Best First Fit (BFF): An Approach to Partially Reconfigurable Hybrid Circuit and Packet Switching

Liang Liu*, Long Gong*, Sen Yang*, Jun (Jim) Xu† and Lance Fortnow†
School of Computer Science, Georgia Institute of Technology
Email: *{lliu315, gonglong, sen.yang}@gatech.edu, †{jx, fortnow}@cc.gatech.edu

***Abstract*—Hybrid switching for data center networks (DCN) has received considerable research attention recently. A hybrid-switched DCN employs a much faster circuit switch that is reconfigurable with a nontrivial cost, and a much slower packet switch, to interconnect its racks of servers. The research problem is, given a traffic demand (between the racks), how to properly schedule the circuit switch so that it removes most of the traffic demand, leaving little for the slower packet switch to handle. All existing solutions make a convenient but unnecessarily restrictive assumption that when the circuit switch changes from one configuration to another, all input ports have to stop data transmission during the reconfiguration period. However, the circuit switch can usually readily support partial reconfiguration in the following sense: Only the input ports affected by the reconfiguration need to pay a reconfiguration delay, while unaffected input ports can continue to transmit data during the reconfiguration. In this work, we propose BFF (best first fit), the first solution to exploit this partial reconfigurability in hybrid-switched DCNs. BFF not only significantly outperforms but also has much lower computational complexity than the state of the art solutions that do not exploit this partial reconfigurability.**

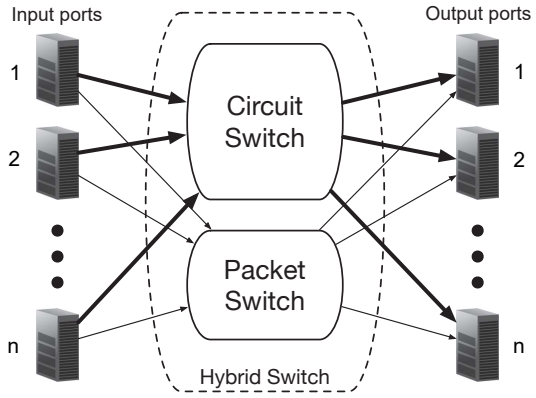

Fig. 1. Hybrid Circuit and Packet Switch

I. Introduction

Fueled by the phenomenal growth of cloud computing services, data center networks (DCN) continue to grow relentlessly both in size, as measured by the number of racks of servers it has to interconnect, and in speed, as measured by the amount of traffic it has to transport per unit of time from/to each rack [1]. A traditional data center network architecture typically consists of a three-level multi-rooted tree of switches that start, at the lowest level, with the Top-of-Rack (ToR) switches that each connects a rack of servers to the network [2]. However, such an architecture has become increasingly unable to scale with the explosive growth in both the size and the speed of the DCN, as we can no longer increase the transporting and switching capabilities of the underlying commodity packet switches without increasing their costs significantly.

A. Hybrid Circuit and Packet Switching

A cost-effective solution approach to this scalability problem, called hybrid DCN architecture, has received considerable research attention in recent years [3], [4], [5]. In a hybrid data center, shown in Figure 1, n racks of computers on the left hand side (LHS) are connected by both a circuit switch and a packet switch to n racks on the right hand side (RHS). Note that racks on the LHS is an identical copy of those on the RHS; however we restrict the role of the former to only transmitting data and refer to them as *input ports*, and restrict the role of the latter to only receiving data and refer to them as *output ports*. The purpose of this duplication (of racks) and role restrictions is that the resulting hybrid data center topology can be modeled as a bipartite graph.

Each switch transmits data from input ports (racks on the LHS) to output ports (racks on the RHS) according to the configuration (modeled as a bipartite matching) of the switch at the moment. The circuit switch is usually an optical switch [3], [6], [7], and the packet switch is an electronic switch. Hence the circuit switch is typically an order of magnitude or more faster than the packet switch. For example, the circuit and packet switches might operate at the respective rates of 100 Gbps and 10 Gbps per port. The flip side of the coin however is that the circuit switch incurs a nontrivial reconfiguration delay δ when the switch configuration (matching) has to change. Depending on the underlying technology of the circuit switch, δ can range from tens of microseconds to tens of milliseconds [3], [6], [7], [8], [9].

The following hybrid circuit and packet switching problem naturally arises in any hybrid DCN architecture: Given a traffic demand matrix D from input ports to output ports, how to schedule the circuit switch so that it removes (*i.e.*, transmit) the vast majority of the traffic demand from D, leaving a residue demand matrix that is small enough for the packet switch to handle.

B. Partial Reconfigurability

All existing works on hybrid switching solve this problem based on the following convenient assumption: When the circuit switch changes from one configuration to another, all input ports have to stop data transmission during the reconfiguration period (of duration δ), including those input ports that pair with the same output ports during both configurations. This is however an outdated and unnecessarily restrictive assumption because all electronics or optical technologies underlying the circuit switch can readily support *partial reconfiguration* in the following sense: Only the input ports affected by the reconfiguration need to pay a reconfiguration delay δ, while unaffected input ports can continue to transmit data during the reconfiguration. For example, in cases where free-space optics is used as the underlying technology (*e.g.*, in [10], [11]), only each input port affected by the reconfiguration needs (to rotate its micro-mirror) to redirect its laser beam towards its new output port and incur reconfiguration delay.

