Near-Optimal Multihop Scheduling in General Circuit-Switched Networks

Himanshu Gupta
Stony Brook University, NY
hgupta@cs.stonybrook.edu

Max Curran
Stony Brook University, NY
mcurran@cs.stonybrook.edu

Caitao Zhan
Stony Brook University, NY
cbzhan@cs.stonybrook.edu

ABSTRACT

Circuit switched networks with high-bandwidth links are essential to handling ever increasing traffic demands in today's data centers. As these networks incur a non-trivial reconfiguration delay, they are mainly suited for bursty traffic or large flows. To address the reconfiguration delay vs. high-bandwidth tradeoff in circuit networks, an essential traffic scheduling problem is to determine a sequence of network configurations to optimally serve a given traffic. Recent works have addressed this scheduling problem for one-hop traffic in fully-connected circuit networks.

In this work, we consider the traffic scheduling problem in general circuit networks with multi-hop traffic load. Such a general model is essential for networks with indirect routes between some nodes, e.g., for recently proposed wireless optical (FSO-based) networks, or to allow multi-hop routes for load balancing. In this context, we develop an efficient algorithm that empirically delivers high network throughput, while also guaranteeing a constant-factor approximation with respect to an objective closely related to network throughput. We generalize our technique and approximation result to more general settings, including to the joint optimization problem of determining flow routes as well as a sequence of network configurations. We demonstrate the effectiveness of our techniques via extensive simulations on synthetic traffic loads based on published traffic characteristics as well as publicly available real traffic loads; we observe significant performance gains in terms of network throughput when compared to approaches *based* on prior work, and very similar performance to an appropriate upper bound.

CCS CONCEPTS

• **Networks** → **Traffic engineering algorithms**; **Network control algorithms**;

KEYWORDS

reconfiguration networks, matching, approximation algorithms

ACM Reference Format:
Himanshu Gupta, Max Curran, and Caitao Zhan. 2020. Near-Optimal Multihop Scheduling in General Circuit-Switched Networks. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3386367.3432589

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3432589

1 INTRODUCTION

Modern data centers support ever-increasing computation and storage demand, as applications shift to the cloud. As inter-traffic of data centers doubles every 12-15 months [33], there is a demand for network architectures and switches that can handle ever increasing traffic bandwidth. Packet-switched networks with electronic (packet) switches offering a typical bandwidth of 10 or 40 Gbps don't seem enough to handle increasingly bandwidth-hungry data centers. Thus, recent works have considered circuit-switched networks based on optical [24, 25, 36, 38] or wireless [18, 19, 21, 40] links that connect optical links using circuit switches.

Circuit-switched architectures enable a dynamic topology tuned to prevailing traffic patterns, and can provide high bandwidth (100-200 Gbps [8]) at a lower cost, power, and reduced cabling complexity [14]. However, circuit switches incur a reconfiguration delay which could range from tens of microseconds to milliseconds [30], during which they cannot carry any traffic. The reason for reconfiguration delay differs across different circuit-switching technologies: delay in 60 GHz wireless [19, 40] links is caused by rotation of RF antennas, while in FSO-based networks [21] and optical circuit-switches [30] delay is caused by steering mirrors. In either case, the high-bandwidth with reconfiguration delay combination makes the circuit-switched networks particularly suitable for serving large flows or bursts of traffic, which is a typical environment in data centers [18]. On the other hand, packet switches are flexible, capable of making forwarding decisions at the granularity of individual packets—making them suitable for sporadic traffic and latency sensitive packets. Thus, it is natural to design hybrid circuit/packet switch architectures [24, 29] wherein long bursty flows use circuit-switched network while short flows use the packet switched networks.

In such hybrid or circuit switched networks, a fundamental traffic scheduling problem arises that must be addressed: given a traffic load, determine the sequence of circuit configurations in order to maximize the traffic delivered via the circuit-switched network. This circuit-switched traffic scheduling problem has been addressed recently [25, 28, 36], but only in limited settings. In particular, prior work considers only direct (single-hop) routing of traffic and thus implicitly assumes a *complete* circuit-switched network; however, in general, circuit network fabrics may not have a complete topology and thus multi-hop routing may be unavoidable, e.g., in recently proposed wireless optical architectures [18, 21]. Moreover, indirect routing is also useful for load balancing [16, 28]. See §3 for further discussion.

In this work, we address the above limitations by considering a general model of circuit-switched networks—as a general (i.e., not necessarily complete) bipartite graph over input/output ports of network nodes. In this circuit network model, we consider the problem of scheduling and routing general *multi-hop* traffic load. Our focus in this paper is to develop efficient algorithms that deliver high network throughput in practice, while also ensuring theoretical performance guarantees. However, performance guarantees in terms of network throughput are particularly challenging in the multi-hop setting. In this paper, we design algorithms that not only yield high throughput empirically but also permit performance guarantees in a *closely related* objective (total number of weighted packet-hops traversed); such performance guarantees lend further credence to its high empirical performance. In the above context, we make the following contributions in this paper.

1. For the traffic scheduling problem in general circuit networks with multi-hop traffic and given flow routes, we develop an efficient algorithm to generate a sequence of network configurations that empirically delivers high network throughput as well as yields a constant-factor approximation guarantee in a closely related objective (§4-5).
2. For the more general joint-optimization problem to select flow routes as well as determine the sequence of network configurations, we design a greedy-cum-backtracking algorithm which also delivers a constant-factor approximate solution for the special-case of two given routes choices per flow (§6).
3. We generalize our technique and approximation results to more general network models such as networks with bidirectional links or with multiple input and output ports per node (§7).
4. Over extensive simulations, we show that our developed techniques outperform an approach based on a prior work by a significant margin. More importantly, our techniques perform close to an appropriate upper bound.

To the best of our knowledge, the multi-hop traffic scheduling problem addressed in our paper has not been addressed before ([28, 36] do consider multi-hop traffic in circuit networks, but in very limited settings, and [18, 21] consider localized reconfigurations; see §2). Our work essentially addresses and solves the open problem mentioned in [36] and [25].

2 RELATED WORK

Zero or Moderate Reconfiguration Delay. Scheduling in crossbar switches is a well-studied problem, with traditional models assuming zero or very small reconfiguration delay. E.g., prior works have considered Birkhoff-von-Neumann decomposition problem [9] of decomposing a given bipartite graph into matchings. Some other works assume moderate configuration delay in that they allow $O(n)$ configurations, where n is the number of nodes/flows, and with that constraint, focus on other optimization objectives such as higher link utilization [34], bounded packet delay [39]. Some works have also addressed minimizing the number of reconfigurations [22] without regard to the total time window.

One-Hop Traffic Load. For arbitrary reconfiguration delay, recently some works have modeled the circuit-switched network as a single $n \times n$ crossbar switch, and considered scheduling of a one-hop traffic load. In this context, [23, 25] design heuristics for the problem of minimizing the total evacuation time for the given traffic load, and present heuristics. Among these, ADJUST [23] does not benefit from traffic matrix sparsity and thus still requires around $O(n)$ configurations, while Solstice [25] designs a greedy heuristic based on Birkhoff-von-Neumann decomposition for scheduling in hybrid networks. In the most related work, [36] designs an approximation algorithm for the problem of maximizing the traffic throughput given a time window W. Our work is inspired by and is a direct generalization of [36] in that we consider the same problem as [36] but with multi-hop traffic load in a general circuit-switched model. Finally, [37] consider online adaptive scheduling policies for one-hop traffic in circuit networks and develop hysteresis-based online policies with derived conditions under which they maintain bounded queues. These online policies however require perfect queue state information at every instant.

Multi-hop Traffic Load. To the best of our knowledge, the circuit-network scheduling problem addressed in our paper for multi-hop traffic has not been addressed before. The closest works are: (i) [36] addresses *routing* a given multi-hop traffic load over a ***given*** sequence of configurations; (ii) Recently, [27, 28] have designed traffic-agnostic schedulers for multi-hop traffic for their very specialized circuit-switched networks; we compare our work to theirs in §8. (iii) Works on FSO-based networks [18, 21] do address engineering multi-hop traffic over a reconfigurable network, but their schemes use *localized* reconfigurations (rather than global configurations, as in our model) for online-arriving flows and have different objectives (e.g., minimizing latency [18]).

3 NETWORK MODEL

We consider a data center network over n network nodes, with each node having a certain number of input as well as output port. A node may represent a rack of servers, in which case the network ports correspond to ports on the Top-of-Rack switches, or a node may represent an individual server with ports. Our network model is a hybrid network fabric which is a combination of a high-bandwidth circuit-switched (typically, optical) network and a low-bandwidth (e.g., an order of magnitude lower) packet-switched electrical network. Each network node is connected to both networks via ports. Our focus in this work is on determining traffic schedules for the high-bandwidth circuit-switched network, though our circuit network scheduling technique can easily be used to develop an efficient scheme for the overall hybrid network (§7).

Our Network and Traffic Model. Prior works [25, 36] that consider scheduling in circuit-switched networks have largely focussed on single-hop traffic, and thus have implicitly assumed a network with *complete* topology, e.g., networks with a *single* $n \times n$ crossbar switch connecting the input and output ports. However, in general, circuit switched network fabrics may not have a complete topology; e.g., (i) FSO-based networks [18, 20, 21], where it may be infeasible to have a complete topology, (ii) circuit network fabrics formed of multiple optical switches [26–28]; note that its infeasible to connect large data centers with a single optical switch due to their low port count [8]. In such networks, indirect (multi-hop) routing is unavoidable. In addition, multi-hop routing may also be useful for

load balancing [35] (especially for skewed traffic [28]) and/or to increase reachability with a smaller number of configurations [36].

To address the above limitations, in this work, we take a general approach and model the circuit-switched network as a *general (not necessarily complete) bipartite graph* over the input and output ports, and consider general multi-hop traffic load (see below). More formally, we represent the circuit network as a bipartite $n \times n$ graph G over the n network nodes, where an edge (i, j) signifies a potential link from the (output port of) node i to the (input port of) node j. In the above model, each link in G is thus implicitly a uni-directional link; we consider bidirectional links and more general graphs in §7.

Time Slots. As in most prior works [25, 36], we divide the time into *slots*, with each slot corresponding to a packet transmission time on the circuit switch. In each time slot, a packet may be transmitted over each active link. This assumption of dividing times into time slots allows us to abstract the scheduling problem sufficiently to design algorithms with provable performance guarantees.

