Scheduling Parallel Optical Switches in Datacenters

Kevin Liang, Litao Qiao, Isaac Keslassy, *Senior Member, IEEE* and Bill Lin, *Senior Member, IEEE*

***Abstract*—We consider the problem of scheduling a single traffic demand matrix D over s parallel optical circuit switches (OCSes) with reconfiguration delays. Our algorithm SPECTRA relies on a three-step approach: DECOMPOSE D into a set of permutations, SCHEDULE these permutations across the switches, then EQUALIZE the loads on the switches. Our evaluation shows SPECTRA vastly outperforms a baseline based on state-of-the-art algorithms, reaching schedule makespans that are shorter on average by a factor of $2.4\times$.**

***Index Terms*—Optical circuit switch, matrix decomposition**

I. INTRODUCTION

THERE has been a significant body of literature motivating and proposing the use of optical circuit switches (OCSes) in datacenters (e.g., [1]–[3]). In comparison with electronic packet switches, OCSes provide much higher bandwidth at much lower power, both of which are important in meeting the rapidly increasing demands on datacenter networks. In addition, a single OCS cannot connect a single input to several outputs at once. Thus, to provide parallelism as well as increased capacity, prior work increasingly considers employing *parallel OCSes* [4]–[8].

Related work. The existing literature mostly minimizes schedule length either assuming a single optical switch, or a multi-stage switch architecture that can be *abstracted as a single switch* [1]–[3]. In particular, these existing works focus on the scheduling of a given traffic demand matrix D onto an OCS by decomposing D into a sequence of configurations of given durations, such that the traffic demands from D are served. A critical difference with an analogous decomposition problem for electronic switches is that OCSes incur a non-trivial reconfiguration delay δ when the *switch configuration* has to change, which adds to the total delay (the *makespan*) in servicing D. Existing works (e.g., [1]–[3]) are effective in solving this scheduling problem with reconfiguration delays for the single OCS case.

LESS [4] is the most relevant work on scheduling a demand matrix D for s parallel OCSes. It splits D into s sub-matrices using a balancing criterion, then schedules each sub-matrix on a separate OCS. However, LESS focuses more on a partial-reconfiguration scheduling algorithm intended for partially-reconfigurable switches. Additional works consider parallel OCSes, but under different assumptions: They (1) achieve a distributed scheduling [5], but with lower matching efficiency; (2) use OCSes to route around failures [6]; (3) extend results to a multi-hop schedule [7], which we do not consider in this paper; and (4) focus on integer-weight decompositions [8].

K. Liang, L. Qiao and B. Lin are with UC San Diego (e-mails: {k3liang, l1qiao,billlin@}ucsd.edu). I. Keslassy is with the Technion and UC Berkeley (isaac@technion.ac.il). This work was partly supported by the Louis and Miriam Benjamin Chair in Computer-Communication Networks and NSF Award No. 2410053.

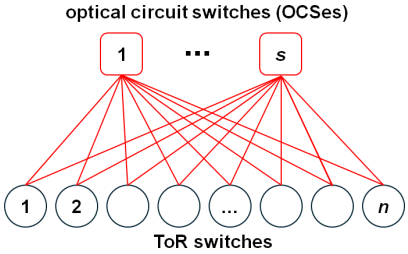

Fig. 1. Datacenter network topology with s parallel OCSes.

Contributions. In this paper, we introduce our SPECTRA algorithm for scheduling D over s parallel switches. A key idea of SPECTRA is to cut the problem into successive sub-problems. SPECTRA relies on a three-step approach: DECOMPOSE D into a set of permutations, SCHEDULE these permutations across the s switches, then EQUALIZE the loads on the switches by splitting the durations of some of the permutations on the most-loaded switches and moving part of their load to the least-loaded ones. We evaluate SPECTRA using a benchmark workload used in past papers. We compare SPECTRA against a BASELINE algorithm that extends the sparsity ideas of LESS [4]. We show that it vastly outperforms BASELINE, with a schedule that is on average shorter by $2.4\times$.