In this work, we propose Best First Fit (BFF), the first (to the best of our knowledge) hybrid switching solution that exploits the partial reconfiguration capability of the circuit switch for performance gains. We will explain in §III that, with the partial reconfiguration capability, the scheduling of the circuit switch only (i.e., without a packet switch) can be modeled as an Open Shop Scheduling (OSS) [12], [13], [14]. As OSS is in general NP-hard [15], BFF is a greedy heuristic solution to it. More specifically, BFF tries to match, through the circuit switch, input ports with output ports as soon as they become available (*i.e.*, after their previous transmissions are over) in the following greedy manner: Each available output port attempts to match with the *best* available input port (*i.e.*, the one with the largest amount of traffic to send to the output port) *at the moment*, and vice versa. Ironically, whereas BFF would be a poor solution for this OSS problem of scheduling circuit switch only, it is a superb solution for hybrid switching (i.e., where there is also a packet switch). We will explain the reason behind this fact in Section III-B. As we will show in §IV that, compared to Eclipse, the state of the art solution to the traditional (*i.e.*, without the partial reconfiguration capability) hybrid switching problem, BFF has not only much better (throughput) performance, but also roughly three orders of magnitude shorter execution times (when $n = 100$ racks). In other words, with this partial reconfiguration capability, the hybrid switch can "work much less hard" to arrive at a schedule that is even better.

The rest of the paper is organized as follows. In §II, we describe the system model and the design objective of this hybrid switch scheduling problem in detail. In §III, we present our solution BFF. In §IV, we evaluate the performance of our solution against Eclipse *and Eclipse++*. Finally, we describe related work in §V and conclude the paper in §VI.

II. System Model and Problem Formulation

In this section, we formulate the problem of hybrid circuit and packet switching precisely. We first specify the aforementioned traffic demand matrix D precisely. Borrowing the term virtual output queue (VOQ) from the input-queued crossbar scheduling literature [16], we refer to, the set of packets that arrive at input port i and are destined for output j, as VOQ(i, j). Each matrix element $D(i, j)$ is the amount of VOQ(i, j) traffic, within a scheduling window, that needs to be scheduled for transmission by the hybrid switch.

It was *effectively* assumed, in all prior works on hybrid switching except Albedo [17] (to be discussed in § V-A), that the demand matrix D is precisely known before the computation of the circuit switch schedule begins (say at time t). Consequently, all prior hybrid switching algorithms except Albedo [17] perform only batch scheduling of this D. In other words, given a demand matrix D, the schedules of the circuit and the packet switches are computed before the transmissions of the batch (*i.e.*, traffic in D) actually happen. Our BFF algorithm also assumes that D is precisely known in advance and is designed for batch scheduling only. Since batch scheduling is offline in nature (*i.e.*, requires no irrevocable online decision-making), BFF algorithm is allowed to "travel back in time" and modify the schedules of the packet and the circuit switches as needed.

In this work, we study this problem of hybrid switch scheduling under the following standard formulation that was introduced in [18]: to minimize the amount of time for the circuit and the packet switches working together to transmit a given traffic demand matrix D. We refer to this amount of time as *transmission time* throughout this paper. An alternative formulation, used in [19], is to maximize the amount of traffic that the hybrid switch can transmit within a scheduling window of a fixed duration. These two formulations are roughly equivalent, as mathematically the latter is roughly the dual of the former.

As stated earlier, the circuit switch being partially reconfigurable offers considerable scheduling flexibility, leading to much lower computational complexities for computing a schedule and higher throughputs of the hybrid switch. We now formulate the operational constraints of a partially reconfigurable circuit switch precisely. Its configurations (schedules) over time can be represented by an $n \times n$ matrix process $S(t) = \big(s_{ij}(t)\big)$, where for any given time t, $S(t)$ is a $0-1$ (sub-matching) matrix that encodes the connections between input ports and the output ports at time t. More specifically, $s_{ij}(t) = 1$ if input port i is connected to output j at time t, and $s_{ij}(t) = 0$ otherwise. After an input port i stops transmitting traffic to an output port j, it has to wait at least a reconfiguration delay δ before starting transmitting traffic to another output port. As mentioned earlier, the good thing about the switch being partial reconfigurable is that no other input port needs to stop its ongoing transmission as a result of this configuration change.

III. Best First Fit (BFF) Algorithm

Since BFF is a greedy heuristic solution to the Open Shop Scheduling (OSS) [12], [13], [14], we first provide some background on OSS in §III-A. Then in §III-B, we introduce a family of heuristic OSS algorithms, called LIST (*i.e.,* list

scheduling), to which BFF belongs. Finally in § III-C, we describe the detailed design of BFF.