Configurations, Reconfiguration Delay (Δ). In any time slot, only a set of links that form a matching of G can be active, since within a circuit network, each input/output port can have at most one active connection (we allow multiple ports/node in §7). A network *configuration* is denoted as (M, α), where M is a matching in G and α is the number of time slots for which the set of links in M are active. To change the set of active links, the circuit network must be reconfigured completely;[1] this incurs a *reconfiguration delay* which can be of the order of tens of microseconds [24, 30] to a few milliseconds. We represent this delay in terms of Δ number of time slots. In addition, we use the notation $\langle, \ldots, \rangle$ to represent a *sequence* of configurations.

Traffic Load. We represent the *traffic load* between the nodes as a set of traffic flows, where each *flow* is represented by:

$$\langle \texttt{ID}, \texttt{size}, \texttt{source}, \texttt{destination}, \texttt{routes} \rangle.$$

Above, the size parameter is in number of packets, and the `routes` is a *set* of potential routes (to choose from) between the source and destination with each route represented as sequence of nodes $\langle \text{source}, x_1, x_2, \ldots, x_l, \text{destination} \rangle$ such that for each i, output port of x_i is connected to the input port of x_{i+1} in the network graph G. Also, we use $\mathcal{D}$ to denote the maximum length of any flow route. Diameter of most realistic networks is small (2-4); thus, we can assume $\mathcal{D}$ to be equally small.

Controller and VOQs. We assume a centralized (possibly, multi-core) controller which has access to the traffic load, on the basis of which the schedule of network configurations is determined. The packets at each output port are organized into virtual-output-queues [31] (VOQs) which hold packets destined to different ports. Within each VOQ, packets may be prioritized based on certain packet parameters, such as flow ID, packet weight, etc. At each network node, output port(s) are connected to the input port(s)—thus, an intermediate node can channel the traffic received on its input port(s) to the appropriate VOQs of its output port(s), for transmission in subsequent configurations or time slots.

[1] FSO-based networks do allow "local" reconfiguration. Traffic scheduling problem in such circuit networks with local reconfigurations is deferred to our future work.

3.1 Multi-Hop Scheduling (MHS) Problem

We now define the addressed MHS problem. We start with an informal formulation and discuss the multi-hop challenges.

Informal Formulation and Multi-Hop Challenges. Given a circuit-switched network G and traffic T, our traffic scheduling problem (denoted as MHS) is to determine a sequence of configurations that is able to "serve" most of the given traffic within the given time window. For the special case of one-hop traffic load T, [36] gives a greedy approximation algorithm `Eclipse` that maximizes the number of packets *delivered*, and leaves the multi-hop traffic case as an open problem. Designing an efficient algorithm for the multi-hop case is challenging, because the objective of maximizing the network throughput (i.e., the number of packets delivered) does not remain "submodular" (see §5), and hence, a greedy algorithm can perform arbitrarily bad. In addition, the multi-hop generalization makes selecting the best configuration at each greedy iteration becomes challenging due to many multi-hop aspects, e.g., due to non-uniform route lengths and the fact that a packet can traverse multiple hops in a single configuration. Finally, a multi-hop traffic scheduler also entails choosing the route for each flow.

MHS **Problem.** Consider a set N of n nodes, and a circuit-switched network over N represented by a bipartite graph $G = (N, N)$ between the given set of nodes, with a reconfiguration delay of Δ time slots. We are given a traffic load T and a window of W time slots. The MHS problem is to determine a sequence of configurations (M_1, α_1), (M_2, α_2), ... and routing of given packets through them, such that the *network throughput* (the number of packets delivered to their final destination) is maximized, under the constraint that

$$\sum_k (\alpha_k + \Delta) \leq W.$$

The above MHS problem is easily NP-hard, as the special case of one-hop traffic load is known to be NP-hard [36]. We address more general network graphs (e.g., multiple ports per node) in §7. The below example illustrates traversal of packets and network throughput for a given sequence of configurations; these terms get formalized in §4.

Example 1: Packet Traversal Through Configurations; Network Throughput. Consider the example in Figure 1, which shows a graph G, traffic T, and a sequence of configurations (not necessarily optimal) for a total W of 300 (here, we assume a reconfiguration delay of 0). Here, we assume a single route for each flow. Thus, each entry in T gives the flow ID, number of packets, and the flow route, with the row and column representing the source and destination respectively. Note that after the first configuration, there are 150 packets "situated" at node a waiting to go to node b, of which 100 packets are chosen to be routed in the second configuration; this choice of packets can be arbitrary, but, in this example, let us prioritize the packets by flow ID. Thus, as the (a, c)-flow has the lower ID, the second configuration routes the 100 (a, c)-flow packets. The overall solution can now be seen as routing the entire (d, b)-flow via first and fifth configurations, the entire (c, a)-flow via the third and fourth configurations, and routing the 100-packets of (a, c)-flow to the intermediate node b via the second configuration. Thus, while the total number of delivered packets is 100. Note that the optimal solution (not shown) is the sequence of configurations:

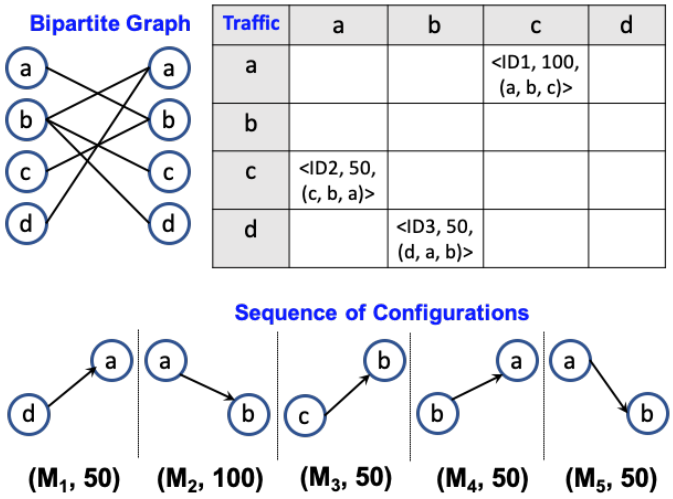

Figure 1: MHS Example.

$(M_1 \cup M_3, 50)$, $(M_4 \cup M_5, 50)$, $(M_2, 100)$, $((b, c), 100)$, which delivers all the packets to their destinations.

4 OCTOPUS APPROXIMATION ALGORITHM

In this section, we design an efficient algorithm, called Octopus, for the MHS problem. At a high-level, Octopus implicitly strives to optimize an objective that is closely related to network throughput. As mentioned before, such a strategy facilitates developing techniques that deliver high network throughput in practice, while also yielding performance guarantees in a closely related objective and thus confirming its high empirical performance. We first state two assumptions made temporarily for clarity of presentation.

Two Assumptions (Relaxed Later). For sake of *clarity of presentation*, we make two assumptions for now and relax them later: **(i)** Each individual packet traverses at most one hop during a configuration; we relax this assumption in §5. **(ii)** For each traffic flow, there is only one route available. Single routes represents the case when either the route is unique or has been independently determined via standard traffic engineering techniques. We relax this assumption in §6.

Optimization Objective ψ. Recall that the objective of our MHS problem is to maximize the *throughput*, i.e., the number of packets delivered. However, this objective is not submodular for multi-hop traffic load, which makes design a provably efficient algorithm challenging. To circumvent this challenge, we consider an optimization objective that is "monotonic," "submodular," and, equally importantly, also results in high network throughput. Keeping the above in mind, we consider the optimization objective of maximizing the *total number of hops traversed* by the packets in the given traffic load T. Also, since each flow route may have a different number of hops, we assign a **weight** to each packet equal to the inverse of the total number of hops in its flow route. *Note that this chosen objective is merely to guide design of an efficient algorithm, and our evaluation metric is still network throughput.*

More formally, let $\mathbb{S}$ be a sequence of configuration, and for each packet p, let $f(p, \mathbb{S})$ be the number of hops traversed by p in $\mathbb{S}$ and w_p be the weight of p as defined above. Then, the objective value of the solution $\mathbb{S}$ is denoted by $\psi(\mathbb{S})$ and is given by:

$$\psi(\mathbb{S}) = \sum_{p \in T} f(p, \mathbb{S}) w_p. \quad (1)$$

We discuss computation of the function f later.

Relationship to Throughput. It is easy to see that the objective function ψ does maximize the throughput, **if** we assume that the final solution leaves zero packets at intermediate nodes—because then, $f(p, \mathbb{S})$ is either 0 or $1/w_p$ for each packet p. In §8, we observe that the number of undelivered packets is indeed small for expected traffic loads, and explore a simple strategy to minimize it further. More importantly, packets undelivered after one application of the algorithm can be considered for continued routing in the next time window; thus, undelivered packets do not result in packet losses.

Packet Prioritizing Scheme. Our proposed Octopus algorithm uses a fixed scheme for prioritizing packets over an active link: first by weight, then by flow ID, as discussed later and also illustrated in Example 1. As will be apparent later, such a scheme *uniquely determines* the exact routing schedule of the packets from a given sequence of configurations; this obviates the need to include such packet-level details as a parameter to the objective function ψ or in the output of the MHS problem.

Objective ψ in Example 1. Consider again the Example 1 and Figure 1. Therein, it is easy to see that the objective value ψ of the given solution is 150. For the optimal solution, which is the sequence of configurations $(M_1 \cup M_3, 50)$, $(M_4 \cup M_5, 50)$, $(M_2, 100)$, $((b, c), 100)$, the ψ value is 200.

4.1 Octopus Algorithm Description

We start with describing the intuition behind our algorithm. As mentioned above, due to the fixed routing scheme and given flow routes, the Octopus algorithm needs to only deliver a sequence of configurations.

Intuition and Design Outline. At a high-level, Octopus is basically a greedy algorithm that at each stage picks the "best" configuration *at that stage*. The basic idea is to look at each possible configuration (M, α) (of cost $(\alpha + \Delta)$) as "serving" the (weighted) packet-hops that are traversed during (M, α). Now, if we iteratively pick the *best* configuration (M, α), i.e., one that serves the packet-hops with maximum total weight per unit cost, then we can hope for a constant-factor approximation w.r.t. the objective ψ—if we can show that the optimization objective ψ is "monotonic" and "submodular" (we define these terms formally in §5). To design an algorithm as suggested above, we need to achieve the following steps.

(1) *Benefit of a Configuration.* First, we need to formally define and develop an algorithm to compute the benefit of a configuration, at any given iteration. The benefit function is essentially derived from the chosen objective function ψ.
(2) *Picking the Best Configuration.* Then, we need to develop an efficient algorithm to select the configuration with the highest benefit per unit cost at each iteration. Note that, in our context, there are infinitely many configurations available for selection.

