. An example of multicast demand scheduling. The example involves ten racks with b_h=20 Gbps, b_l=10 Gbps (a P2MPC can be shared among at most two multicast flows simultaneously) and a circuit switch with δ=1 ms. (a) The input demands. (b) The optimal scheduling with multi-hopping and segmented transfer. The established P2MPCs are shown on all hops. The multicast flow is only marked on the hop it uses. (c) A comparison between segmented and non-segmented transfer (average DCT 412/6 ms vs 508/6 ms).

index k and it includes the sender rack index s_k, the receiver racks represented by an indicator vector $r_k \in \{0,1\}_{[1\times n]}$, and the size of the multicast data d_k. In r_k, the jth element r_{kj} represents whether rack j is the receiver of demand k.

2.3. Scheduling Objective

To speed up the performance of applications, the objective of the scheduling algorithm is to minimize the average demand completion time (average DCT) for the input demands. The DCT of $demand_k$, DCT_k, is defined as the time duration from 1) the time when the demand request is received by the scheduler (demand arrival time, denoted as t^{arr}_k) to 2) the time when the data is received by all the receivers (demand delivery time, denoted as t^{del}_k), i.e., DCT_k=t^{del}_k-t^{arr}_k. Minimizing average DCT is equivalent to minimizing the sum of all the DCTs, i.e., $\sum_{k=1}^{g} DCT_k$.

2.4. The Scheduling Problem and Approaches

In the design of the scheduling algorithm, we adopt two approaches, i.e. multi-hopping and segmented transfer.

Multi-hopping: With multi-hopping, a multicast flow can reach a receiver ToR via other ToRs (called relay ToR) as intermediate hops. This brings the following benefits. First, it enriches the reachability of a ToR because each additional hop allows the source ToR to reach the ToRs connected via the P2MPCs rooted from the relay ToRs. In Fig. 2's example, at Epoch 1 (epoch is defined later), Rack 5 (R5) is able to reach R1 & R4 via the P2MPC rooted from R5 as the first hop and the P2MPC rooted from R3 as the second hop. Thus, in transferring Demand 2 (D2), R3 acts as a relay ToR to enrich the reachability of R5 so as to cover the destination ToRs of D2 (red dotted line). Second, it increases the utilization of the circuit switch bandwidth. With multi-hopping, the bandwidth of the circuit switch port on a ToR can be shared among the multicast flows from the servers in the rack and the multicast flows using the ToR as a relay. For example, in Fig. 2(b) the P2MPC rooted from R3 is shared between D1 & D2. Third, it reduces the occurrence of circuit reconfigurations because P2MPCs can be shared by more demands.

On a receiver ToR of a flow (e.g. R1 of D1 Fig. 2(b)), the ToR is installed with the rules forwarding the flow to the destination servers in that rack. On a relay ToR of a flow (e.g. R3 of D2 Fig. 2(b)), the ToR is installed with the rules forwarding the flow back to circuit switch port. For the flow going to a ToR which is neither the relay nor the receiver ToR (e.g. R1 of D2 Fig. 2(b)), the ToR simply discards the packets of the multicast flow by installing no rule for the flow, so that the packets are discarded by the hardware of the ToR. Thus, discarding the packets does not involve any extra overhead on the switch so it won't slow down the forwarding of any other packets.

Segmented transfer: With segmented transfer, the scheduling may allocate multiple transfer sessions for a single multicast demand rather than require the demand to run to completion in a single session. This is directly beneficial to the minimization of average DCT because small demands are not blocked by large demands. Thus, the scheduler creates multiple *epochs* in scheduling the given demand set (Sec. 2.2). Each epoch includes (1) a fixed circuit configuration ($C^t \in \{0,1\}_{[n\times n]}$), (2) the demands served in the epoch and the set of relay racks of each of the served demand ($R^t \in \{0,1\}_{[g\times n]}$), and (3) the epoch duration (D^t). In Fig. 2's example, D1 & D2 are transferred in Epoch 1. However, D5 & D6 cannot be served in Epoch 1 due to the contention at R4 with D1 & D2. In this situation, D3 & D4 are served simultaneously with D1 & D2 for better circuit switch bandwidth utilization. With segmented transfer, a new epoch can be created at 21 ms (lower part in Fig. 2(c)) to serve D5 & D6. Otherwise, these two demands have to wait until the completion of D3 & D4 (101 ms in Fig. 2(c) upper part), which results in larger average DCT.