A. Background on Open-Shop Scheduling

When partial reconfiguration is allowed, there is no packet switch, and only direct routing is considered (*i.e.*, all circuit-switched data packets reach their respective final destinations in one-hop), scheduling the circuit switch alone can be converted into the classical *open-shop scheduling problem* (OSS) [12], [13], [14]. In an OSS problem, there are a set of N jobs, a set of m machines, and a two-dimensional table specifying the amount of time (could be 0) that a job must spend at a machine to have a certain task performed. The scheduler has to assign jobs to machines in such a way, that at any moment of time, no more than one job is assigned to a machine and no job is assigned to more than one machine. The mission is accomplished when every job has all its tasks performed at respective machines. The OSS problem is to design an algorithm that minimizes, to the extent possible, the makespan of the schedule, or the amount of time it takes to accomplish the mission. In this circuit switching (only) problem, input ports are jobs, output ports are machines, each VOQ(i, j) is a task that belongs to job i and needs to be performed at machine j for the amount of time $D(i, j)$. In OSS, a machine may need some time to reconfigure between taking on a new job, which corresponds to the reconfiguration delay δ in circuit switching. The OSS problem is in general NP-hard [15], so only heuristic or approximate solutions to it [20], [21], [22] exist that run in polynomial time. Finally, we emphasize that this circuit switching (only) problem has nothing to do with *concurrent OSS* [23], which allows any job (input port) to be concurrently processed by multiple machines (output ports) at the same time, because in this problem, an input port is not allowed to transmit to multiple output ports at the same time.

There are two types of OSS problems: one that allows a task to be preempted at a machine for another task (*i.e.*, preemptive) and one that does not (*i.e.*, non-preemptive). In this circuit switch (only) scheduling problem, preemption means that the traffic in a VOQ(i, j) is split into multiple bursts to be sent over multiple noncontiguous connections between i and j. For this circuit switching (only) problem, non-preemptive solutions generally do not perform worse because each preemption costs us a nontrivial reconfiguration delay δ, and such preemption costs can hardly be compensated by the larger solution space and more scheduling flexibility that preemptive solutions can provide.

B. LIST: A Family of Heuristics

LIST (list scheduling) is a well-known family of polynomial-time heuristic OSS algorithms [20], [21], [22]. LIST starts by attempting to assign an available job (*i.e.*, not already being worked on by a machine) to one of the available machines on which the job has a task to perform, according to a machine preference criterion (can be job-specific and time-varying). If multiple jobs are competing for the same machine, one of the jobs is chosen according to a job preference criterion (can be machine-specific and time-varying). After all initial assignments are made, the scheduler "sits idle" until a task is completed on a machine, in which case both the corresponding job and the machine become available. Once a machine becomes available, any available job that has a task to be performed on the machine can compete for the machine.

Our BFF algorithm is an adaptation of a non-preemptive LIST algorithm [22] that uses LPT (longest processing time) as the preference criterion for both the machines and the jobs. In LPT LIST, whenever multiple jobs compete for a machine, the machine picks the "most time-consuming task", *i.e.*, the job that takes the longest time to finish on the machine; whenever a job has multiple available machines to choose from, it chooses among them the machine that has the "most time-consuming task" to perform on the job. In other words, LPT gives preference to longer tasks, whether a machine is choosing jobs or a job is choosing machines. LPT is a perfect match for our problem, because with a packet switch to "sweep clean" all short tasks (*i.e.*, tiny amounts of remaining traffic left over in VOQs by the circuit switch), the circuit switch can afford to focus only on a comparatively small number of long tasks.

However, it is by no means obvious that adapting any LIST algorithm would be a good idea for this hybrid switching problem. In fact, no algorithm in the LIST family, including LPT LIST, is a good fit for the problem of circuit switching only (*i.e.*, where there is not a packet switch), when the circuit switch is partially reconfigurable [13]. In particular, it was shown in [13] that, whenever a scheduling algorithm from the LIST family is used, whether the circuit switch is partially reconfigurable or not makes almost no difference in the the performance (measured by transmission time) of the resulting schedule. This is because, without the help from a packet switch, the circuit switch would have to "sweep clean" the large number of short tasks (VOQs) all by itself, and each such short task costs the circuit switch a reconfiguration delay δ that is significant compared to its processing (transmission) time. To the best of our knowledge, BFF is the first time that a LIST algorithm is adapted for hybrid switching.

C. BFF Algorithm

As explained earlier, BFF is an adaptation of the LPT LIST algorithm for open-shop scheduling. There are two differences between BFF and LPT LIST. First, at the beginning of the scheduling (*i.e.*, $t = 0$), when all jobs and all machines are available, BFF runs a maximum weighted matching (MWM) algorithm [24] to obtain the heaviest (*w.r.t.* to their weights D) initial matching between jobs and machines. BFF does not use LPT LIST for this initialization step because it would likely result in a sub-optimal (*i.e.*, lighter in weight) matching to start with. Second, BFF terminates when the remaining demand D_{rem} becomes small enough for the packet switch to handle. Here the remaining demand matrix D_{rem} denotes what remains of the traffic demand (matrix) after we subtract from D the amounts of traffic to be served by the circuit switch according

Algorithm 1: Action taken by BFF after a machine is done with a job.

When input port i finishes transmitting VOQ(i,j) to output port j at time τ:

1 Output_Seek_Pairing(j,τ);
2 Input port i reconfigures during $[\tau,\tau+\delta]$;
3 Input_Seek_Pairing$(i,\tau+\delta)$;

Procedure *Output_Seek_Pairing*(j,t)

4 Update D_{rem};
5 **if** $\{l \in I_a \mid D_{rem}(l,j) > 0\} \neq \phi$ **then**
6 $l = \arg\max_u D_{\text{rem}}(u,j)$;
7 Connect output j with input l;
8 **else**
9 $O_a \leftarrow O_a \cup \{j\}$;
10 **end**

Procedure *Input_Seek_Pairing*(i,t)

11 Update D_{rem};
12 **if** $\{j \in O_a \mid D_{rem}(i,j) > 0\} \neq \phi$ **then**
13 $j = \arg\max_v D_{\text{rem}}(i,v)$;
14 Connect input i with output j;
15 **else**
16 $I_a \leftarrow I_a \cup \{i\}$;
17 **end**

to the previous actions, *i.e.,* those computed in the pervious iterations.