(3) *Approximation Algorithm.* Based on the above, the approximation algorithm is merely an iterative procedure that picks the best configuration in each iteration. We discuss necessary details below.
(4) *Performance Guarantee via Submodularity.* Prove the objective function ψ to be monotonic and submodular. However, as shown in §5, ψ is monotonic but not submodular; fortunately, we are able to circumvent this challenge by showing that the ψ function satisfies a weaker property that we can use to prove an appropriate approximation factor.

We consider each of the above tasks below, with the last task deferred to the following section.

Configuration Benefit and Cost. Let $\mathbb{S}$ be the sequence of configurations already picked `Octopus` algorithm at the start of a given iteration. *Benefit of a configuration* (M, α), at the stage when $\mathbb{S}$, a sequence of configurations, has already been picked by `Octopus`, is denoted as $\mathcal{B}((M, \alpha), \mathbb{S})$ and is essentially the improvement of the objective ψ value due to (M, α), i.e.,

$$\mathcal{B}((M, \alpha), \mathbb{S}) = \psi(\langle \mathbb{S}, (M, \alpha) \rangle) - \psi(\mathbb{S}). \tag{2}$$

To illustrate the dependence of the benefit function over its second argument, observe that in Example 1, $\mathcal{B}((M_4, 50), \emptyset)$ is zero, while $\mathcal{B}((M_4, 50), (M_3, 50))$ would be 25. More generally, $\mathcal{B}((M_4, 50), (M_3, \alpha))$ would be $\alpha/2$ for any $\alpha \leq 50$. This is because (M_3, α) configuration routes α packets of (c, a)-flow over (c, b), and thus make the (M_4, α) useful for routing (c, a)-flow packets over (b, a).

Cost of a configuration (M, α) is defined as $(\alpha + \Delta)$, as this is the number of time slots consumed by the configuration.

Beyond the above definition of configuration benefit, we need to develop an algorithm to actually compute it. To use Equation 2 to compute benefit directly, we will have to first develop an algorithm to compute the function $f()$, used to define ψ in Equation 1. But, for sake of clarity, rather than designing an algorithm to compute $f()$, we instead develop an algorithm to compute benefit using other related notions. We define these notions below, before describing how to compute a configuration's benefit.

Remaining Traffic T^r; Functions $g()$, and $h()$. *Remaining* traffic load T^r essentially represents the *current* status of the packets and flows in the network, after the packets have been routed via $\mathbb{S}$. Similar to T, T^r is represented as a set of traffic subflows, with each *subflow* represented as $\langle$`flowID, size, remainingRoute`$\rangle$. Note that T^r may have multiple subflows with the same flow ID.

Given T^r, for each link $(i, j) \in G$, we define $g(i, j, \alpha)$ as the maximum weight of α packets in T^r that are situated at i with next hop (i, j). As shown in Figure 2, $g(i, j, \alpha)$ can be easily computed by first sorting the packets by their weight, and taking the weight of the top α packets. The value $g(i, j, \alpha)$ is essentially the maximum weight of packets that can be routed over (i, j) by a (M, α) where $(i, j) \in M$.

Finally, for each link $(i, j) \in G$, we define $h(i, j, k)$ as the *number* of packets in T^r of weight $1/k$ situated at i with the next hop of (i, j). We will use this notion of $h()$ to determine the set of α's to consider for computing the *best* configuration. See Figure 2.

Computing Benefit $\mathcal{B}((M, \alpha), \mathbb{S})$. Note that benefit $\mathcal{B}((M, \alpha), \mathbb{S})$ of configuration (M, α), when $\mathbb{S}$ has been already selected, is the maximum weight of the packets that can traverse over the links in M in α time slots. Thus, $\mathcal{B}((M, \alpha), \mathbb{S})$ can be computed as the sum of $g(i, j, \alpha)$ values for links (i, j) in M. Overall, the the steps to compute $\mathcal{B}((M, \alpha), \mathbb{S})$ are: (i) Compute T^r from $\mathbb{S}$ as discussed below, (ii) Compute $g(i, j, \alpha)$ for each $(i, j) \in M$, and then, (iii) Use the equation below.

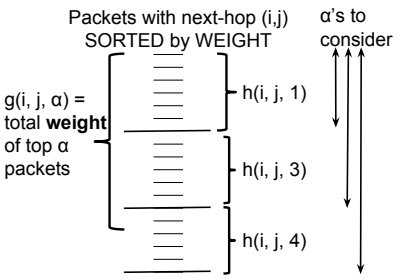

Figure 2: Functions $g()$ and $h()$, and set of α values to consider, for a link (i, j).

$$\mathcal{B}((M, \alpha), \mathbb{S}) = \sum_{(i,j) \in M} g(i, j, \alpha). \tag{3}$$

Computing or Updating T^r. Originally, when no configuration has been scheduled, T^r is just the original traffic load T. To compute T^r for a given sequence $\mathbb{S}$, we update it iteratively for each configuration (M, α) in $\mathbb{S}$ as follows. For each link $(i, j) \in M$, we find the packets in current (before (M, α) is added) T^r that are situated at i and whose next hop is j, sort them by their weights, pick the top α packets, and "route" them over (i, j) to update T^r. Here, routing the selected packets essentially means adding subflows and/or updating subflow parameters in T^r appropriately (we skip the tedious details). Note that the set of packets picked above for routing may not be unique, as we saw in routing packets via $(M_2, 100)$ in Example 1. To resolve this potential non-deterministic behavior (and thus, ensure accurate computation of later benefit values), we use a simple and scheme of prioritizing packets based on flow ID (among packets of the same weight). Thus, for the above routing of packets over (i, j), we first sort the relevant packets by weight and then flow IDs, and then pick the top α packets.

Selecting the Best Configuration. We now describe how to pick the configuration (M, α) with the maximum value of $\mathcal{B}((M, \alpha), \mathbb{S})/(\alpha + \Delta)$, highest benefit per unit cost, for a given schedule $\mathbb{S}$. Using insights from [36], our technique consists of the following steps.

(1) First, we note that we only need to consider a small number of α values. In particular, we claim that, for a given T^r, we only need to consider α's that are of the form $\sum_{k=1}^{m} h(i, j, k)$ for some link $(i, j) \in G$ and a positive integer m. We defer the formal proof to the Appendix; see Figure 2 and the psuedocode in Procedure 1. The restriction to the above set of α values is essentially ensured by the fact the benefit-per-unit-cost function is monotonic in α in between the above set of values, and thus, the $\mathcal{B}$ function's maxima occurs at to one of these α values.
(2) Second, note that by Equation 3, for a *given* α, the best configuration (M, α) can be computed by just finding the maximum weighted matching in a weighted bipartite graph G' where G' is the weighted version of the network graph G with weight of $g(i, j, \alpha)$ assigned to each link (i, j). This is

because $g(i, j, \alpha)$ is the maximum weight of packets that can be routed over (i, j), for a given α.

See Procedure 2 for the overall pseudo-code.

Procedure 1: SetOfAlphas(G, T^r)

Data: Network Graph G, Remaining Traffic Load T^r
Result: Set of α's to consider in Procedure 2

$\mathcal{A} \leftarrow \emptyset$;
for $(i, j) \in G$ **do**
  for $k = 1$ *to* $\mathcal{D}$ **do**
    $h(i, j, k)$ = number of packets in T^r with weight $1/k$ that are situated at i to go to next-hop j;
  end
  for $k = 1$ *to* $\mathcal{D}$ **do**
    $\mathcal{A} = \mathcal{A} \cup (h(i, j, 1) + h(i, j, 2) \ldots h(i, j, k)$;
  end
end
return $\mathcal{A}$;

Procedure 2: BestConfiguration(G, T^r)

Data: Network Graph G, Remaining Traffic Load T^r
Result: (M, α) with most benefit per unit cost.

$u \leftarrow 0$;
$\mathcal{A}$ = Set of α's to consider for T^r; /* See Figure 2 */;
for $\alpha \in A$ **do**
  $g(i, j, \alpha)$ = Maximum weight of α packets in T^r situated at i to go to next-hop j;
  $G' \leftarrow$ weighted graph from G with a weight of $g(i, j, \alpha)$ on link (i, j);
  $M' \leftarrow$ maximum weighted matching in G';
  $y = \frac{\text{weight of } M'}{(\alpha+\Delta)}$;
  if $y > u$ **then**
    $u \leftarrow y$;
    $M \leftarrow$ unweighted M';
    $s \leftarrow (M, \alpha)$;
  end
end
return s;

Octopus **Algorithm.** Based on the above concepts and procedures, our overall Octopus algorithm can be simply described as follows. The Octopus Algorithm goes through iterations, and in each iteration, it (i) picks the configuration (M, α) with the highest benefit per unit cost, and (ii) updates the T^r as described above. The algorithm stops as soon as the entire traffic is served, or the total cost of the configurations *exceeds* W; in the latter case, to ensure that the total solution cost is at most W, we reduce the number of time slots of the *last* configuration appropriately.

Octopus Time Complexity. Let $|T|$ be the total number of flows in T. Then, the maximum number of subflows in T^r at any stage is at most $|T|\mathcal{D}$. Thus, the overall time complexity of Octopus is $O(|T|^2\sqrt{n}\mathcal{D}^2\frac{W}{\Delta}\log W)$, as there are at most W/Δ iterations, and in each iteration, there are at most $|T|\mathcal{D}^2$ α's and thus calls to maximum weight matching, each of which takes $O(|T|\sqrt{n}\log(W))$ time [13].[2] However, of most practical significance is the time taken to execute a single iteration—since the iterative nature of Octopus means that we only need to compute and implement one iteration at a time; in other words, each iteration can be computed while the traffic is actually being routed using the configuration output by the previous iteration. Now, since computation of weighted matchings within each iteration can be run in parallel on a large multi-core machine, each iteration of Octopus can be computed in $O(|T|\sqrt{n}\log(W))$ time which can be approximated to $O(n^{1.5}\log(W))$ as the traffic matrix T is typically sparse with $|T| = O(n)$. Thus, each iteration can be executed in a few milliseconds (see §8) even for $n = 1000$. To reduce the run time further, one can employ an approximation scheme for the weighted matching; in §8, we show that the simple greedy approximation for the weighted matching can be executed in a fraction of a millisecond for $n = 1000$, with only a minimal degradation in performance.

5 OCTOPUS APPROXIMATION PROOF

Iterative greedy algorithms are known to yield constant-factor approximations, if the underlying objective function is monotonic and submodular. Below, we discuss why our objective function ψ is not submodular, and then, use the derived insight to prove a weaker form of submodularity which is sufficient to derive an approximation factor.