II. PROBLEM FORMULATION

Demand matrix. As Fig. 1 illustrates, we consider a datacenter core of s parallel *optical circuit switches* (OCSes). Each OCS has n input and output ports, connected to the Top-of-Rack (ToR) switches of n datacenter racks. Periodically, a centralized controller computes a new $n \times n$ traffic demand matrix D. Each D_{ij} corresponds to the amount of traffic that needs to be switched from rack i to rack j over the next period.

Switch schedule. Each OCS has a schedule that consists of a sequence of permutations, where each input rack i is connected to a single output rack j. A sequence of k permutations $P_1, P_2, \ldots, P_k$, with corresponding weights $\alpha_1, \alpha_2, \ldots, \alpha_k > 0$ is said to *cover* D if the following is satisfied:

$$\sum_{i=1}^{k} \alpha_i P_i \geq D. \tag{1}$$

In addition, the optical switch needs a *reconfiguration delay* δ before each permutation. Thus, the time needed to schedule the k permutations in a single switch is

$$\sum_{i=1}^{k} (\delta + \alpha_i) = k \cdot \delta + \sum_{i=1}^{k} \alpha_i. \tag{2}$$

Makespan. We now consider the s schedules of the s optical switches. As above, consider some switch h, with $h \in [1, s]$.

Assume that each such switch uses a schedule with k^h permutations P_i^h and their corresponding weights α_i^h, for $i \in \left[1, k^h\right]$. Then, extending Eq. (1), the set of all s switch schedules covers D when

$$\sum_{h=1}^{s} \left[\sum_{i=1}^{k^h} \alpha_i^h P_i^h \right] \geq D. \tag{3}$$

The *makespan* of the s parallel switch schedules is the delay of the longest one, i.e., using Eq. (2),

$$\max_{1 \leq h \leq s} \left[\sum_{i=1}^{k^h} \left(\delta + \alpha_i^h \right) \right]. \tag{4}$$

Objective. The goal of this paper is to find all permutations and weights in the s schedules of the s parallel optical switches, so that they manage to *cover* D (Eq. (3)) while *minimizing the makespan*.

The general minimization problem of the above objective is NP-hard, even with $s = 1$ [9]. Thus, in this paper, we focus on finding a practical algorithm that typically *attains a small makespan in practice*.

III. Algorithms

A. Overview

In this section, we introduce our SPECTRA algorithm (short for Scheduling parallEl Circuit switches for data center TRAffic). As formally defined in §II, SPECTRA finds s schedules of D over s parallel optical switches, such that the schedules *cover* D while attaining a *small makespan*. We start by illustrating SPECTRA's three-step approach using an example, before formally defining each step.

DECOMPOSE. The first step in SPECTRA is to DECOMPOSE a demand matrix D into a set of permutations $\{P_1, P2, \ldots, P_k\}$ with a corresponding set of weights $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ such that their weighted sum *covers* D. i.e., $\sum_{i=1}^{k} \alpha_i P_i \geq D$.

Consider the demand D shown in Fig. 2. Fig. 3 provides an example of how D can be decomposed into 3 permutations, with the corresponding weights 0.61, 0.3, and 0.1. Fig. 3 also shows that their weighted sum indeed covers D. As detailed later in this section, given at most k nonzero elements in any row or column of D, our DECOMPOSE algorithm *guarantees* that exactly k permutations will be generated to cover D. It minimizes the number of reconfiguration delays needed while also reducing the sum duration of the permutations.

SCHEDULE. After the DECOMPOSE step, we next SCHEDULE the k permutations across the s switches. This step is equivalent to scheduling k jobs across s machines, with each α_i representing the amount of time that a job (permutation) P_i needs to run, with added consideration for a reconfiguration delay δ each time a machine (switch) needs to be (re)configured to run a job (switch permutation).