In BFF, for each input port i_1, the task of deciding with which outputs the input port i_1 should be matched with over time is almost independent of that for any other input port i_2. Hence, to describe BFF precisely, it suffices to describe the actions taken by the scheduler after a job i gets its task performed at a machine j (*i.e.*, after input port i transmits all traffic in VOQ(i,j), in the amount of $D(i,j)$, to output port j).

We do so in Algorithm 1. Suppose the machine j is done with the job i at time τ. Then machine (output port) j immediately looks to serve another job by calling "Output_Seek_Pairing(j,τ)" (Line 1 in Algorithm 1). The job (input port) i, on the other hand, is not ready to be performed on another machine (output port) until $\tau+\delta$ (*i.e.*, after a reconfiguration delay), so it calls "Input_Seek_Pairing$(i,\tau+\delta)$" (Line 3 in Algorithm 1). However, since in this batching scheduling setting, the whole schedule $S(t)$ is computed before any transmission (according to the schedule) can begin, input port i knows which machine it will be paired with at time $\tau+\delta$. Hence input port i can start reconfiguring to pair with that machine at time τ (Line 3 in Algorithm 1) so that the actual transmission can start at time $\tau+\delta$.

In Algorithm 1, I_a and O_a denote the sets of available jobs (input ports) and of available machines (output ports) respectively. Clearly, Procedure "Output_Seek_Pairing( )" (Lines 4 through 10) follows the LPT (longest processing time first) preference criteria: It tries to identify, for the machine (output port) j, the job (input port) that brings with it the largest task, among the set of available jobs I_a. Similar things can be said about procedure "Input_Seek_Pairing( )" (Lines 11 through 17). In other words, each output port, at the very *first* moment it becomes available (to input ports), attempts to match with the *best* input port (*i.e.*, the one with the largest amount of work for it to do), and vice versa. Therefore, we call our algorithm BFF (Best First Fit).

Computational Complexity: In addition to the $O(n^{5/2}\log B)$ complexity needed to obtain an MWM (using [24]) at the very beginning, with a straightforward implementing using a straightforward data structure, BFF has a computational complexity of $O(Kn^2)$, where K is the average number of times each input port needs to reconfigure over time, B is the largest entry in the demand matrix D, W is the maximum row/column sum of the demand matrix. Hence the overall complexity of BFF is $O(Kn^2+n^{5/2}\log B)$, which is asymptotically smaller than that of Eclipse, which is $O(Kn^{5/2}\log n\log B)$ (see Table I). Empirically, BFF runs about three orders of magnitude faster than Eclipse when $n=100$, as will be shown in §IV-E.

TABLE I
COMPARISON OF TIME COMPLEXITIES

Algorithm	Time Complexity
Eclipse	$O(Kn^{5/2}\log n\log B)$
Eclipse++	$O(WKn^3(\log K+\log n)^2)$
BFF	$O(Kn^2+n^{5/2}\log B)$

Wondering whether allowing indirect routing can bring further performance improvements, we have made several attempts at combining indirect routing with BFF. However, we found this direction not promising for two reasons. First, BFF leaves little "slack" in the schedule for the indirect routing to gainfully exploit. Second, any extension for exploiting indirect routing would increase the computational complexity of BFF considerably.

IV. EVALUATION

In this section, we evaluate the performance of our solution BFF, and compare it with that of the state of the art algorithm Eclipse, under various system parameter settings and traffic demands. We do not compare our solutions with Solstice [18] (to be described in §V-A) in these evaluations, since Solstice was shown in [19] to perform worse than Eclipse in all simulation scenarios. For all these comparisons, we use the same performance metric as that used in [18]: the total time needed for the hybrid switch to transmit the traffic demand D.

A. Traffic Demand Matrix D

For our simulations, we use the same traffic demand matrix D as used in other hybrid scheduling works [18], [19]. In this matrix, each row (or column) contains n_L large equal-valued elements (large input-output flows) that as a whole account for c_L (percentage) of the total workload to the row (or