Recall that, in our context of MHS problem, an MHS schedule/solution is a *sequence* (rather than a set) of configurations. Thus, we need slightly different notions of monotonicity and submodularity than the traditional notions which are defined for functions over sets.

Monotonicity of ψ. We start with addressing the monotonicity of the objective function ψ, which informally says that prefixing or appending a sequence of configurations with another sequence can only increase the objective value (we omit the proof here).

Lemma 1. *The objective function ψ is monotonic, i.e., $\psi(\langle \mathbb{S}_1, \mathbb{S}_2 \rangle) \geq \max(\psi(\mathbb{S}_1), \psi(\mathbb{S}_2)$.* ■

Non-Submodularity of ψ. The objective function ψ (over sequences) would be considered submodular if

$$\psi(\langle \mathbb{S}, (M, \alpha) \rangle) - \psi(\mathbb{S}) \geq \psi(\langle \mathbb{S}', (M, \alpha) \rangle) - \psi(\mathbb{S}')$$

for any two sequences $\mathbb{S}$ and $\mathbb{S}'$ such that $\mathbb{S}$ is a *prefix* of $\mathbb{S}'$. The above equation can also be written in terms of the benefit function $\mathcal{B}$:

$$\mathcal{B}((M, \alpha), \mathbb{S}) \geq \mathcal{B}((M, \alpha), \mathbb{S}'), \tag{4}$$

which essentially says that the benefit of a configuration *never increases* with the growth of $\mathbb{S}$ (i.e., over the iterations of the Octopus algorithm). Unfortunately, the above condition *does not hold* for the case of multi-hop traffic load for the objective function ψ. For instance, as mentioned before in Example 1, benefit of configuration

[2] In §8, we also evaluate a binary-search scheme which reduces the number of weighted matchings needed in every iteration from $|T|\mathcal{D}^2$ to $(\log \min(n, W))$ by using a binary search over the range of flow sizes in T. In our evaluations, we observe that this binary-search approach incurs only a minimal loss in performance.

(M_4, α) increases with selection of $(M_3, 50)$ —making the objective function ψ is not submodular. Note that the number-of-packets-delivered objective is also not submodular (see Example 1).

Weaker Submodularity and Approximation Results. The above Eqn. 4 can be shown equivalent to:

$$\mathcal{B}(\langle O_1, O_2, \ldots, O_{k'}\rangle, \mathbb{S}) \leq \sum_{j=1}^{k'} \mathcal{B}(O_j, \mathbb{S}) \quad \forall k'$$

where we have used an extended definition of the $\mathcal{B}$ function, by allowing a *sequence* of configurations in the first argument. Though the $\mathcal{B}$ function does *not satisfy* the above equation (as ψ is not submodular), we claim that $\mathcal{B}$ satisfies a slightly weaker condition—see the below lemma. We use the lemma to prove the approximation result of `Octopus` Algorithm. We defer the proofs to Appendix.

Lemma 2. *Let $\mathbb{S}$ be a sequence of configurations, and $O_1, O_2, \ldots, O_{k'}$ be a set of k' configurations. We claim that:*

$$\mathcal{B}(\langle O_1, O_2, \ldots, O_{k'}\rangle, \mathbb{S}) \quad \leq \quad \mathcal{D} \times \sum_{j=1}^{k'} \mathcal{B}(O_j, \mathbb{S}). \qquad \blacksquare$$

Theorem 1. *The `Octopus` algorithm delivers a solution whose objective ψ value is at least $(1 - 1/e^{1/\mathcal{D}})\frac{W}{W+\Delta}$ times that of the optimal ψ possible.*[3] $\blacksquare$

Note that $\frac{W}{W+\Delta} \approx 1$ for large W, and $\mathcal{D}$ is expected to be a very small constant (about 2-4; see §3). Assuming $\frac{W}{W+\Delta} \approx 1$, the approximation ratios for $\mathcal{D}$ = 1,2,3,4 are 63%, 39%, 28%, and 22%.

Traversing Multiple Hops in a Configuration. Recall our first assumption that a packet traverses at most one hop in any configuration; here, we relax this assumption. Note that traversing multiple hops in a configuration could be useful. For instance, in Example 1, if the first configuration contains both links (d, a) and (a, b), then all the packets of (d, a, b)-flow can be delivered to their final destination in just a single configuration. In terms of feasibility, note that network switch latency can be as low as a few 100s of nanoseconds [11], while a time slot in our context is expected to be on the order of a few microseconds (see §8). Thus, the time needed to transfer a packet from an intermediate node's input port to an appropriate VOQ of its output port, should be at most 1-2 time slots.

The biggest challenge in allowing multiple hops per configuration is in ensuring a performance guarantee; more specifically, the challenge is in computing the best configuration—as configuration (M, α)'s benefit would now depends on the *multi-hop paths* enabled by M with these multi-hop paths from different flows "competing" for the common links in M. We address the above challenge by using a greedy algorithm to compute an approximate (rather than optimal, as before) matching for a given α. It is well-known [6] that a greedy algorithm yields a a 1/2-approximate weighted matching to the traditional maximum-weighted matching problem. However, in our context wherein the weights on the edges come from multi-hop flows, the 1/2-approximation factor doesn't hold. In our context, we propose a greedy matching algorithm that essentially creates the approximate matching (for a given α) by adding edges iteratively, by picking the edge that increases the benefit of the configuration by most at each stage. It can be shown (we omit the proof) that such a greedy matching algorithm yields a $\frac{1}{2\mathcal{D}}$-approximate configuration. This introduces an additional factor of $\frac{1}{2\mathcal{D}}$ in the exponent of the original approximation factor, and thus, yielding an overall approximation factor of $(1 - 1/e^{1/(2\mathcal{D}^2)})\frac{W}{W+\Delta}$.

Theorem 2. *When a packet may traverse multiple hops within a configuration, the modified `Octopus` algorithm as described above delivers a solution whose objective ψ value is at least $(1-1/e^{1/2\mathcal{D}^2})\frac{W}{W+\Delta}$ times that of the optimal value possible.* $\blacksquare$

6 TRAFFIC ROUTING AND SCHEDULING

In this section, we consider a more general joint-optimization problem of choosing flow routes as well as determining the sequence of network configuration, and thus, relaxing the second assumption made in §4. In particular, we now associate each traffic flow with with multiple possible routes and the general `MHS` problem is to choose a route for each flow as well as the sequence of configurations over which to route the flows over the chosen routes. All the terminology from previous sections holds here too.

`Octopus+` Algorithm. We refer to our generalized algorithm as `Octopus+`. We first describe our `Octopus+` algorithm, under the following two assumptions: (i) For each flow, no two routes in the parameter `routes` have the same first hop. (ii) We do not "backtrack" on the choice of a route, i.e., for each *packet*, the choice of its route is made at its first hop and is final. We relax these assumptions momentarily. Under these assumptions, the `Octopus+` algorithm remains largely same as the `Octopus` algorithm, except for the way $g(i, j, \alpha)$ and $h(i, j, k)$ are computed; we describe these changes below. These changes reflect the choice a packet has *at the source*: to pick one of the given set of routes (or the first hop, as per assumption (i) above). In essence, when computing $g(i, j, \alpha)$ and $h(i, j, k)$ values at link (i, j), for packets with source i we take into consideration all the potential first hops. Formally, the changes to computing $g()$ and $h()$ functions is as follows.

(1) $h()$ *Computation.* In `Octopus+`, when determining the set of α's to consider, we consider all potential first-hops simultaneously, for packets at their original source. Thus, $h(i, j, k)$ is now computed as the number of packets in T^r with weight $1/k$ that are either: (i) situated at their original source i with (i, j) as the first hop of one of the routes available, or (ii) situated at an intermediate node i to go to the next hop j.
(2) $g()$ *Computation.* In `Octopus+`, we compute $g(i, j, \alpha)$ in a similar way as $h(i, j, *)$'s above. That is, we define and compute $g(i, j, \alpha)$, as the maximum weight of α packets in T^r that are either (i) situated at their original source i with (i, j) as the first hop of one of the routes available, or (ii) situated at an intermediate node i to go to the next hop j.

In addition, T^r needs to be updated appropriately to reflect the (final) choice made by the packets at their first hop. Note that in the above, different subflows of a flow may take different route choices—resulting in out of order packets at the destination, which can be handled by packet reordering at the destination [10].

[3] The corresponding optimal solution need not prioritize the packets by flow IDs, as in `Octopus`.

Backtracking. We now enhance our above algorithm by allowing packets to "backtrack" on their choice of route. Note that Octopus+ algorithm runs only offline, and is not actually routing the packets in real-time. Thus, any "routing" being done within Octopus+ algorithm is merely for bookkeeping purposes to generate a schedule, and similarly, any *backtracking* is merely rethinking of the schedule/solution to be delivered. We exploit backtracking in Octopus+, since it is essential to guaranteeing a constant-factor approximation bound. Fortunately, limited backtracking, i.e., only to a direct (one-hop) route, is sufficient. More formally, if a packet p with source s and destination d has been routed to an intermediate node y (on some chosen route) via an already chosen sequence $\mathbb{S}$ of configurations, then, in the next configuration (M, α), we allow p to be routed directly via link (s, d) if $(s, d) \in M$. In such a case, p's prior routing from s to y in $\mathbb{S}$ is annulled.[4] See Figure 3.

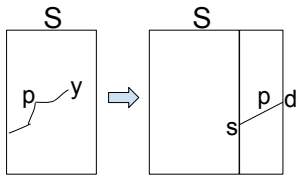

Figure 3: Backtracking.

To incorporate backtracking to a direct link as described above into Octopus+, we further modify the above computations of $h(i, j, k)$ and $g(i, j, \alpha)$ at (i, j) by also considering packets whose source is i and destination is j irrespective of where they may be currently situated in T^r. In addition, when updating T^r, we prioritize routing of packets on their direct link (s, d) compared to them being routed over their next hop, say (y, y'), if both links (s, d) and (y, y') exist in the selected configuration.

Allowing Routes with Common First Hops. Now, we discuss relaxing the other assumption made earlier: i.e., we allow routes parameter to contain routes with the same first hop. In this case, the computation of $g(i, j, \alpha)$ (and $h(i, j, k)$) could be incorrectly inflated by a single packet contributing to a $g(i, j, \alpha)$ value for *each* of its route choice that contain (i, j) as the first hop. The simple fix to this issue is to ensure that in computation of $g(i, j, \alpha)$ or $h(i, j, k)$ each packet is considered only once.

Approximation Result. Using techniques similar to the previous section, we can show that the above described Octopus+ algorithm delivers a constant-factor approximation solution, for the special case when for each given flow, there are at most two routes, one of which is a direct link. We defer the proof to Appendix.