With these two approaches, we define the output of the scheduling algorithm as a series of epochs (C^t, R^t and D^t). Denote T_k as the index of the epoch where demand k finishes. Then, the total number of epochs is $\max_k(T_k)$. We assume that the sender transfers multicast flows at the full rate b_l since this helps minimize the flow completion time. To handle the case having multiple flows going to a receiver server, the data can be sent to another server within the same rack and transferred from that server to the receiver. This can be done efficiently since it only involves intra-rack traffic. The constraints to the solution are shown in Equ. 1.

$$\begin{aligned}
\textbf{subject to}:\ &\forall t,\, j, \sum_{i=1}^{n} C_{ij}^t \le 1, && \text{one circuit port per ToR}\\
&\forall t,\, i, \sum_{j=1}^{n} C_{ij}^t \le f, C_{ii}^t = 0 && \text{P2MPC fanout limit}\\
&\forall t, j,\, k, \sum_{i=1}^{n} R_{ki}^t C_{ij}^t \ge r_{kj} R_{ks_k}^t, && \text{cover receivers}\\
&\forall t, j \ne s_k,\, k, \sum_{i=1}^{n} R_{ki}^t C_{ij}^t \ge R_{kj}^t R_{ks_k}^t, && \text{cover relays}\\
&\forall t,\, i, \sum_{k=1}^{g} R_{ki}^t \le \frac{b_h}{b_l}, && \text{bandwidth limit}\\
&\forall k, \sum_{t=1}^{T_k} R_{ks_k}^t D^t \ge \frac{d_k}{b_l}, && \text{data transfer completion}
\end{aligned} \tag{1}$$

The goal of the scheduling can be written as Equ. 2.

$$\begin{aligned}
\textbf{goal}:\ &minimize \sum_{k=1}^{g} DCT_k\\
&= \sum_{k=1}^{g}((\sum_{t=1}^{T_k} D^t) - ((\sum_{t=1}^{T_k} R_{ks_k}^t D^t) - \frac{d_k}{b_l}) + T_k\delta)
\end{aligned} \tag{2}$$

NP-hardness of the problem: We prove by contradiction that the scheduling problem is NP-hard. In a special case of the scheduling problem, each of the multicast demands has only one receiver rack, the data sizes are the same, b_h equals to b_l, and δ is zero. In such special case, the problem is equivalent to the problem of scheduling the unicast flows in a packet switch where a flow exclusive takes the entire bandwidth of the input and output ports when it is transferred. We assume that our scheduling problem can be solved with a polynomial algorithm. Then, such algorithm can also solve the problems in the special case in polynomial time. However, the special case is a *sum coloring problem* [11] which is NP-hard. This contradicts the assumption. Thus, the original problem is also NP-hard.

3. Related Work

Scheduling algorithm: The problem of traffic scheduling in hybrid data centers is being actively investigated in recent years. Previous works propose scheduling algorithms for different types of traffic with various optimization goals. These works either deal with unicast traffic parttern or do not optimize for individual demands. Solstice [8] minimizes the makespan of a batch of unicast traffic demands (a demand matrix) in a hybrid data center. It decides the amount of traffic to be transferred through the circuit switch and the packet-switched network and creates a schedule for the circuit switch traffic. Eclipse [9] maximizes the circuit switch utilization within a given time duration for a unicast demand matrix. However, Solstice [8] and Eclipse [9] do not optimize for the demand from individual applications, instead, they achieve optimization goal for the aggregated traffic demands. To bring impactful performance improvement to applications, Sunflow [7] minimizes the average *coflow* completion time as the improved scheduler beyond Solstice and Eclipse (coflow is defined as a set of unicast demands comes from an appliation [6]). However, Sunflow is designed for unicast flows as well. For multicast scheduling, Blast [1] picks the multicast demands that should be served by the circuit switch and leaves the rest demands to the packet-switched network in order to maximize the number of multicast demands (or the bytes of multicast demands) being served by the circuit switch. However, Blast does not optimize for the demands of individual applications and does not consider exploiting the high-bandwidth circuit switch.

High-bandwidth circuit switch: Composite-path switching [12] in a hybrid data center leverages the high-bandwidth in the circuit switch. A Composite-path is a high-bandwidth link connecting the packet-switched network and the circuit switch. The composite path can be simultaneously shared by multiple unicast flows going to or coming from the same ToR. Our work differs from [12] in that the high-bandwidth link is shared by multicast flows.