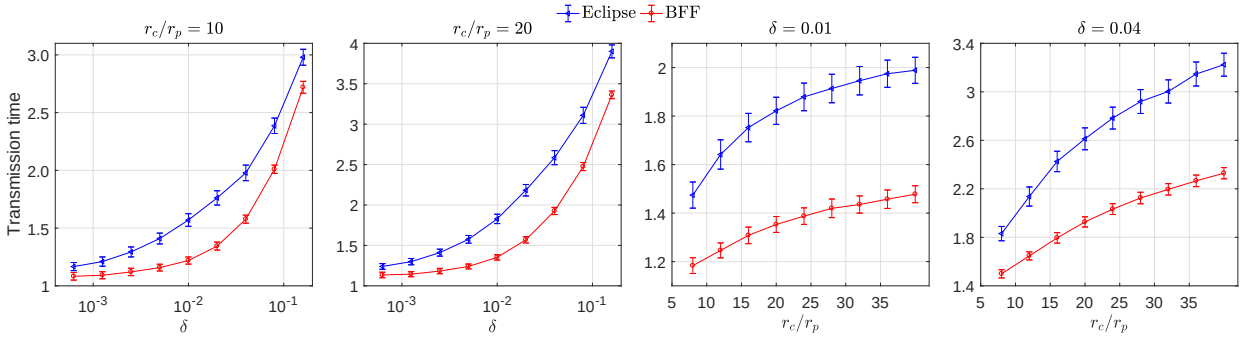

Fig. 2. Performance comparison of Eclipse and BFF under different system setting

column), n_S medium equal-valued elements (medium input-output flows) that as a whole account for the rest $c_S = 1 - c_L$ (percentage), and noises. Roughly speaking, we have

$$D = (\sum_{i=1}^{n_L} \frac{c_L}{n_L} P_i + \sum_{i=1}^{n_S} \frac{c_S}{n_S} P'_i + \mathcal{N}_1) \times 90\% + \mathcal{N}_2 \quad (1)$$

where P_i and P'_i are random $n \times n$ matching (permutation) matrices.

The parameters c_L and c_S control the aforementioned skewness (few large elements in a row or column account for the majority of the row or column sum) of the traffic demand. Like in [18], [19], the default values of c_L and c_S are 0.7 (*i.e.*, 70%) and 0.3 (*i.e.*, 30%) respectively, and the default values of n_L and n_S are 4 and 12 respectively. In other words, in each row (or column) of the demand matrix, by default the 4 large flows account for 70% of the total traffic in the row (or column), and the 12 medium flows account for the rest 30%. We will also study how these hybrid switching algorithms perform when the traffic demand has other degrees of skewness by varying c_L and c_S.

As shown in Equation (1), we also add two noise matrix terms $\mathcal{N}_1$ and $\mathcal{N}_2$ to D. Each nonzero element in $\mathcal{N}_1$ is a Gaussian random variable that is to be added to a traffic demand matrix element that was nonzero before the noises are added. This noise matrix $\mathcal{N}_1$ was also used in [18], [19]. However, each nonzero (noise) element here in $\mathcal{N}_1$ has a larger standard deviation, which is equal to $1/5$ of the value of the demand matrix element it is to be added to, than that in [18], [19], which is equal to 0.3% of 1 (the normalized workload an input port receives during a scheduling window, *i.e.*, the sum of the corresponding row in D). We increase this additive noise here to highlight the performance robustness of our algorithm to such perturbations.

Different than in [18], [19], we also add (truncated) positive Gaussian noises $\mathcal{N}_2$ to a portion of the zero entries in the demand matrix in accordance with the following observation. Previous measurement studies have shown that "mice flows" in the demand matrix are heavy-tailed [25] in the sense the total traffic volume of these "mice flows" is not insignificant. To incorporate this heavy-tail behavior (of "mice flows") in the traffic demand matrix, we add such a positive Gaussian noise – with standard deviation equal to 0.3% of 1 – to 50% of the zero entries of the demand matrix. This way the "mice flows" collectively carry approximately 10% of the total traffic volume. To bring the normalized workload back to 1, we scale the demand matrix by 90% before adding $\mathcal{N}_2$, as shown in (1).

B. System Parameters

In this section, we introduce the system parameters (of the hybrid switch) used in our simulations.

Network size: We consider the hybrid switch with $n = 100$ input/output ports throughout this section. Other reasonably large (say ≥ 32) switch sizes produce similar results.

Circuit switch per-port rate r_c and packet switch per-port rate r_p: As far as designing hybrid switching algorithms is concerned, only their ratio r_c/r_p matters. This ratio roughly corresponds to the percentage of traffic that needs to be transmitted by the circuit switch. The higher this ratio is, the higher percentage of traffic should be transmitted by the circuit switch. This ratio varies from 8 to 40 in our simulations. As explained earlier, we normalize r_c to 1 throughout this paper.

Since both the traffic demand to each input port and the per-port rate of the circuit switch are all normalized to 1, the (idealistic) transmission time would be 1 when there was no packet switch, the scheduling was perfect (*i.e.*, no "slack" anywhere), and there was no reconfiguration penalty (*i.e.*, $\delta = 0$). Hence we should expect that all these algorithms result in transmission times larger than 1 under realistic "operating conditions" and parameter settings.

Reconfiguration delay (of the circuit switch) δ: In general, the smaller this reconfiguration delay is, the less time the circuit switch has to spend on reconfigurations. Hence, given a traffic demand matrix, the transmission time should increase as δ increases.

C. Performances under Different System Parameters

In this section, we evaluate the performances of Eclipse and BFF for different value combinations of δ and r_c/r_p under the traffic demand matrix with the default parameter settings (4 large flows and 12 small flows accounting for roughly 70% and 30% of the total traffic demand into each input port). We perform 100 simulation runs for each scenario, and report the simulation results in Figure 2.