Theorem 3. *For the special case, wherein each traffic flow has at most two route choices one of which is a direct link, the* Octopus+ *algorithm delivers a solution whose objective ψ value is at least $(1 - 1/e^{1/c})(\frac{W}{W+\Delta})$ times that of the optimal value possible, where $c = \mathcal{D}$ if $\mathcal{D} \leq 2$ and $c = (4/3)\mathcal{D}$ otherwise.* ■

We evaluate Octopus+ for the general case in §8.

7 MORE GENERAL NETWORKS

More General Network Graphs. We now consider more general circuit-switched network models—in particular, allowing more general subsets of links (rather than just matchings) to be active at any instant. For simplicity, we focus here only on generalizing the Octopus algorithm and state (without proof) the impact on the approximation factor.

K Ports per Node. We now consider the network model, wherein a node may have multiple input and output ports–thus, allowing multiple links per node to be active simultaneously. E.g., in wireless optical networks [21], each rack (node) can be connected to 10s of wireless optical (FSO) transceivers. In such networks, any r-regular subgraph (i.e., a combination of r matchings) over the racks/nodes is a valid configuration, where r is the number of input as well as output ports per node. In this context, to compute the best configuration for a given α, we employ a greedy approach that iteratively selects the best matching available, until r disjoint matchings have been selected, and returns the combination of these r matchings. Using similar proof techniques as in Theorem 1, we can show that such a greedy algorithm yields a $(1 - 1/e)$-approximate matching. The overall approximation factor of the Octopus algorithm for above then changes to $(1 - 1/e^{(1-1/e)/\mathcal{D}})\frac{W}{W+\Delta}$.

Bidirectional Ports per Node. In networks with full-duplex optical switches or with bidirectional wireless optical links (e.g., FireFly [21]), each node is connected to full-duplex ports and each link when active is bidirectional. Such network architectures can be represented by general (not bipartite) undirected graphs over n nodes, and the valid configurations are matchings (with bidirectional links) over the graph. To tailor Octopus algorithm for general graphs, we need to compute maximum weighted matchings in general (rather than mere bipartite) graphs. Rest of the algorithm remains the same with its performance guarantee; the overall time complexity increases by a factor of just $\sqrt{\log n}$ by using [17] for general graph matching.

Scheduling in a Hybrid Network. Our algorithms can be easily extended to a hybrid network comprised of a circuit network and a packet (electrical) network as follows. Given a traffic load T and window W, first route as much of T as possible over the packet network, and then use Octopus or Octopus+ to route the remaining traffic over the circuit network. The above strategy can be easily shown to have the same performance guarantees as Octopus or Octopus+ for circuit networks.

Makespan Minimization Problem. Our algorithms can also be used to solve the *makespan minimization* problem of finding the shortest window W to fully serve a given traffic load—using simple binary search, yielding a $O(\log |T|)$ (instead of a constant) approximation factor .

8 EVALUATION

In this section, we empirically evaluate the performance of our developed techniques in terms of network throughput, i.e., the number of packets delivered. We use traffic loads generated based on published characteristics, as well as publicly available actual traffic loads from Facebook and Microsoft data centers.

Simulation Setup. We use a simple custom packet-level simulator that routes traffic synchronously, one packet transmission in each time slot over each active link. Such a simulator is sufficient for our context, due to our assumption of division of time into time slots. Unless otherwise stated, we consider a circuit network over 100 network nodes each with a single input and output port, and use

[4] In principle, we could reuse the freed up time slots due to annulment; however, we do not exploit this optimization in Octopus+, for simplicity.

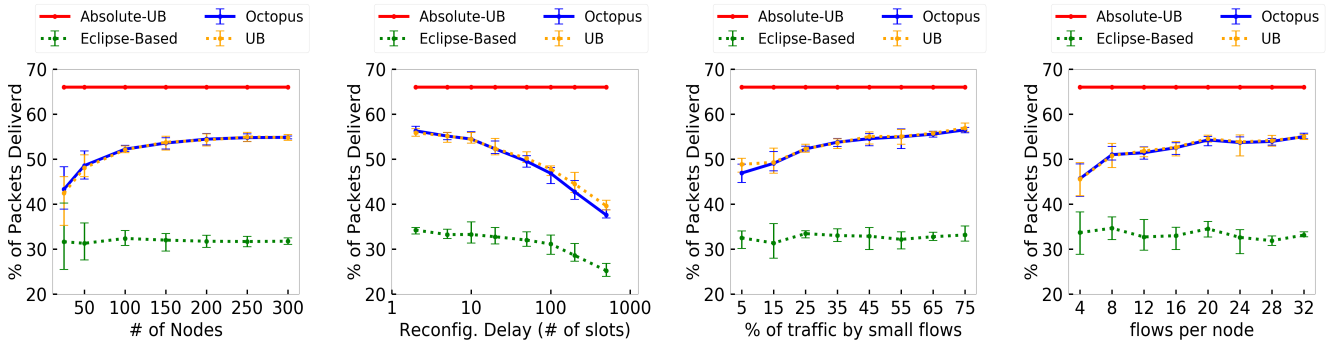

Figure 4: Packets delivered (%) for varying: (a) Number of nodes, (b) Reconfiguration delay, (c) Traffic skew (i.e., c_S as a percentage of $(c_S + c_L)$), (d) Traffic sparsity (i.e., $n_L + n_S$)

$W = 10,000$ and $\Delta = 20$ time slots. We are implicitly considering one microsecond time slots, corresponding to transmission time for 100 Kbits (12.5 KB) packets over 100 Gbps links. Each data point in the plots is an average over 10 random instances.

Traffic Load. We generate traffic loads similar to prior related works [25, 36] based on traces from the University of Wisconsin [7] and Alizadeh et al. [5]. In particular, we assume 4 large flows (n_L) and 12 small flows (n_S) to each input or output port; these values are for 100-node networks, and are changed linearly for other networks. Note that a network of size n has n input ports and n output ports. Let c_L (c_S) be the total traffic carried by the large (small) flows on each port; we use $c_L + c_S = W = 10,000$, $c_L = 7000$ (70% of total) and $c_S = 3000$. Using these parameters, we generate random traffic load matrices exactly as in [36]. For each traffic flow, we assign a random route in the network of 1 to 3 hops, with equal number of flows getting 1/2/3 hops route.

Real Traffic Loads. In addition to the above generated traffic loads, we also use traces from: (i) a Microsoft datacenter in the form of traffic-matrix heatmaps [4] which determine the relative size of flows between pairs of nodes, and (ii) three different Facebook cluster types [2, 32] (Hadoop, front-end web server, and database server). From the above loads, we randomly select 100 rows and columns to create load for a 100 node network and then, scale the flow size values such that the maximum value of a flow is 10,000 (W).

Algorithms Compared. To the best of our knowledge, there have been no prior work on the MHS problem of scheduling multi-hop traffic in circuit networks. Thus, for the case of single given route per flow, we compare `Octopus` with the following `Eclipse-Based` approach based on [36], while for the case of multiple routes per flow, we compare `Octopus+` with a simple scheme based on `Octopus` (as discussed later). The `Eclipse-Based` approach is as follows. We first compute "one-hop traffic" T^{one} from the given multi-hop traffic by ignoring the ordering of hops in multi-hop paths, use `Eclipse` (one-hop algorithm) over T^{one} to compute a near-optimal sequence of configurations $\mathbb{S}_{one}$, and then, apply `Eclipse++` [36] (an algorithm to route multi-hop traffic over a *given* sequence of configurations) over $\mathbb{S}_{one}$. Later, we also compare our `Octopus` scheme with the traffic-agnostic scheme of [28].

Performance Metrics. We primarily focus on the metric of percentage of packets delivered to the destination, with respect to the total number of packets in T. In addition, we also report the `Link Utilization` metric, primarily to analyze the source of throughput efficiency of a scheme. The link utilization is the ratio of total number of packets traversed to the sum of the number of active links over all time slots.

Upper Bounds. To demonstrate the empirical near-optimality of our techniques, we also derive two upper bounds.

- The first upper bound is an *absolute upper bound* of 66% packets delivered from the generated traffic T; this 66% value is derived by observing that at most 10^6 hops can be traversed in 10k time slots in a 100-node network, and there are about 10^6 total packets with equal number of packets for 1-hop, 2-hop, and 3-hop routes. For the real-traffic loads, the absolute upper bound is 100%.
- We also consider a tighter upper-bound, denoted by UB, which is derived by running the following UB algorithm: Compute T^{one}, the unordered one-hop traffic (as defined above) from the given multi-hop traffic matrix T, and then run `Eclipse` over it. Essentially, UB is the best achievable performance by a polynomial algorithm for the MHS problem, since the UB algorithm: (i) works with fewer constraints, and (ii) uses the best approximation algorithm over the resulting optimization problem.[5] For the packet-delivered metric, we count a packet as delivered in UB only if all its hop have been served (in any order).

Throughput Performance and Link Utilization. In the first set of experiments, we compare the performance of algorithms in generated traffic loads for varying number of nodes (25 to 300), reconfiguration delay, traffic skewness, and sparsity. See Figures 4-5. Here, to vary skew, we vary $c_S/(c_L + c_S)$ for a fixed $(c_L + c_S)$, and to vary sparsity, we vary $n_L + n_S$ for a fixed $n_L/n_S = 1/3$. We observe that `Octopus` easily outperforms `Eclipse-Based` scheme by a significant margin; the reason for this is that the `Eclipse-Based` scheme has a very poor link utilization due to a poor choice of sequence of configurations based on T^{one}. More surprisingly, `Octopus` performs almost identical to the UB upper bound, with only a small gap with the absolute upper bound of 66%. The performance trend for other

[5] The one-hop version of the MHS problem is essentially a submodular function maximization problem with a cardinality constraint, which cannot be approximated better than `Eclipse`'s approximation ratio of essentially $(1 - 1/e)$ unless P=NP [15]. Strictly speaking, this argument holds only for the ψ objective.