Algorithm 1 Skeleton of the scheduling algorithm

```
 1: procedure SCHEDULE($demand_{1,...,g}$, $\delta$)
 2:     $m^{set}$←Set($demand_{1,...,g}$)                      ▷ convert to a set of demands
 3:     $t$← 0                                                ▷ initialize epoch index
 4:     $E^{list}$←List()                                     ▷ initialize the returned list of epochs
 5:     while not $m^{set}$.isEmpty() do
 6:         $C^t_{[n\times n]}$, $R^t_{[g\times n]}$←CreateEpochSchedule($m^{set}$) ▷ Sec. 4.1 & Alg. 2
 7:         $D^t$←DecideEpochDuration($R^t_{[g\times n]}$, $\delta$)          ▷ Sec. 4.2
 8:         $E^t$←Tuple($C^t_{[n\times n]}$, $R^t_{[g\times n]}$, $D^t$)      ▷ get an epoch
 9:         $m^{set}$←Serve($m^{set}$, $E^t$)              ▷ update the remaining demands
10:         $E^{list}$.append($E^t$)                        ▷ add the epoch to the schedule
11:         $t$←$t+1$                                       ▷ increase epoch index
12:     return $E^{list}$
```

Algorithm 2 Create the circuit configuration and choose the demands to be scheduled in an epoch

```
 1: procedure CREATEEPOCHSCHEDULE($m^{set}$)
 2:     $m^{list}$←sort($m^{set}$, key=$\lambda$ $m_k$:$m_k$.remain())
 3:     $G$←HyperGraph()                                     ▷ initialize a hyper-graph
 4:     $M$←List()                          ▷ initialize the list of demands to be served
 5:     for $d$ in $m^{list}$ do
 6:         if $p^{set}$←SolveConflict($G$, $d$, True) then                ▷ Line 16
 7:             DecideToServeDemand($d$, $M$, $p^{set}$)                   ▷ Line 13
 8:     for $d$ in $m^{list}$ do
 9:         if $d$ not in $M$ and $p^{set}$←SolveConflict($G$, $d$, False) then ▷ Line 16
10:             DecideToServeDemand($d$, $M$, $p^{set}$)                   ▷ Line 13
11:     $C$, $R$←ConvertRepresentation($G$, $M$)
12:     return $C$, $R$
13: procedure DECIDETOSERVEDEMAND($d$, $M$, $p^{set}$)
14:     $p$.addDemand($d$) for $p$ in $p^{set}$
15:     $M$.append($d$); $d$.addP2MPC($p^{set}$)
16: procedure SOLVECONFLICT($G$, $d$, $loopfree$)
17:     $cr^{list}$←List([$r$ for $r$ in $d$.rcvrs() if $G$.vertex($r$).indegree==1])
18:     $ncr^{list}$←$d$.rcvrs()-$cr^{list}$
19:     $p_{sndr}$←$G$.vertex($d$.sndr()).P2MPC()
20:     $rts^{set}$←Set()
21:     if $p_{sndr}$ is not None then                                     ▷ Case 1
22:         for $c$ in $cr^{list}$ do
23:             if $G$.isConnected($d$.sndr(), $c$) then
24:                 if not $G$.freeCapacity($d$.sndr(), $c$) then
25:                     return None                                  ▷ demand skipped
26:             else
27:                 $root$←$G$.rootAncestor($c$)
28:                 if not ($root$ and $G$.freeCapacity($root$, $c$)) then
29:                     return None                                  ▷ demand skipped
30:                 $rts^{set}$.add($root$)
31:         $p^{set}_{new}$, $p_{etnd}$←extendP2MPC($p_{sndr}$, $rts^{set}$, $ncr^{list}$, $G$)
32:     else                                                           ▷ Case 2
33:         for $c$ in $cr^{list}$ do
34:             $root$←$G$.rootAncestor($c$)
35:             if not ($root$ and $G$.freeCapacity($root$, $c$)) then
36:                 return None                                      ▷ demand skipped
37:             $rts^{set}$.add($root$)
38:         $p^{set}_{new}$←addP2MPC($p_{sndr}$, $rts^{set}$, $ncr^{list}$, $G$)
39:     if $p^{set}_{new}$ is None then return None                     ▷ demand skipped
40:     if $loopfree$ and adding $p^{set}_{new}$ (and extending $p_{etnd}$) make $G$ cyclic then
41:         return None                                              ▷ demand skipped
42:     $G$.addP2MPC($p^{set}_{new}$, $p_{etnd}$)          ▷ add new hyperedges to the graph
43:     return G.getP2MPCs($d$.sndr(), $d$.rcvrs())  ▷ return the set of P2MPCs the
        demand uses
```