For this step, we first sort the permutations by non-increasing α_i weights, and then we schedule the corresponding P_i in non-increasing-weight-order to the *least* loaded switch each time. Fig. 4 illustrates the SCHEDULE process for the 3 permutations from Fig. 3 across $s = 2$ switches with

$$\mathbf{D} = \begin{pmatrix} \mathbf{0.6} & \mathbf{0.3} & 0 & \mathbf{0.1} \\ 0 & \mathbf{0.61} & \mathbf{0.39} & 0 \\ 0 & \mathbf{0.09} & \mathbf{0.61} & \mathbf{0.3} \\ \mathbf{0.4} & 0 & 0 & \mathbf{0.6} \end{pmatrix}$$

Fig. 2. Example of demand matrix D.

reconfiguration delay $\delta = 0.01$. Initially, both switches are empty, as shown in Fig. 4. We first schedule the permutation with $\alpha_1 = 0.61$ to the first switch, so the load of the first switch becomes $0.61 + 0.01 = 0.62$. We next schedule the permutation with $\alpha_2 = 0.3$ to the second switch since it is empty and hence the least loaded. The load of the second switch becomes $0.3 + 0.01 = 0.31$. Finally, we schedule the last permutations with $\alpha_3 = 0.1$ to the second switch again, since it remains the least loaded of the two switches, making its final load 0.42. The makespan after the SCHEDULE step is $\max(0.62, 0.42) = 0.62$, as shown in Fig. 4(a).

EQUALIZE. As the final step, we aim to EQUALIZE the loads on the switches by splitting the duration (α_i) of a permutation (switch configuration) P_i into *multiple* segments and moving some segments to different switches in order to load-balance the workloads. In particular, our EQUALIZE step iteratively balances the loads between the *most* loaded switch h_{max} and the *least* loaded switch h_{min} by moving a portion of the longest-duration permutation in h_{max} to h_{min}. This is shown in Fig. 4(b), where we split $\alpha_1 = 0.61$ in the first switch into two parts, an $\alpha_1^1 = 0.515$ part that remains in the first switch and an $\alpha_1^2 = 0.095$ part that moves to the second switch, resulting in the final makespan of 0.525, as shown in Fig. 4(b).

Each of the above steps is further detailed below.

B. DECOMPOSE

Support matrix. Given an $n \times n$ matrix D, let $S \in \{0,1\}^{n \times n}$ be its *support* matrix defined by $S_{ij} = 1$ if $D_{ij} > 0$ and $S_{ij} = 0$ otherwise. Let k be the *degree* of D, corresponding to the maximum number of nonzero elements in any row or column in D. By definition, the support matrix S for D will also have a degree of k, corresponding to a maximum of k 1's in any row or column in S. The following property follows from König's Line Coloring Theorem:

Property 1. *k permutations are necessary and sufficient to cover any matrix of degree k.*

DECOMPOSE intuition. As Fig. 4(a) illustrates, the sum of all the schedule delays will include (1) δ times *the number of permutations* (in this example, $0.01 \cdot 3$), and (2) *the sum of the permutation weights* ($0.61 + 0.3 + 0.1 = 1.01$). Thus, we want to minimize both of these components.

(1) First, by Property 1, k permutations can cover D via some $\alpha_1, \alpha_2, \ldots, \alpha_k$ such that $\sum_{i=1}^{k} \alpha_i P_i \geq D$. In our approach, the goal of *minimizing the number of permutations* is to later minimize the number of reconfiguration delays. Therefore, our approach is to decompose D into exactly this minimum number k of permutations.

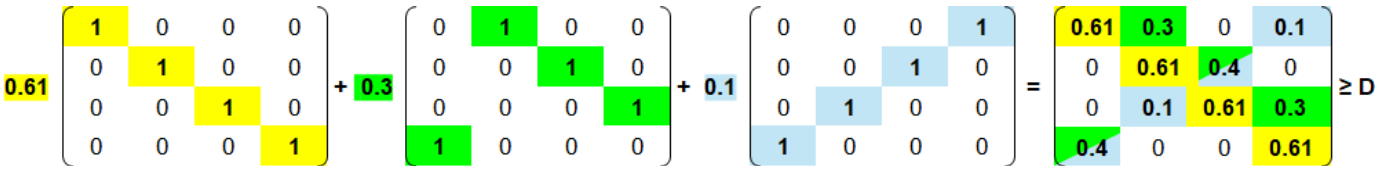

Fig. 3. Example decomposition into $k = 3$ weighted permutations that cover D.