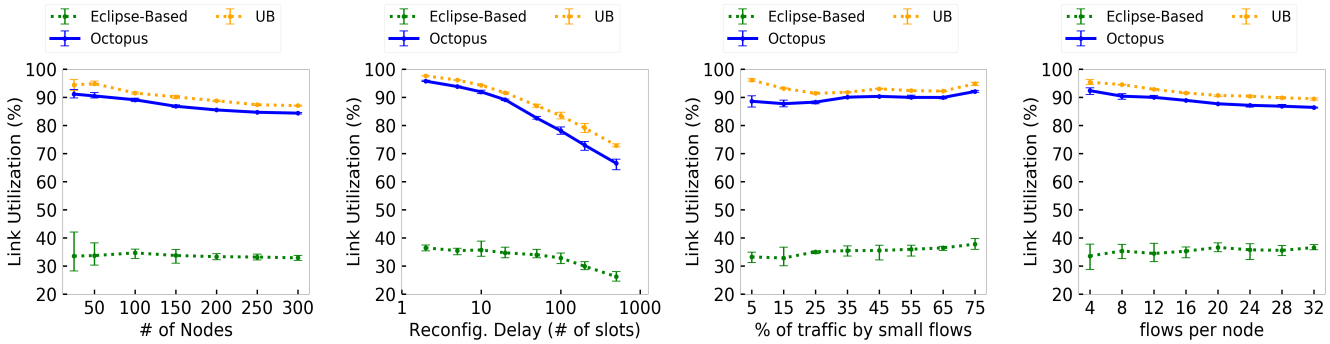

Figure 5: Link utilization (%) for varying: (a) Number of nodes, (b) Reconfiguration delay, (c) Traffic skew (i.e., c_S as a percentage of $(c_S + c_L)$), (d) Traffic sparsity (i.e., $n_L + n_S$)

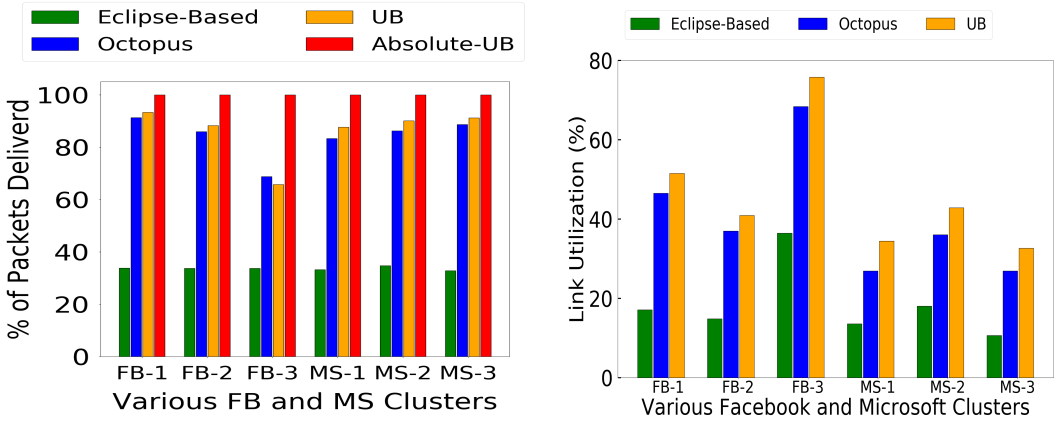

Figure 6: Performance over traffic from Facebook (FB) and Microsoft (MS) clusters.

varying parameters is as expected, except for the case of varying skewness wherein the performance seems to improve slightly with the increase in traffic carried by n_S flows. Increase in skewness with our chosen parameters essentially has an effect of making the flow sizes uniform across n_L and n_S flows, which is likely the reason for slightly improved performance. The binary-search variant `Octopus-B` (discussed later) performed near-identically with `Octopus` (including for real traffic loads below in Figure 7(a)), but has not been shown here for clarity.

Real Traffic Loads. We observe a similar pattern for the real traffic traces obtained from Microsoft and Facebook data centers, as described above. See Figure 6. Here, the percent of packets delivered is much than that for the generated traffic data due to the much lighter traffic—as evidenced by a much higher value of near-100% absolute upper bound. We see similar relative performance for link utilization (not shown), with `Octopus` again outperforming `Eclipse-Based` and also closely matching UB confirming `Octopus`'s effectiveness in choosing useful matchings. Note that however the link utilization values are much lower than the generated traffic loads; the reason behind this is that the real traffic load seem to be dominated by a very small number of large flows which results in many links being unused in each matching. Such a situation doesn't arise even for low C_S values in Figure 5(c), since the total number of large flows is still sufficiently high, i.e., n_L per node and thus $n_L \times N$.

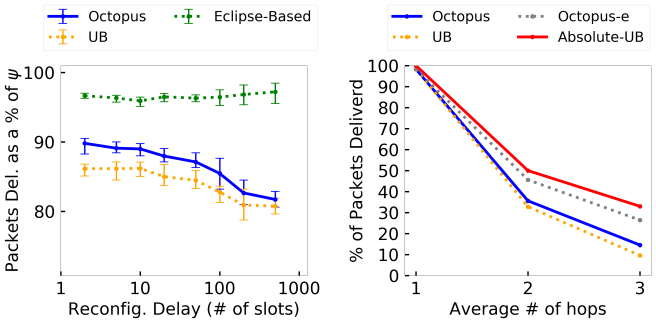

Figure 7: Evaluating the issue of undelivered packets: (a) Packets delivered as a percentage of objective value (ψ), for varying reconfiguration delay, and (b) Relative performance of `Octopus-e` for varying average hop count.

Evaluating Undelivered Packets. We now plot packets-delivered as a percentage of the objective value ψ, to get an idea of the percentage of packets that are left undelivered by the algorithms. In Figure 7(a), we see that the ratio is in the 80-90% range for `Octopus`, indicating that the issue of undelivered packets is not significant. Slightly worse ratio is observed for the UB scheme, as it likely ends up serving later hops of certain packets without ever serving their earlier hops; this is likely the reason for its inferior performance compared to `Octopus` for FB-3 in Figure 6(a) and Figure 7(b) (see below). More importantly, a high ratio for `Eclipse-Based` indicates

that its poor performance in previous plots is not due to undelivered packets, but largely due to poor link utilization.

Octopus-e Variant. To potentially mitigate the issue of undelivered packets, we evaluate a variant of Octopus, called Octopus-e, wherein we assign a slightly higher weight to hops closer to the destination.[6] In particular, we assign a weight of $1 + x\epsilon$ to a flow-route's hop that is x hops away from the source, for some small ϵ; this is independent of the weights assigned to packets based on flow route lengths as before. We evaluated Octopus-e, and observed that Octopus-e performed near-identical to Octopus in the previous experiments plotted in Figures 4–6 over generated and real traffic traces; we didn't show Octopus-e in those plots, for sake of clarity. The reason for Octopus-e not performing any better than Octopus in previous experiments was due to sufficiently many one-hop flows that giving higher precedence to later hops did not offer much advantage.

Higher Number of Hops. To further investigate the performance comparison of Octopus-e and Octopus, we consider a new setting where we consider flows with higher number of hops on an average. See Figure 7(b), where the traffic is as before, except that all the flows have the same route length (varied from 1 to 3). Here, we clearly see that Octopus-e results in a higher packets-delivered metric, with the gap increasing with increase in average number of hops. This shows the benefit of Octopus-e in such traffics. Interestingly, we also see that Octopus and Octopus-e both outperform the UB, likely due to the issue of undelivered packets (observe its worse performance in Figure 7(a)) which is accentuated with increase in average number of hops for reasons mentioned above.

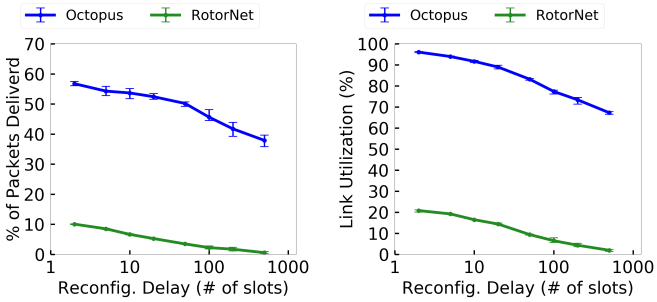

Figure 8: Octopus vs. RotorNet Performance Comparison.

Performance Comparison with RotorNet [28]. We now compare the performance of our Octopus algorithm with the traffic-agnostic RotorNet [28] schedule which is not particularly suited for our setting and problem—but the comparison shows the benefit of taking the given traffic flows into consideration while designing an efficient schedule. Contrary to our setting, RotorNet assumes a complete bipartite graph over the input and output ports, but it can still be applied to our MHS problem by assuming availability of all edges anyway (for scheduling purposes). See Fig. 8 which shows the throughput and link utilization comparison of Octopus with RotorNet. As expected, RotorNet performs poorly in our setting. The main reason for the poor performance of RotorNet is that it uses many edges in each matching that are not carrying any flow traffic as evidenced by its very low link utilization. Other reasons for its poor performance are: (ii) it uses a fixed and uniform duration (we used $10 \times \Delta$ [18]) for each matching, and (iii) it doesn't prioritize packets as Octopus does, i.e,. by assigning higher weights to packets with shorter routes. Note that RotorNet's key advantage is that its controller is traffic agnostic and thus very simple and decentralizable. RotorNet works well in complete networks where it can (and does) route "most" packets via direct one-hop routes, while exploiting 2-hop routes only "opportunistically." However, in our MHS problem, the flow routes are given/fixed and the multi-hop flows can't be routed directly (also, because the direct route may not even exist in the network).

[6]This weight assignment is done in the scheduler at controller, and hence does not require any special packet-prioritization techniques at a switch.

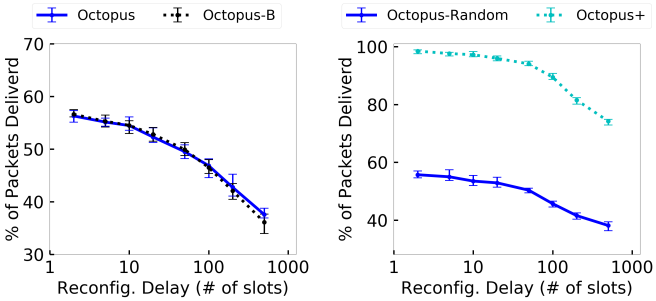

Figure 9: (a) Octopus-B Performance. (b) MHS problem with multiple routes per flow.

Evaluating Octopus-B: Binary Search Over α's. To evaluate the time complexity vs performance trade-off, we implement and evaluate a variant called Octopus-B which does a binary search over the α's to try to find the best configuration, as suggested in §4. Essentially, Octopus-B finds one of the maxima, not necessarily the global maximum. As mentioned before, Octopus-B performed near-identically to Octopus, for all data points in previous plots i.e., Figures 4, 5, and 6, but not shown therein for clarity. However, as an example, we show Octopus-B vs. Octopus performance for varying reconfiguration delay in Figure 9(a). The near-identical performance suggests that the benefit per unit cost function's characteristics are very amenable to a binary search for maximum. In effect, the above observations help reduce the worst-case time complexity by a factor of $|T|\mathcal{D}^2$.