4. Scheduling

We propose a heuristic scheduling algorithm as the solution whose skeleton is shown in Alg. 1. The algorithm works in an iterative manner until all the demands have been completely scheduled (Line 5 (L5)). Each iteration creates an epoch. Creating the circuit configuration (C^t) and picking the demands to be served (R^t) are closely related questions because the demands are served by the circuit. So creating a circuit configuration should consider the senders and the receivers of the demands to be served. Determining the epoch duration (D^t) is relatively independent from determining C^t and R^t. However, given an R^t, as we will show in Sec. 4.2, D^t has a great impact on the effective utilization rate of the circuit switch. Thus, the algorithm first determines C^t and R^t (L6 & Sec. 4.1) and then determines D^t according to R^t (L7 & Sec. 4.2) After the epoch is created (L8), the remaining bytes of the demands are updated so that only the incomplete demands are given to the next iteration (L9). Time complexity of the algorithm is $\mathcal{O}(g^2n(g+n))$.

4.1. Create the Circuit Configuration and Choose the Demands to be Scheduled in an Epoch

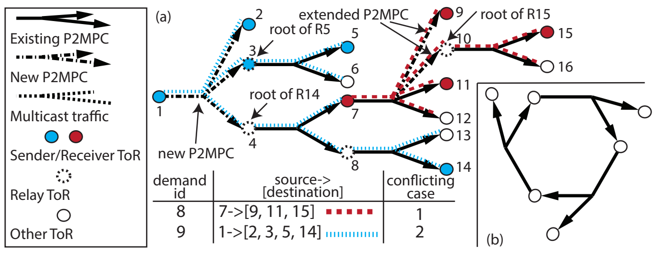
Alg. 2 shows the function that creates the circuit configuration and picks the demands to be served in an epoch. The algorithm iteratively considers the demands in the increasing order of the remaining data bytes (L2); in each iteration, the algorithm may assign the circuit resources to a demand (L7,10). By doing this, the demands with smaller remaining data bytes have higher chances in getting the circuit resources, which is beneficial in minimizing average DCT.

Previous work [1] models the circuit configuration as a hypergraph where the vertices are the racks in the data center and the P2MPCs are directed hyperedges. The hyperedge originates from a single vertex in the hypergraph and points to multiple vertices. In our problem, each hyperedge has a capacity limit of b_h/b_l, which limits the number of multicast demands simultaneously transferred through the P2MPC. In order to serve a multicast demand, the sender vertex must connect to the receiver vertices via hyperedges having free capacities. A demand can be served by multiple cascaded hyperedges since multi-hopping is adopted. Thus, the problem to be addressed in each iteration is to find a set of P2MPCs (hyperedges) that satisfy the connectivity required by the demands and have free capacity. New P2MPCs (hyperedges) may be created if necessary.

In the function, hyperedges are added to hypergraph G in two stages. In the first stage (L5-7), when a new hyperedge is added to the graph, loop is not allowed. That is to say, a demand is skipped by the first stage if serving the demand results in creating a loop in G. This is because having a loop in the graph reduces the chance of sharing the hyperedges in the loop by more demands (explained in following paragraphs and evaluated in Sec. 5.3). After the first stage, the hypergraph forms a forest. In order to increase the utilization of the circuit switch, in the second stage (L8-10), loop is allowed when adding new hyperedges. This is to serve the demands skipped by the first stage.

When considering a demand in either stage, the receiver racks of the demand may already have the output ports of P2MPCs connected. We call these racks as "conflicting racks" which are expected to be connected via a new P2MPC but they have already been occupied by other P2MPCs (L17). For example, in Fig. 3(a), receiver rack R11 & R15 of D8 and receiver rack R5 & R14 of D9 are conflicting racks. Solving the conflicting racks is the fundamental crux in leveraging the high-bandwidth of the circuit switch. Our algorithm solves the conflicting racks by multi-hopping as follows (L16).