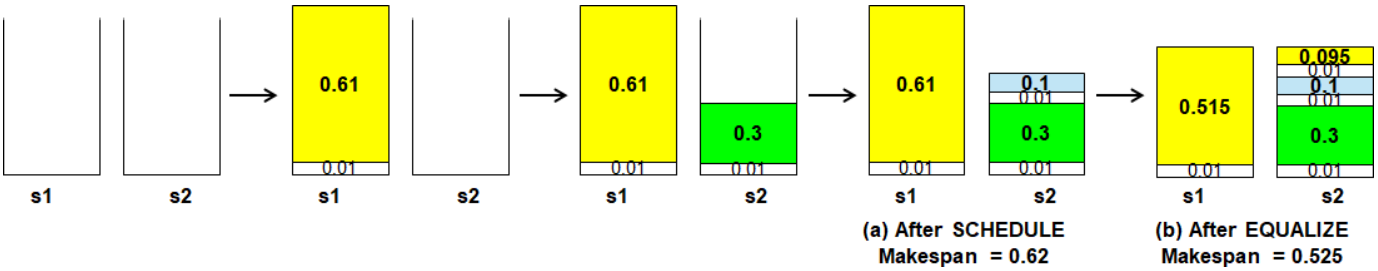

Fig. 4. (a) Scheduling the $k = 3$ permutations across $s = 2$ switches. (b) Equalizing the loads among the switches.

Algorithm 1 DECOMPOSE

1: **Input:** D and δ
2: **Assign:** S = support matrix of D, $D_{rem} = D$, $S_{rem} = S$, $i = 1$, $\mathcal{A} = \{\}$ and $\mathcal{P} = \{\}$
3: **while** $S_{rem} \neq \emptyset$ **do**
4: $P_i \leftarrow$ MWM-Node-Coverage(D_{rem}, S_{rem})
5: $\alpha_i \leftarrow \min_{(a,b),(P_i)_{ab}=1}(D_{rem})_{ab}$
6: $\mathcal{P} \leftarrow \mathcal{P} \cup \{P_i\}$
7: $\mathcal{A} \leftarrow \mathcal{A} \cup \{\alpha_i\}$
8: $D_{rem} \leftarrow D_{rem} - (\alpha_i P_i)$
9: $S_{rem} \leftarrow S_{rem} - (P_i \cap S_{rem})$
10: $i \leftarrow i + 1$
11: $\hat{\mathcal{A}} \leftarrow$ REFINE$(D, \mathcal{A}, \mathcal{P})$
12: **return** $(\hat{\mathcal{A}}, \mathcal{P})$

Algorithm 2 Subroutine REFINE

1: **Input:** D, $\mathcal{A}$ and $\mathcal{P}$
2: $\hat{\mathcal{A}} \leftarrow \mathcal{A}$
3: $D_{rem} \leftarrow D - \left(\sum_{i=1}^{k} \alpha_i P_i\right)$
4: **for** $i = 1$ to k **do**
5: $d \leftarrow \max_{(a,b),(P_i)_{ab}=1}(D_{rem})_{ab}$
6: $\hat{\alpha}_i \leftarrow \hat{\alpha}_i + d$
7: **for** all (a, b) such that $(P_i)_{ab} = 1$ **do**
8: $(D_{rem})_{ab} \leftarrow \max(0, (D_{rem})_{ab} - d)$
9: **return** $\hat{\mathcal{A}}$

(2) However, there may be many possible sets of k permutations that can cover D. Among these, we want to find a set of permutations along with a set of parameters $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ that also *minimizes the total duration* $\sum_{i=1}^{k} \alpha_i$.

DECOMPOSE pseudocode. Alg. 1 shows the pseudocode of our DECOMPOSE algorithm. Given D, we first derive its support matrix S. DECOMPOSE then iteratively generates a set of k permutations $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ that cover S by solving a *maximum weight matching* (MWM) problem *under node coverage constraints* [10] in each iteration.