Evaluating Octopus+: Multiple Routes per Flow. We now evaluate the performance of Octopus+ when there are multiple routes given for each flow. Neither Eclipsed-Based nor the previously mentioned upper bounds apply here, so we compare Octopus+ with an approach that picks a route randomly for each flow and then applies Octopus algorithm; we refer to this as Octopus-random algorithm. We evaluate the algorithms for the generated traffic loads above, except here, for each flow, we provide 10 different route choices of varying lengths chosen uniformly between 1 and 3 hops. See Figure 9(b). It's not surprising to see that Octopus+ easily outperforms Octopus-random.

Execution Time. As mentioned in §4.1, the time taken by a single weighted matching algorithm is of the most practical significance—since Octopus only needs to be implemented and executed one iteration at a time and multiple matchings within an iteration can be run in parallel. Optimal weighted matching in a bipartite graph has

been studied extensively with the best known time complexity [13] of $O(|T|\sqrt{n}\log(W))$. In our experiments, we used an available C++ implementation [3] for the weighted bipartite matching, which can take up to a few milliseconds for large n, on a 3.2 GHz desktop. The run time could possibly be reduced using an optimized custom code or other algorithms such as [17]. To demonstrate practical viability of our techniques, we instead implemented a simple 2-approximate greedy matching algorithm that adds edges in decreasing order of weights. Since the weights in our context are integers and bounded by W, this greedy algorithm can actually be implemented in linear time using $O(\max(W, |T|))$ space. An `Octopus-G` algorithm based on the greedy weighted matching scheme also incurs only a minimal loss of performance. In Fig. 10, we plot execution times (of a single iteration) and network throughput for the `Octopus` and `Octopus-G` algorithms. We observe that `Octopus-G`'s iteration took about 650 microseconds for a $n = 1000$ network (and 100, 0000 traffic flows), and `Octopus-G`'s performance is very close (95% or above) to that of `Octopus`. Thus, `Octopus-G` is practically viable on a supercomputer with sufficiently large number of cores. We make three further remarks: (i) Due to edge weights being integral and bounded by W, the 2-approximation greedy matching algorithm is incredibly simple in our context: it involves merely updating and accessing a W-size array. Thus, special-purpose computing cores could easily reduce the run time significantly. (ii) Theoretically, the number of cores required could be as high as $|T|\mathcal{D}^2$ which could be as high as about 1.5 million for $n = 1000$ and $\mathcal{D} = 3$. However, supercomputers with so many cores are feasible; e.g., the recent Cerebras Wafer Scale Engine supercomputer delivers 400,000 programmable compute cores [1]. (iii) More sophisticated approximation algorithms for bipartite weighted matching exist with linear-time complexity and near-optimal performance guarantees, with perhaps the most promising being [12]. These may offer better performance and run time than `Octopus-G`.

9 CONCLUSIONS

To the best of our knowledge, ours is the first work to address the problem of determining a sequence of network (global) configurations for multi-hop traffic in general circuit networks, and essentially solves the open problem mentioned in [36] and [25]. There are two generalizations of our work that are very challenging and of significant interest: (i) *Localized reconfigurations*, e.g., in the context of recently proposed wireless optical architectures, network reconfigurations are not necessarily global and specialized scheduling algorithms with "localized" reconfigurations can be very useful; (ii) *Online Scheduling*, wherein the input is not a fixed traffic load, but a sequence of flow arrivals and the problem is to configure the network and schedule the packets/flows as flows arrive (and complete). Finally, in our context, distributing the `Octopus` algorithm across switches could have the benefit of overcoming challenges due to non-uniform latencies from the central controller to each switch. The above directions are the focus of our future work.

ACKNOWLEDGMENTS

We would like to acknowledge Janardhan Kulkarni for initial useful discussions, and the anonymous referees for their comments and suggestions. The work was supported by NSF Award # 1815306: NeTS: Small: A Wireless Backhaul for Multi-Gigabit Picocells Using Steerable Free Space Optics.

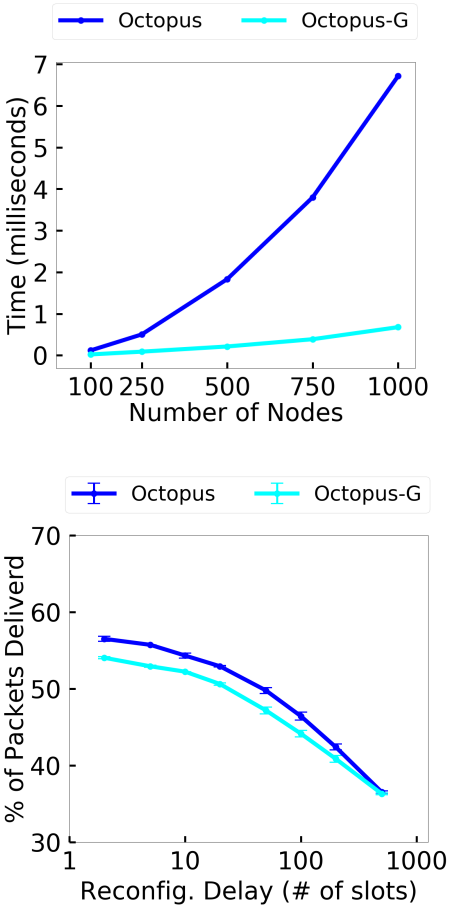

Figure 10: `Octopus` vs `Octopus-G`. (a) Execution times of an iteration on a large multi-core processor for increasing network size, (b) Packets delivered (%) for varying reconfiguration delay, over a 1000-node network.

REFERENCES

[1] Cerebras Wafer Scale Engine. https://www.cerebras.net/product/#chip.
[2] Facebook, FBFlow dataset. https://www.facebook.com/network-analytics.
[3] Google OR Tools: Linear assignment. https://developers.google.com/optimization/assignment/linear_assignment.
[4] Microsoft datacenter traffic heatmaps. https://www.microsoft.com/en-us/research/project/projector-agile-reconfigurable-data-center-interconnect/, 2016.
[5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). *ACM SIGCOMM*, 2011.
[6] David Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, 1983.
[7] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of ACM IMC*, 2010.
[8] Calient. Edge|640 optical circuit switch, 2018.
[9] Cheng-Shang Chang, Wen-Jyh Chen, and Hsiang-Yi Huang. Birkhoff-von neumann input-buffered crossbar switches for guaranteed-rate services. *IEEE Transactions on Communications*, 49(7), 2001.
[10] Li Chen, Yuan Feng, Baochun Li, and Bo Li. Promenade: Proportionally fair multipath rate control in datacenter networks with random network coding.

IEEE Trans. on Par. and Dist. Sys., 2019.
[11] Paul T Congdon, Prasant Mohapatra, Matthew Farrens, and Venkatesh Akella. Simultaneously reducing latency and power consumption in openflow switches. *IEEE/ACM Transactions on Networking (TON)*, 22(3), 2014.
[12] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM (JACM)*, 61(1):1, 2014.
[13] Ran Duan and Hsin-Hao Su. A scaling algorithm for maximum weight matching in bipartite graphs. In *SODA*, 2012.
[14] Nathan Farrington. Optics in data center network architecture. http://nathanfarrington.com/papers/dissertation.pdf.
[15] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.
[16] Leslie G. Valiant. A scheme for fast parallel communication. *SIAM J. Comput.*, 11:350–361, 05 1982.
[17] Harold Gabow and Robert E. Tarjan. Faster scaling algorithms for general graph-matching problems. *JACM*, 3(4), 1991.
[18] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *SIGCOMM*. ACM, 2016.
[19] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *ACM SIGCOMM*, 2011.
[20] Navid Hamedazimi, Himanshu Gupta, Vyas Sekar, and Samir R Das. Patch panels in the sky: A case for free-space optics in data centers. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.
[21] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R Das, Jon P Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *SIGCOMM*, 2014.
[22] Janardhan Kulkarni, Euiwoong Lee, and Mohit Singh. Minimum Birkhoff-von Neumann decomposition. In *Springer IPCO*, 2017.
[23] Xin Li and Mounir Hamdi. On scheduling optical packet switches with reconfiguration delay. *IEEE Journal on Selected Areas in Comm.*, 21(7), 2003.
[24] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M Voelker, George Papen, Alex C Snoeren, and George Porter. Circuit switching under the radar with REACToR. In *NSDI*, 2014.
[25] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. Scheduling techniques for hybrid circuit/packet networks. In *ACM CoNEXT*, 2015.
[26] W. M. Mellette and J. E. Ford. Scaling limits of mems beam-steering switches for data center networks. *Journal of Lightwave Technology*, 33(15), 2015.
[27] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *NSDI*, 2020.
[28] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *SIGCOMM*, 2017.
[29] Matthew Mukerjee, Christopher Canel, Daehyeok Kim, and Srinivasan Seshan. Adapting TCP for reconfigurable datacenter networks. In *NSDI*, 2020.
[30] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *SIGCOMM*, 2013.
[31] Balaji Prabhakar and Nick McKeown. On the speedup required for combined input-and output-queued switching. *Automatica*, 35(12), 1999.
[32] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM CCR*, volume 45, 2015.
[33] Arjun Singh et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM*. ACM, 2015.
[34] Brian Towles and William J. Dally. Guaranteed scheduling for switches with configuration overhead. *IEEE/ACM Trans. Networks*, 11(5), 2003.
[35] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
[36] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. Costly circuits, submodular schedules and approximate carathéodory theorems. In *SIGMETRICS*, 2016.
[37] C.-H. Wang and T. Javidi. Adaptive policies for scheduling with reconfiguration delay: An end-to-end solution for all-optical datacenters. *IEEE/ACM Trans. on Networks*, 2017.
[38] Guohui Wang, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, TS Eugene Ng, Michael Kozuch, and Michael Ryan. c-Through: Part-time optics in data centers. In *ACM SIGCOMM*, 2010.
[39] B. Wu and K. L. Yeung. NXG05-6: Minimum delay scheduling in scalable hybrid electronic/optical packet switches. In *IEEE Globecom 2006*, pages 1–5, Nov 2006.
[40] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y Zhao, and Haitao Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. In *SIGCOMM*, 2012.

A PROOF FOR SET OF α'S

LEMMA 3. *Let $\alpha_1, \alpha_2, \ldots,$ be the sorted set of α's selected by Procedure 1. Consider a configuration (M, α) such that $\alpha_k < \alpha < \alpha_{k+1}$. We claim that there is a configuration with higher benefit per unit cost than that of (M, α).*

PROOF. Consider the configuration (M, α) with $\alpha_k < \alpha < \alpha_{k+1}$. Consider all the links (u, v) in M that have more than α_k packets in T^r situated at u and waiting to go to next hop v. Let the set of these links be $M' = \{(u_1, v_1), (u_2, v_2), \ldots, \}$. If M' is empty, then benefit of (M, α_k) is same as that of (M, α) and the lemma trivially holds. Thus, let us assume M' to be non-empty.