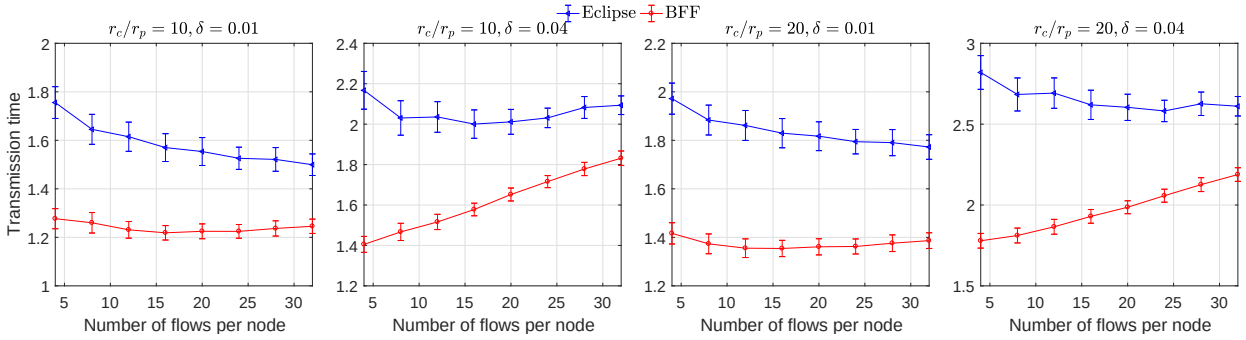

Fig. 3. Performance comparison of Eclipse and BFF while varying sparsity of demand matrix

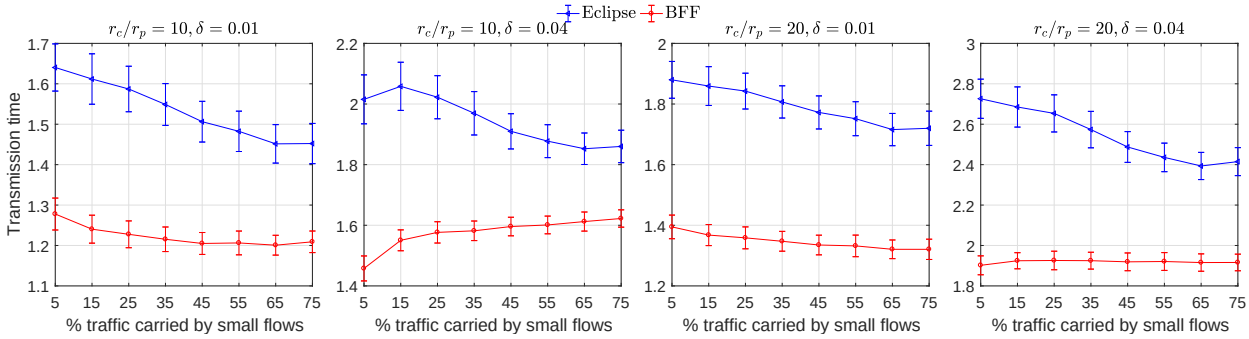

Fig. 4. Performance comparison of Eclipse and BFF while varying skewness of demand matrix

Here each point on a plot represents the average transmission time, and the corresponding error bar represents their 95% confidence interval. The results, presented in Figure 2, show that the schedules generated by BFF are consistently better, as indicated by shorter and less variable transmission times, than those generated by Eclipse, especially when reconfiguration delay δ and rate ratio r_c/r_p are large. More specifically, when $\delta = 0.01, r_c/r_p = 10$ (default setting), schedules generated by BFF result in approximately 19% shorter average transmission time than those generated by Eclipse. When $\delta = 0.04, r_c/r_p = 20$, schedules generated by BFF result in 23% shorter average transmission time than those generated by Eclipse. Meanwhile, the transmission time confidence intervals of the schedules generated by BFF in all these scenarios are about 40% shorter (i.e., less variable) than those of the schedules generated by Eclipse.

D. Performances under Different Traffic Demands

In this section, we evaluate the performance robustness of our BFF algorithm under a large set of traffic demand matrices that vary by sparsity and skewness. We control the sparsity of the traffic demand matrix D by varying the total number of flows $(n_L + n_S)$ in each row from 4 to 32, while fixing the ratio of the number of large flow to that of small flows (n_L/n_S) at $1:3$. We control the skewness of D by varying c_S, the total percentage of traffic carried by small flows, from 5% (most skewed as large flows carry the rest 95%) to 75% (least skewed). In all these evaluations, we consider four different value combinations of system parameters δ and r_c/r_p: (1) $\delta = 0.01, r_c/r_p = 10$; (2) $\delta = 0.01, r_c/r_p = 20$; (3) $\delta = 0.04, r_c/r_p = 10$; and (4) $\delta = 0.04, r_c/r_p = 20$.

Figure 3 compares the transmission time of BFF and Eclipse when the sparsity parameter $n_L + n_S$ varies from 4 to 32 and the value of the skewness parameter c_S is fixed at 0.3. Figure 4 compares the transmission time of BFF and Eclipse when the the skewness parameter c_S varies from 5% to 75% and the sparsity parameter $n_L + n_S$ is fixed at 16 $(= 4 + 12)$. In each figure, the four subfigures correspond to the four value combinations of δ and r_c/r_p above.