At a high-level, we track the *remaining traffic* in D and the *remaining uncovered elements* in S after each round, denoted as D_{rem} and S_{rem}, respectively. They are initially $D_{rem} = D$ and $S_{rem} = S$. Based on the updated D_{rem} and S_{rem} in each round, we would like to generate a permutation that satisfies two properties. Based on the above intuition, (1) we want to choose a permutation that is *guaranteed* to reduce the degree of S_{rem} in each round. This way, the algorithm will end after exactly k rounds, generating exactly k permutations, where k is the degree of the initial support matrix S. (2) Second, among the permutations that will reduce the degree of S_{rem} by 1, we want to choose a permutation that will reduce the amount of remaining traffic in D_{rem} as much as possible.

Fortunately, this is a well-known optimization problem called *MWM under node coverage constraints* [10], [11], where efficient polynomial-time algorithms already exist to satisfy our two desired properties. In this formulation, we identify *critical* rows and columns in S_{rem}, which are the rows and columns that have k nonzero entries, where k is the degree of S_{rem}. To ensure that the degree of S_{rem} gets reduced by 1, we *constrain the matching of every critical row or column to a nonzero entry in that row or column*. Among the permutations that satisfy this constrained matching, we use D_{rem} as weights to optimize the MWM. This constrained MWM problem can be solved in polynomial time using a modified Hungarian algorithm [10], which can handle the above coverage constraints. This is shown in Line 4 of Alg. 1.

For the generated P_i, we derive the initial weight α_i by taking the minimum entry $(D_{rem})_{ab}$ where the corresponding entry $(P_i)_{ab}$ is a 1 (Line 5). We then update D_{rem} and S_{rem} at the end of each round accordingly (Lines 8-9), and we repeat the while-loop until all 1-elements in S are covered by a permutation in $\mathcal{P}$. Note that S_{rem} at each round corresponds to the remaining *uncovered* elements of S till that round, and not necessarily to the support of D_{rem}.

REFINE subroutine. After the while-loop in Lines 3-10 of Alg. 1, we are guaranteed that the set of permutations $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ will cover S, but the weighted sum with

Algorithm 3 SCHEDULE

1: **Input:** $\mathcal{A}$, $\mathcal{P}$, s and δ
2: Sort $\mathcal{P}$ by non-increasing weights in $\mathcal{A}$ such that $\alpha_{\sigma(1)} \geq \alpha_{\sigma(2)} \geq \cdots \geq \alpha_{\sigma(k)}$
3: **for** $h = 1$ to s **do**
4: $\quad L_h \leftarrow 0$, $\mathcal{P}^h \leftarrow \{\}$, $\mathcal{A}^h \leftarrow \{\}$
5: **for** $i = 1$ to k **do**
6: $\quad h^* = \text{argmin}_h\, L_h$
7: $\quad \mathcal{P}^{h^*} \leftarrow \mathcal{P}^{h^*} \cup P_{\sigma(i)}$
8: $\quad \mathcal{A}^{h^*} \leftarrow \mathcal{P}^{h^*} \cup \alpha_{\sigma(i)}$
9: $\quad L_{h^*} \leftarrow L_{h^*} + \delta + \alpha_{\sigma(i)}$
10: $\mathcal{S} \leftarrow \{(\mathcal{A}^1, \mathcal{P}^1), (\mathcal{A}^2, \mathcal{P}^2), \ldots, (\mathcal{A}^s, \mathcal{P}^s)\}$
11: **return** $\mathcal{S}$

the corresponding $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ may not necessarily cover D. This is because in Line 5, we are taking the *minimum* value in D_{rem} covered by P_i, but our stopping criterion is just that the initial support matrix S is covered by the set of permutations. The role of the final REFINE step in Line 11 of Alg. 1 is to refine the set of weights to ensure that the weighted sum indeed covers D.

Given a set of k permutations $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ that covers S, we could readily derive a set of new weights $\hat{\mathcal{A}} = \{\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_k\}$ such that the weighted sum covers D, by solving the following linear programming (LP) problem:

$$\min \sum_{i=1}^{k} \hat{\alpha}_i, \qquad \text{s.t. } \sum_{i=1}^{k} \hat{\alpha}_i P_i \geq D. \tag{5}$$

Instead, starting from the initial weights $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ from Lines 3-10, we greedily increase the weights to a new set of weights $\hat{\mathcal{A}}$ such that the weighted sum covers D (see Alg. 2). We find that in practice this greedy approach works just as well as solving the LP.