There are two cases in solving the conflicting racks. In Case 1 (L21), the sender rack connects to the input of a



demand id	source-> [destination]	conflicting case
8	7->[9, 11, 15] 🟥	1
9	1->[2, 3, 5, 14] 🟦	2

Figure 3. Example of solving conflicting racks. (a) Considering Demand 8 (D8) and D9 during an execution of Alg. 2. The algorithm executes to Iteration i where the P2MPCs rooted from Rack 3 (R3), R4, R7, R8, and R10 are added to the network. In Iteration i+1, D8 is added to the schedule by extending the outputs of the P2MPC rooted from R7 to R9 and R10; in Iteration i+2, D9 is served by adding a new P2MPC rooted from R1 and connecting to R2, R3, and R4. (b) An example of the loop in the hypergraph. All the racks (nodes) can't be reached by any racks (nodes) out of the loop by neither extending existing P2MPCs nor adding new P2MPCs.

P2MPC. In this situation, the sender rack is connected to some of the conflicting racks via existing P2MPCs (L23). In Fig. 3(a)'s example, the sender rack of D8 has connected to the input of a P2MPC, via which the receiver rack R11 can be reached from the sender rack R7. For the conflicting racks that cannot be reached from the sender rack (L26), the algorithm finds the roots of these conflicting racks (L27) and extends the sender's P2MPC to connect these roots as well as the non-conflicting racks (L31). The roots and the racks on the path from the roots to the receiver racks become the relay racks of the demand. In Fig. 3(a), the P2MPC rooted from R7 is extended to R10 (root of conflicting rack R15) and the non-conflicting rack R9. Finding the root of a rack fails if the rack is in a loop. In Fig. 3(b), three P2MPCs form a loop in the hypergraph where all racks cannot be reached from any racks out of the loop by neither adding new P2MPCs nor extending existing P2MPCs since these racks are connected to the outputs of P2MPCs. Thus, once a demand has conflicting racks in loops, the demand cannot be served in the current epoch. This is why the algorithm avoids creating a loop in the first stage when adding P2MPCs.

In Case 2 (L32), the sender rack is not connected to a P2MPC. The algorithm creates a new P2MPC from the sender rack and connects the new P2MPC to all the roots of the conflicting racks as well as the non-conflicting racks (L38). In Fig. 3(a)'s example, when considering D9, a new P2MPC is created and it connects to R4 (root of conflict rack R14), R3 (non-conflicting rack, root of conflict rack R5), and R2 (non-conflicting rack).

When extending the outputs of a P2MPC or adding a new P2MPC, the expected number of outputs may exceed the upper limits f. The algorithm uses the racks not connecting to any P2MPC input/output as relays to expand the reachability of the sender's P2MPC (L31,38).

4.2. Decide the Epoch Duration

Determining epoch duration can greatly affect the performance of the scheduling algorithm because the circuit reconfiguration introduces a non-trivial delay δ and it varies from 10 μs to 100 ms in different circuit switching technologies. Specifically, for the circuit switch having large δ, frequently reconfiguring the circuits results in paying too much overhead due to circuit reconfiguration. On the other hand, for the circuit switch having small δ, after some of the demands finishes in the current epoch, keeping the circuit configuration for extra long time results in a sub-optimal circuit configuration for the remaining demands. Thus, the algorithm should wisely choose the duration of an epoch.

Epoch duration has a direct impact on the *effective circuit switch utilization rate* (denoted as $EU_{(R^t,\delta)}(D^t)$), which is a function of the epoch duration D^t given the demand schedule of the epoch (R^t) and δ. EU represents the effective usage of the circuit switch bandwidth in serving the demands in an epoch and it is defined as Equ. 3.

$$\begin{aligned} EU_{(R^t,\delta)}(D^t) &= \frac{\text{bytes of the demands transfered in } D^t}{\text{bytes can be transfered by circuit switch in } (D^t+\delta)} \\ &= \frac{\Sigma_k(R^t_{ks_k} \times \min(d_k.\text{remain}(), b_l \times D^t) \times \Sigma_j r_{kj})}{b_h \times n \times (D^t+\delta)} \end{aligned} \quad (3)$$

We prove that $EU_{(R^t,\delta)}(D^t)$ is a continuous and weakly unimodal piecewise linear function (we skip the proof due to the space constraint). That is to say, $EU_{(R^t,\delta)}(D^t)$ has a unique extreme value, which is also the maximum effective utilization rate. To maximize the effective utilization rate of the circuit switch, in determining the duration of an epoch, the duration that maximizes EU is chosen. This also greatly helps in reducing the average DCT (evaluated in Sec. 5.3).