Both Figure 3 and Figure 4 show that schedules computed by BFF result in approximately $20\% - 30\%$ shorter average transmission times than those computed by Eclipse, under various traffic demand matrices. They also show that the transmission times of the former schedules (i.e., those generated by BFF) are less variable (as indicated by shorter 95% confidence interval bars) or more stable than those of the latter schedules,

E. Execution time comparison of Eclipse and BFF

In this section, we present the execution times of Eclipse and BFF (all implemented in C++) for different δ, under the traffic demand matrix with the default parameter settings ($n_L = 4$, $n_S = 12$, $c_L = 0.7$, $c_S = 0.3$). We set $r_c/r_p = 10$ for each scenario. These execution time measurements are performed on a Dell Precision Tower 3620 workstation equipped with an Intel Core i7-6700K CPU @4.00GHz processor and 16GB RAM, and running Windows 10 Professional. We perform 100 simulation runs for each scenario. The average execution times are shown in Table II.

TABLE II
COMPARISON OF AVERAGE EXECUTION TIME FOR ECLIPSE AND BFF

	$n = 32$			$n = 100$		
δ	0.0025	0.01	0.04	0.0025	0.01	0.04
Eclipse	$1.25s$	$0.80s$	$0.44s$	$34.6s$	$16.4s$	$6.88s$
BFF	$2.50ms$	$2.34ms$	$1.93ms$	$30.1ms$	$22.6ms$	$17.4ms$

As shown in Table II, the execution time of BFF is roughly three orders of magnitude smaller than those of Eclipse. We have also implemented Eclipse++ and measured its execution time. It is roughly three orders of magnitude higher than those of Eclipse; the same observation [26] was made by the first author of [19] (the Eclipse/Eclipse++ paper).

V. RELATED WORK

A. Hybrid Switch Scheduling Algorithms

As stated earlier, all existing hybrid switching solutions are designed based the assumption that the circuit switch is not partially reconfigurable. Liu et al. [18] first characterized the mathematical problem of the hybrid switch scheduling using direct routing only and proposed a greedy heuristic solution, called Solstice. In each iteration, Solstice effectively tries to find the Max-Min Weighted Matching (MMWM) in D, which is the full matching with the largest minimum element. The duration of this matching (configuration) is then set to this largest minimum element. Although its asymptotic computational complexity is a bit lower than Eclipse's, our experiments show that its actual execution time is similar to Eclipse's since Solstice has to compute a larger number of configurations K than Eclipse, which generally produces a tighter schedule. The Solstice [18] work mentioned the technological feasibility of partial reconfiguration, but made no attempt at exploiting this capability.

This hybrid switching problem has also been considered in two other works [10], [11]. Their problem formulations are a bit different than that in [19], [18], and so are their solution approaches. In [10], the problem of matching senders with receivers is modeled as a (distributed) stable marriage problem, in which a sender's preference score for a receiver is equal to the age of the data the former has to transmit to the latter in a scheduling epoch, and is solved using a variant of the Gale-Shapely algorithm [27]. This solution is aimed at minimizing transmission latencies while avoiding starvations, and not at maximizing network throughput, or equivalently minimizing transmission time. The innovations of [11] are mostly in the aspect of systems building and are not on matching algorithm designs.

As mentioned earlier, the state of the art solution, Eclipse [19], which performs better than Solstice [18], is also a greedy heuristic, but optimizes a very different objective function than that in Solstice. More specifically, in each iteration, Eclipse tries to extract and subtract a configuration (M, α) (using matching M for a "net" duration of α) from the demand matrix D that tries to maximize the cost-adjusted utility function $\frac{U(M,\alpha)}{\delta+\alpha}$, where $U(M, \alpha)$ is the total amount of traffic the configuration would remove from (what remains of) D and $\delta + \alpha$ is the "gross" duration (cost) of the configuration including the reconfiguration delay δ. Since this optimization problem is very computationally expensive, it was shown in [19] that a computationally efficient heuristic was proposed in [19] that empirically produces the optimal value most of time on real-world instances. This heuristic solution, invoking the scaling algorithm for computing *maximum weighted matching* (MWM) [24] $O(\log n)$ times, still requires much higher computational complexity than BFF, as shown in Table I.

Eclipse, like most other hybrid switching algorithms, considers and allows only direct routing in the following sense: All circuit-switched data packets reach their respective final destinations in one-hop (*i.e.*, enters and exits the circuit switch only once). A separate algorithm named Eclipse++ was proposed in [19] to explore indirect routing. The computational complexity of Eclipse++ is however extremely high, since Eclipse++ has to perform a large number of single-source shortest-path computations. Both us and the authors of [19] found that Eclipse++ is roughly three orders of magnitude more computationally expensive than Eclipse [26] for a data center with $n = 100$ racks.

To the best of our knowledge, Albedo [17] is the only other indirect routing solution for hybrid switching, besides Eclipse++ [19]. Albedo was proposed to solved a different type of hybrid switching problem: dealing with the fallout of inaccurate estimation of the traffic demand matrix D. It works as follows. Based on an estimation of D, Albedo first computes a direct routing schedule using Eclipse or Solstice. Then any unexpected "extra workload" resulting from the inaccurate estimation is routed indirectly. However, Albedo has a high computational complexity, since it needs to perform a larger number of single-source shortest path computations.