C. SCHEDULE

After the DECOMPOSE step, we have a set of permutations $\mathcal{P}$ and their corresponding weights $\hat{\mathcal{A}}$. In the SCHEDULE step, we want to assign each of these permutations to one of the s switches to minimize the makespan (Eq. (4)). This is equivalent to the makespan minimization problem on identical parallel machines, for which the *Longest Processing Time (LPT) First* algorithm [12] is a well-known and effective greedy heuristic. We adopt it for our SCHEDULE step (Alg. 3).

SCHEDULE pseudocode. In this greedy algorithm, we first sort the permutations by non-increasing weights, as shown in Line 2. We then initialize the load L_h for all switches to 0 with no permutations assigned to switch h yet, as shown in Lines 3-4. Then in Lines 5-9, following a non-increasing order of weights, we assign in a greedy manner the corresponding permutation to the least loaded switch (Line 6), add the permutation to the set of permutations already assigned to that switch (Lines 7-8), and update the load of that switch, including the reconfiguration delay δ (Line 9). In the end, the procedure returns the schedule $\mathcal{S}$ of the s switches in the form of weights and permutations $(\mathcal{A}^h, \mathcal{P}^h)$ per switch.

Algorithm 4 EQUALIZE

1: **Input:** $\mathcal{S} = \{(\mathcal{A}^1, \mathcal{P}^1), (\mathcal{A}^2, \mathcal{P}^2), \ldots, (\mathcal{A}^s, \mathcal{P}^s)\}$, s and δ
2: **for** $h = 1$ to s **do**
3: $\quad L_h \leftarrow \sum_{i=1}^{k^h} (\delta + \alpha_i^h)$
4: cont $\leftarrow$ TRUE
5: **while** cont **do**
6: $\quad h_{max} = \text{argmax}_h\, L_h$
7: $\quad h_{min} = \text{argmin}_h\, L_h$
8: $\quad$ **if** $(L_{h_{max}} - L_{h_{min}}) > \delta$ **then**
9: $\quad\quad \mu \leftarrow (L_{h_{max}} + L_{h_{min}} + \delta)/2$
10: $\quad\quad z \leftarrow \text{argmax}_i\, \alpha_i^{h_{max}}$
11: $\quad\quad$ **if** $\alpha_z^{h_{max}} > (L_{h_{max}} - \mu)$ **then**
12: $\quad\quad\quad \tau \leftarrow L_{h_{max}} - \mu$
13: $\quad\quad\quad \alpha_z^{h_{max}} \leftarrow \alpha_z^{h_{max}} - \tau$
14: $\quad\quad\quad$ Copy permutation $P_z^{h_{max}}$ to $\mathcal{P}^{h_{min}}$
15: $\quad\quad\quad$ Add τ as corresponding weight to $\mathcal{A}^{h_{min}}$
16: $\quad\quad\quad L_{h_{max}} \leftarrow L_{h_{max}} - \tau$
17: $\quad\quad\quad L_{h_{min}} \leftarrow L_{h_{min}} + \delta + \tau$
18: $\quad\quad$ **else**
19: $\quad\quad\quad$ cont $\leftarrow$ FALSE
20: $\quad$ **else**
21: $\quad\quad$ cont $\leftarrow$ FALSE
22: $\hat{\mathcal{S}} \leftarrow$ updated $\{(\mathcal{A}^1, \mathcal{P}^1), (\mathcal{A}^2, \mathcal{P}^2), \ldots, (\mathcal{A}^s, \mathcal{P}^s)\}$
23: **return** $\hat{\mathcal{S}}$

D. EQUALIZE

Finally, Fig. 4(b) illustrates how we can reduce the makespan by redistributing some of the switching load from one switch to another.