5. Evaluation

5.1. Simulation Setup

Multicast demands: The simulation takes input multicast demands synthesized based on the execution of real applications. We run iterative natural language processing algorithms, i.e. Word2Vec and LDA, and a database query benchmark, i.e. TPC-H, on Apache Spark. The sizes of the multicast models in Word2Vec with Wikipedia corpus input and LDA with 20 Newsgroups dataset input are 480 MB and 700 MB respectively. The size of the multicast data created in the execution of the TPC-H benchmark ranges from 24 MB to 5.9 GB when the aggregated database table size is 16 GB. We adopt this empirical distribution in creating the multicast data sizes. This distribution (size in GB) fits well to a beta distribution with α=0.7 and β=1.7. The number of receiver racks is a uniform distribution from 2 to the half of the racks in the data center. The sender and receiver racks are uniformly distributed among all the racks. We use *traffic intensity* to quantify the total amount of input multicast demands given to the scheduling algorithm. The *traffic intensity* is defined as $\Sigma_k(d_k \times \Sigma_j r_{kj})$ divided by $b_l \times n$, which shows the finishing time when all the multicast traffic is sent at the aggregated circuit switch bandwidth when b_h=b_l. In the simulation, the *traffic intensity* of the input multicast demands ranges from 1 s, 5 s and 10 s.

Circuit switch properties: In the simulation, the number of racks ranges from 32 to 256, which are typical data center sizes and cover the network scales evaluated in all the related works. The server NIC bandwidth is 10 Gbps and the circuit switch port bandwidth are set to 10 Gbps, 40 Gbps and 100 Gbps. δ ranges from 10 μs to 100 ms, which covers all the recently proposed circuit switch designs. The limit on

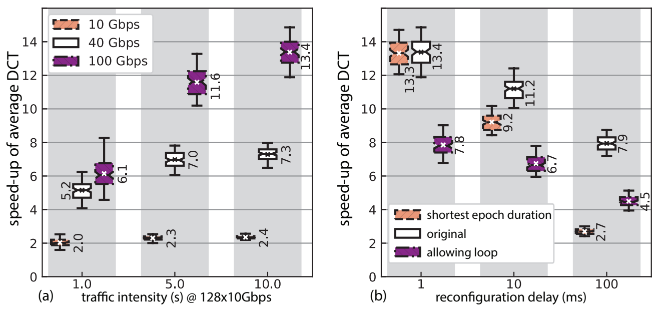


Figure 4. Box plot of the speed-up of average DCT using Blast as the baseline. The center quartile and the 'x'(and numbers) are median and average respectively; the upper(lower) quartiles and the upper(lower) whiskers shows 75(25)- and 95(5)-percentile respectively. Each box represents a hundred sets of input demands in the setting with n=128, b_l=10 Gbps. (a) Various circuit switch port bandwidth and traffic intensity. δ=1 ms. (b) Various δ and algorithm policies. b_h=100 Gbps, traffic intensity=10 s.

P2MPC fanout, f, is set to 16, which can be easily handled by the sensitivity of today's optical transceivers.

Comparison baseline: We compare our scheduling algorithm against the algorithm proposed in Blast [1]. Blast schedules the multicast demands in hybrid data centers and shows 37× better performance comparing against overlay peer-to-peer multicast. When scheduling demands in the circuit switch, Blast models the problem as a maximum weighted hypergraph matching problem where the weight of an edge is the data size. The algorithm proposed in Blast first sorts the demands in a decreasing order of $d_k/\Sigma_j r_{kj}$, and checks if the demands can be served based on this order. In our simulation, the algorithm runs multiple rounds until all the demands are scheduled. Multi-hopping and segmented transfer are not considered in Blast.

5.2. Greatly Improved Average DCT

We have analyzed the results of all the settings listed in Sec. 5.1. We found that, for different numbers of racks, our solution shows similar trends. For the settings having $\delta \leq 1$ ms, the results are similar as well since in these cases, δ is negligible compared with the time to transfer the data. Due to space limitation, we present the results of the settings having 128 racks and $\delta \geq 1$ ms as representative cases. Fig. 4(a) presents the box plot of the speed-ups in average DCT. Each box plot corresponds to 100 data points in a setting. For each data point, the speed-up is defined as the average DCT of Blast divided by that of our solution. So the higher the speed-up is, the larger improvement shown by our solution. We summarize our observation as follows.