For each $(u_i, v_i) \in M'$, consider the packets in T^r situated at u_i waiting to go to the next hop v_i, sort them by their weights, and let $\mathcal{P}_i$ be the set of packets of rank r such that $\alpha_k < r \leq \alpha_{k+1}$. It is easy to see that (M, α)'s benefit is comprised of traversing at least one packet from each of the $\mathcal{P}_i$'s. Below, we claim that benefit per unit cost of (M, α) can be improved by traversing an additional packet from each of the $\mathcal{P}_i$'s, and the lemma follows.

To prove the above claim, observe the following: (i) $\mathcal{P}_1, \mathcal{P}_2, \ldots$ are disjoint, (ii) $|\mathcal{P}_i| = (\alpha_{k+1} - \alpha_k + 1)$, for all i, (iii) Since $\alpha < \alpha_k$, there is at least one packet in each $\mathcal{P}_i$ that is not traversing a hop in (M, α), (iv) each $\mathcal{P}_i$ consists of packets of uniform weight (say, w_i). The second and fourth observations are true because α_k and α_{k+1} are consecutive in the sorted list of α's selected by Procedure 1. Now, the above claim easily follows by showing that if $(B - w)/(\alpha - 1 + \Delta) < B/(\alpha + \Delta)$ then $(B + w)/(\alpha + 1 + \Delta) > B/(\alpha + \Delta)$, for arbitrary values of B, α, w and Δ (in any case, here $w = \sum_k w_k$). □

B PROOF OF LEMMA 2

Proof: Let N_j and W_j be the number and aggregate weight of packets that would traverse a hop during the configuration O_j, if O_j were picked right after $\mathbb{S}$. Then, $\mathcal{B}(O_j, \mathbb{S}) = W_j$ for each j. Now, let N'_j and W'_j be the number and aggregate weight of "new" packets that would traverse a hop during the configuration O_j, if O_j were picked after the sequence $\langle \mathbb{S}, O_1, O_2, \ldots, O_{j-1} \rangle$; here, by *new* packets, we mean the packets that have not traversed a hop in $O_1, O_2, \ldots, O_{j-1}$. We make two observations.

(1) $N'_j \leq N_j$. This follows from the fact that any *new* packet can traverses a hop in O_j when $\langle \mathbb{S}, O_1, O_2, \ldots, O_{j-1} \rangle$ have already been picked, can also traverse a hop when none of the the $O'_i s$ have been picked.
(2) $W'_j \leq W_j$. This follows from the fact that the N_j packets are chosen with optimal weight (as per $g()$ computation on each link in the matching of O_j).

Note that above we do not make any assumptions about how the N'_j packets are picked for routing in the sequence O_j after $\langle \mathbb{S}, O_1, O_2, \ldots, O_{j-1} \rangle$. This remark is important in ensuring the generality of Theorem 1's proof.

Now, the lemma easily follows from the above two observations since $\mathcal{B}(\langle O_1, O_2, \ldots, O_{k'}\rangle, \mathbb{S}) \leq \mathcal{D} \sum_j W'_j$, since $\sum_j W'_j$ is the aggregate weight of all the packets that traverse a hop in $\langle O_1, O_2, \ldots, O_{k'}\rangle$ and $\mathcal{D} \sum_j W'_j$ is the maximum benefit they can confer. ■

C PROOF OF THEOREM 1

Let the Octopus solution be $\langle G_1, G_2, \ldots, G_k\rangle$, and Let a_i be the benefit of configuration G_i, when it was selected. Thus, the total benefit (or objective value)[7] of the Octopus solution $\mathbb{S}$ is $(a_1 + a_2 \ldots + a_k)$. Let the optimal solution of $\langle O_1, O_2, \ldots, O_{k'}\rangle$ and its total optimal benefit be P. To make the configurations O_i or G_i uniform, we assume that even the last configurations G_k and $O_{k'}$ are appended with a reconfiguration delay Δ, and thus the solutions have a total cost of $W + \Delta$. Now, let us consider the i^{th} iteration of Octopus, when Octopus has already selected $\langle G_1, G_2, \ldots, G_{i-1}\rangle$ configurations. We can observe the following. We use $W' = (W+\Delta)$ below.

- The total objective value of the sequence of configurations G_1 to G_{i-1} is $\sum_{j=1}^{i-1} a_j$. Now, since the total objective value of the sequence of configurations $\langle G_1, G_2, \ldots, G_i, O_1, O_2, \ldots, O_{k'}\rangle$ must be at least P (by monotonocity[8] of the objective function), the *benefit* of the optimal sequence $\langle O_1, O_2, \ldots, O_{k'}\rangle$ at *this stage* is at least $P - \sum_{j=1}^{i-1} a_j$, assuming $\sum_{j=1}^{i-1} a_j$ is at most P which is trivially true.
- By Lemma 2,[9] the sum of the benefits of the *individual* configurations O_1 to $O_{k'}$, each considered at *this stage*, is at least $(P - \sum_{j=1}^{i-1} a_j)/\mathcal{D}$. That is,

$$\sum_{j=1}^{k'} \mathcal{B}(O_j, \mathbb{S}_i) \geq (P - \sum_{j=1}^{i-1} a_j)/\mathcal{D},$$

  where $\mathbb{S}_i$ defines the current stage (i^{th} iteration) and is the sequence $\langle G_1, G_2, \ldots, G_i\rangle$.
- By the pigeon hole principle, there must exist a configuration O_l in the optimal sequence whose benefit *per unit cost* is at least $(P - \sum_{j=1}^{i-1} a_j)/(W'\mathcal{D})$, since the total cost of the optimal configurations is at most W'.
- Thus, the next configuration G_i picked by Octopus must have a benefit per unit cost of at least

$$(P - \sum_{j=1}^{i-1} a_j)/(W'\mathcal{D}).$$

  Thus, we have

$$a_i/c_i \geq \frac{1}{W'\mathcal{D}}(P - \sum_{j=1}^{i-1} a_j),$$

  where c_i is the cost of G_i.

[7]Recall that benefit $\mathcal{B}(\mathbb{S},) = \psi(\mathbb{S})$ for any sequence $\mathbb{S}$.
[8]Note that the monotonicity property holds without assuming the fixed routing scheme, and thus, the optimal solution here without loss of any generality.
[9]The remark made in Lemma 2's proof ensures that we do not assume any routing scheme in the optimal solution.

Now, using the above equation, it is easy to show by induction that $(P - \sum_{j=1}^{i} a_j) \leq P(1 - 1/(W'\mathcal{D}))^{c_1+c_2\ldots c_i}$. Thus, for $i = k$, we get

$$(P - \sum_{j=1}^{k} a_j) \leq P(1 - 1/(W'\mathcal{D}))^{c_1+c_2\ldots c_k}.$$

Since $c_1 + c_2 \ldots c_k \leq W'$ and $(1 - 1/(W'\mathcal{D}))^{W'\mathcal{D}} \leq 1/e$ for all $W'\mathcal{D}$, we get

$$\frac{\sum_{j=1}^{k} a_j}{P} \geq (1 - 1/e^{1/\mathcal{D}}).$$

The above analysis however ignore the "truncation" of the last configuration G_k done by the if statement in Octopus; it is easy to see that the impact of that is a factor at most a loss of benefit of Δ time slot, and thus introducing a factor of $\frac{W}{W+\Delta}$.

D PROOF OF THEOREM 3.

We prove Theorem 3, i.e., approximability of Octopus+ for the MHSR problem, by generalizing the proof of Theorem 3. We do this by showing the following: (i) the objective function ψ continues to be monotonic in this new setting, (ii) Lemma 2 still holds, and (iii) the overestimation of benefit function due to backtracking adds only a constant factor of $c/\mathcal{D}$. Of the three tasks above, the second is obvious—as it is easy to see that the argument used in the proof of Lemma 2 continues to hold in the generalized setting of the MHSR problem. We show (i) and (iii) below.

Monotonicity of ψ for Octopus+. Note that the monotonicity of the objective function is used in Theorem 3's proof in making the first observation that $\psi(\texttt{concat}(\mathbb{S}_i, O)) \geq \psi(O)$, where $\mathbb{S}_i$ is the greedy sequence $\langle G_1, G_2, \ldots, G_i\rangle$ and O is the optimal sequence. Recall that the proof of monotonicity of ψ in Lemma 1 was based on the fact that each packet has a unique route; this fact doesn't hold within the MHSR problem's setting. In fact, appending $\mathbb{S}_i$ by O can reduce the objective value to be less than $\psi(O)$ only in the scenarios where a packet x uses different routes in $\mathbb{S}_i$ and O. Recall the assumption that there are only two routes possible for each packet, one of which is a direct route. Now, let us consider the various scenarios for a packet x:

(1) Packet x has reached its final destination in $\mathbb{S}_i$. In this case, benefit due to x in $\mathbb{S}_i$ is already maximum, and is preserved in $\langle \mathbb{S}_i, O\rangle$ which will just annul the routing of packet x in O.
(2) Packet x is still at source in $\mathbb{S}_i$. In this case, the routing of x in $\langle \mathbb{S}_i, O\rangle$ is done purely by O, and thus, the benefit of x in O is preserved in $\langle \mathbb{S}_i, O\rangle$.
(3) Now, the only case that remains is: packet x used a multi-hop route in $\mathbb{S}_i$ and is at an intermediate node y, while it used a direct link in O. In this case, $\langle \mathbb{S}_i, O\rangle$ is still able to preserve the full benefit of O by backtracking x to a direct link in O. This is *the* reason why backtracking was incorporated in Octopus+, to ensure a performance guarantee.

Thus, $\psi(\langle \mathbb{S}_i, O\rangle) \geq \psi(O)$ for all $\mathbb{S}_i$ and O.

Overestimation of a Configuration Benefit. Recall computation of $g(i, j, \alpha)$ values with backtracking in Octopus+. Here, for a packet x with source-destination (s, d) that is at an intermediate node y, we may add appropriate benefit of x to both links (y, next hop of y) as well as (s, d). Now, even if both links exist in the best selected configuration, x would be routed only through

the direct link (s, d)—this scenario suggests overestimation of the benefit of the configuration. It can be easy that such overestimation occurs only for $\mathcal{D} > 2$, and is of the factor of at most (4/3) which occurs in routes of length 3. This explains the factor of c in the theorem.