EQUALIZE pseudocode. The pseudocode for this EQUALIZE step is shown in Alg. 4. We first compute the loads L_h for each switch after the SCHEDULE step in Lines 2-3. We then iteratively equalize the loads in Lines 5-21 by moving a portion of the load from the most loaded switch h_{max} to the least loaded switch h_{min}. Lines 6-7 identify h_{max} and h_{min}. If the difference between them is more than δ, we compute the target load μ to be the average of the two loads that includes a δ delay when a portion of h_{max} is moved to h_{min} (Lines 8-9). In the most loaded switch h_{max}, we identify the index z of the longest-duration permutation in h_{max} in Line 10. If the weight of this permutation $\alpha_z^{h_{max}}$ is more than the difference between the current and target load of h_{max}, then we move the difference τ from $\alpha_z^{h_{max}}$ in h_{max} to h_{min} by copying the corresponding permutation $P_z^{h_{max}}$ to switch h_{min} and assigning a weight of τ to it (Lines 11-15). We then adjust the loads of the two switches accordingly in Lines 16-17. The iterations continue as long as load balancing is possible.

IV. EVALUATIONS

A. Settings

We compare SPECTRA with previous state-of-the-art algorithms that were designed for similar objectives to understand the impact of different characteristics of D on SPECTRA. To do so, we use a standard benchmark that has already been used in the evaluations of several papers.

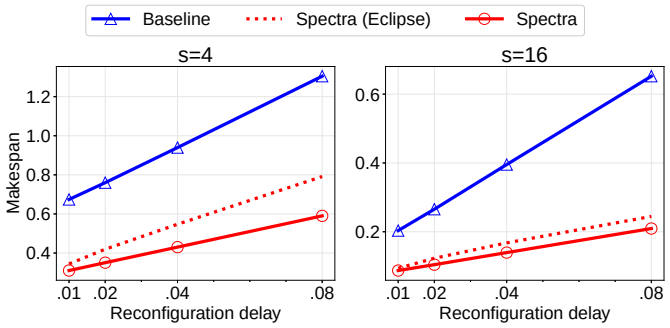

Fig. 5. *Benchmark traffic*: SPECTRA vs. BASELINE with varying δ. SPECTRA achieves a lower makespan across the span of δ values.

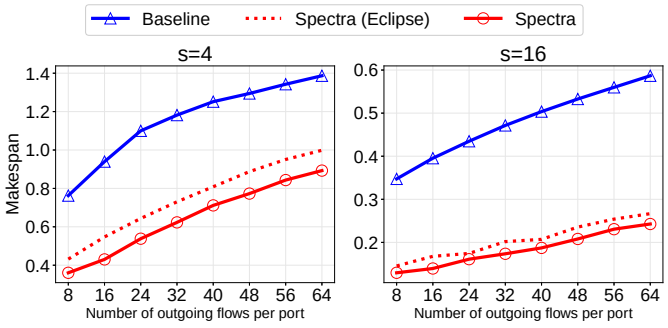

Fig. 6. *Sensitivity to sparsity*: SPECTRA vs. BASELINE with varying number m of flows for $\delta = 0.04$. SPECTRA outperforms across a wide range.

Benchmark. We use the standard benchmark in the field [1], [2], [4]. It builds a 100×100 demand D that consists of $m = 16$ random flows per source port, including 4 large flows, which evenly split 70% of the bandwidth, and 12 small flows, which evenly split 30%, to reflect the sparse and skewed datacenter traffic. Each flow corresponds to a permutation matrix, and the sum of flows yields D. Finally, the nonzero demands are perturbed with noise that has a standard deviation of 0.3% of the link bandwidth.

BASELINE. We want to evaluate our SPECTRA algorithm against the best state-of-the-art optical switch scheduling algorithms. Unfortunately, most algorithms in the literature consider a single optical switch and cannot be applied to scheduling across s parallel optical switches. The closest to our work is LESS [4], which considers scheduling a demand matrix D across s parallel optical switches by splitting D into s sub-workload matrices $D_1, \ldots, D_s$, each D_i to be independently scheduled on switch i. The splitting objective is to maximize the *sparsity* in these sub-workload matrices by minimizing the sum of nonzero elements across $D_1, \ldots, D_s$. While LESS uses a partial-reconfiguration scheduling algorithm intended for partially-reconfigurable switches, we adopt its sparsity-based approach as our BASELINE for comparisons. For an apples-to-apples comparison, we also use our DECOMPOSE algorithm to schedule the sub-workload matrices in BASELINE, since LESS does not provide a similar decomposition algorithm.