Our solution exhibits up to 13.4× speed-up comparing against the algorithm in Blast. The improvement comes from two aspects. The first aspect is the order in which the demands are considered. In each iteration, our solution starts picking the demand having the smallest remaining size, which is beneficial in reducing average DCT. The improvement can be seen in the cases where b_h=b_l=10 Gbps. The speed-up is about 2.0×. The second aspect is that our solution is capable of leveraging the high-bandwidth of the circuit switch port, which can be demonstrated by the following observations. (1) Given the same input demands (fixed traffic intensity), as b_h increases, the speed-up increases significantly, e.g., in the case with 10s traffic intensity, the speed-up increases from 2.4× to 13.4× as b_h increases from 10 Gbps to 100 Gbps. (2) Given the same b_h, as the traffic intensity increases, the speed-up increases as well, e.g., in the case having 100 Gbps circuit switch port, the speed-up increases from 6.1× to 13.4×.

5.3. Effective Algorithm Features

Our algorithm design is carefully considered, which is shown by comparing our solution against an algorithm having some algorithm features turned off.

Requiring loop-freedom in adding P2MPCs in the first stage significantly improves the speed-up. We compare our algorithm against a similar algorithm whose only difference is that loops are allowed in adding P2MPCs to the graph in the first stage. Fig. 4(b) shows that the loop-freedom requirement exhibits more than 67% increase in speed-up. This is because that maintaining P2MPCs as a forest increases the chance of solving conflicting racks, which effectively reduces average DCT.

Maximizing circuit switch effective utilization rate significantly improves speed-up when δ is large. We compare our solution against a similar algorithm whose only difference is that the epoch duration is always the time used to finish the demand with the smallest remaining size. As δ increases from 1 ms to 100 ms (Fig. 4(b)), maximizing EU exhibits increasing benefits. This is because with large δ, frequently reconfiguring the circuit makes δ dominate the DCT. Maximizing EU effectively helps to reduce the number of reconfigurations so as to minimize average DCT.

6. Conclusion

We propose an algorithm scheduling the multicast demands in a high-bandwidth circuit switch capable of building P2MPC connections. We adopt multi-hopping and segmented transfer as the approaches. The algorithm effectively leverages the high-bandwidth of the circuit switch ports and minimizes the average DCT of the multicast demands. Our solution exhibits significant improvement comparing against the state-of-the-art scheduling algorithm.

References

[1] Y. Xia *et al.*, "Blast: Accelerating high-performance data analytics applications by optical multicast," in *INFOCOM'15*, April 2015.
[2] G. Wang *et al.*, "c-through: Part-time optics in data centers," in *SIGCOMM'10*, 2010.
[3] N. Hamedazimi *et al.*, "Firefly: A reconfigurable wireless data center fabric using free-space optics," in *SIGCOMM'14*, 2014.
[4] X. Zhou *et al.*, "Mirror mirror on the ceiling: Flexible wireless links for data centers," in *SIGCOMM'12*, 2012.
[5] J. Bao *et al.*, "Flycast: Free-space optics accelerating multicast communications in physical layer," in *SIGCOMM'15*, 2015.
[6] M. Chowdhury *et al.*, "Efficient coflow scheduling with varys," in *SIGCOMM'14*, 2014.
[7] X. S. Huang *et al.*, "Sunflow: Efficient optical circuit scheduling for coflows," in *CoNEXT'16*, 2016.
[8] H. Liu *et al.*, "Scheduling techniques for hybrid circuit/packet networks," in *CoNEXT'15*, 2015.
[9] S. Bojja Venkatakrishnan *et al.*, "Costly circuits, submodular schedules and approximate carathéodory theorems," in *SIGMETRICS'16*, 2016.
[10] B. Prabhakar *et al.*, "On the speedup required for combined input- and output-queued switching," *Automatica*, vol. 35, pp. 1909 – 1920, 1999.
[11] A. Bar-Noy *et al.*, "Sum multicoloring of graphs," *J. Algorithms*, vol. 37, pp. 422–450, Nov. 2000.
[12] S. Vargaftik *et al.*, "Composite-path switching," in *CoNEXT'16*, 2016.