SPECTRA (ECLIPSE). Since ECLIPSE [1] is considered the state-of-the-art in decomposition with reconfiguration delays, we also compare against a SPECTRA version that uses ECLIPSE [1] for the DECOMPOSE step in place of our decomposition algorithm described in §III-B.

Runs. We conduct our experiments on a computer with a 3.7 GHz AMD Ryzen Threadripper 3970X Processor and 128 GB of memory, running on Ubuntu 20.04.6 LTS. The runtimes for SPECTRA are from under 1 ms to tens of ms. Each datapoint is the average of 50 runs with random matrices.

B. Results

Fig. 5 compares SPECTRA against BASELINE on the benchmark workload. SPECTRA vastly outperforms BASELINE, with a $2.4\times$ shorter schedule on average. Using ECLIPSE for the DECOMPOSE step always yields worse results.

Fig. 6 analyzes the sensitivity of BASELINE and SPECTRA to a varying number m of outgoing flows per port, and therefore to a varying sparsity of the matrix D, given $\delta = 0.04$. SPECTRA is flexible enough to adapt to these differing sparsity patterns, again vastly outperforming BASELINE. Using an ECLIPSE decomposition step never helps SPECTRA.

V. CONCLUSION

In this paper, we showed how the increasing network demands in datacenters could be serviced by a set of OCSes, in which we want to minimize the makespan. We then explained how SPECTRA can schedule a demand matrix D over s parallel switches using three steps: DECOMPOSE, SCHEDULE and EQUALIZE. Our evaluation of SPECTRA shows that it vastly outperforms BASELINE by an average factor of $2.4\times$. As future work, we intend to explore how SPECTRA can be applied to AI collective communication patterns. Its focus on makespan minimization fits the AI needs for a low collective completion time that depends on the last packet.

REFERENCES

[1] S. Bojja Venkatakrishnan, M. Alizadeh, and P. Viswanath, "Costly circuits, submodular schedules and approximate carathéodory theorems," *Queueing Systems*, vol. 88, no. 3, pp. 311–347, 2018.
[2] H. Liu *et al.*, "Scheduling techniques for hybrid circuit/packet networks," in *ACM CoNEXT*, 2015, pp. 1–13.
[3] N. Farrington *et al.*, "Helios: a hybrid electrical/optical switch architecture for modular data centers," in *ACM SIGCOMM*, vol. 40, no. 4, 2010, pp. 339–350.
[4] L. Liu, J. J. Xu, and M. Singh, "LESS: A matrix split and balance algorithm for parallel circuit (optical) or hybrid data center switching and more," in *IEEE/ACM Utility and Cloud Computing (UCC)*, 2019.
[5] C. Liang *et al.*, "NegotiaToR: towards a simple yet effective on-demand reconfigurable datacenter network," in *ACM SIGCOMM*, 2024.
[6] Y. Zu, A. Ghaffarkhah, H.-V. Dang, B. Towles, S. Hand, S. Huda, A. Bello, A. Kolbasov, A. Rezaei, D. Du *et al.*, "Resiliency at scale: Managing {Google's}{TPUv4} machine learning supercomputer," in *Usenix NSDI*, 2024, pp. 761–774.
[7] F. De Marchi, J. Li, Y. Zhang, W. Bai, and Y. Xia, "Unlocking superior performance in reconfigurable data center networks with credit-based transport," in *ACM SIGCOMM*, 2025, pp. 842–860.
[8] V. Addanki *et al.*, "Vermilion: A traffic-aware reconfigurable optical interconnect with formal throughput guarantees," *arXiv preprint arXiv:2504.09892*, 2025.
[9] X. Li and M. Hamdi, "On scheduling optical packet switches with reconfiguration delay," *IEEE JSAC*, vol. 21, no. 7, pp. 1156–1164, 2003.
[10] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*, ser. Algorithms and Combinatorics. Berlin: Springer, 2003, vol. 24, no. 2.
[11] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
[12] P. Bergé, M. Chaikovskaia, J.-P. Gayon, and A. Quilliot, "Approximation algorithms for job scheduling with reconfigurable resources," *arXiv preprint arXiv:2401.00419*, 2023.