B. Optical Switch Scheduling Algorithms

Scheduling of circuit switch alone (*i.e.*, no packet switch), that is not partially reconfigurable, has been studied for decades. Early works often assumed the reconfiguration delay to be either zero [28], [9] or infinity [13], [29], [30]. Further studies, like DOUBLE [13], ADJUST [31] and other algorithms such as [29], [32], take the actual reconfiguration delay into consideration. Recently, a solution called Adaptive MaxWeight (AMW) [33], [34] was proposed for optical switches (with nonzero reconfiguration delays). The basic idea of AMW is that when the maximum weighted configuration (matching) has a much higher weight than the current configuration, the optical switch is reconfigured to the maximum weighted configuration; otherwise, the configuration of the optimal switch stays the same. However, this algorithm may lead to long queueing delays (for packets) since it usually reconfigures infrequently.

Towles et al. [13] first considered the scheduling of circuit switch (alone) that is partially reconfigurable and discovered that such a scheduling problem is algorithmically equivalent to open-shop scheduling (OSS) [12]. They tried to adapt List

scheduling (LIST) [20], [21], [22], the well-known family of polynomial-time heuristic algorithms, to tackle this problem. However, no algorithm in the LIST family benefits much from the partial reconfiguration capability, as explained earlier. Recently, Van et al. [14] proposed a solution called adaptive open-shop algorithm (AOS), for scheduling partially reconfigurable optical switches. It essentially runs an optimal preemptive strategy [35] and a non-preemptive LIST strategy [12], [36] in a dynamic and flexible fashion to find a good schedule. However, the computational complexity of the optimal preemptive strategy [35] alone is $O(n^4)$, which is much higher than that of BFF.

VI. Conclusion

Although considerable research effort has been made on hybrid circuit and packet switching, no solution has been proposed to exploit the partial reconfigurability of circuit switches, which has become increasingly widely available. In this work, we propose BFF (best first fit), the first such solution. BFF not only significantly outperforms Eclipse, given the same workloads (the traffic demand matrices) used in [19], but also has much lower computational complexity than Eclipse.

Acknowledge

This work is supported in part by US NSF through award CNS 1423182.

References

[1] C. DeCusatis, "Optical interconnect networks for data communications," *J. Lightw. Technol.*, vol. 32, no. 4, pp. 544–552, 2014.

[2] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: cloud scale load balancing," in *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4. ACM, 2013, pp. 207–218.

[3] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 339–350, 2010.

[4] H. Wang, A. S. Garg, K. Bergman, and M. Glick, "Design and demonstration of an all-optical hybrid packet and circuit switched network platform for next generation data centers," in *OFC*. Optical Society of America, 2010, p. OTuP3.

[5] N. Farrington, A. Forencich, P.-C. Sun, S. Fainman, J. Ford, A. Vahdat, G. Porter, and G. C. Papen, "A 10 us hybrid optical-circuit/electrical-packet network for datacenters," in *OFC*. Optical Society of America, 2013, pp. OW3H–3.

[6] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen, "OSA: An optical switching architecture for data center networks with unprecedented flexibility," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 498–511, 2014.

[7] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan, "c-through: Part-time optics in data centers," in *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4. ACM, 2010, pp. 327–338.

[8] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter, "Circuit switching under the radar with reactor." in *NSDI*, vol. 14, 2014, pp. 1–15.

[9] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat, "Integrating microsecond circuit switching into the data center," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 447–458, Aug. 2013.

[10] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper, "Projector: Agile reconfigurable data center interconnect," in *SIGCOMM*, 2016, pp. 216–229.

[11] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer, "Firefly: A reconfigurable wireless data center fabric using free-space optics," in *Proceedings of the ACM SIGCOMM*, 2014, pp. 319–330.

[12] M. L. Pinedo, *Scheduling: theory, algorithms, and systems*. Springer, 2016.

[13] B. Towles and W. J. Dally, "Guaranteed scheduling for switches with configuration overhead," *IEEE/ACM Trans. Netw.*, vol. 11, no. 5, pp. 835–847, 2003.

[14] D. P. Van, M. Fiorani, L. Wosinska, and J. Chen, "Adaptive open-shop scheduling for optical interconnection networks," *J. Lightw. Technol.*, 2017.

[15] V. J. Rayward-Smith and D. Rebaine, "Open shop scheduling with delays," *RAIRO-Theoretical Informatics and Applications*, vol. 26, no. 5, pp. 439–447, 1992.

[16] A. Mekkittikul and N. McKeown, "A starvation-free algorithm for achieving 100% throughput in an input-queued switch," in *ICCCN*, vol. 96. Citeseer, 1996, pp. 226–231.

[17] C. Li, M. K. Mukerjee, D. G. Andersen, S. Seshan, M. Kaminsky, G. Porter, and A. C. Snoeren, "Using indirect routing to recover from network traffic scheduling estimation error," in *ANCS*. IEEE Press, 2017, pp. 13–24.

[18] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky *et al.*, "Scheduling techniques for hybrid circuit/packet networks," *CoNEXT*, 2015.

[19] S. Bojja Venkatakrishnan, M. Alizadeh, and P. Viswanath, "Costly circuits, submodular schedules and approximate carathéodory theorems," in *SIGMETRICS*. ACM, 2016, pp. 75–88.

[20] D. B. Shmoys, C. Stein, and J. Wein, "Improved approximation algorithms for shop scheduling problems," *SIAM J. Comput*, vol. 23, no. 3, pp. 617–632, 1994.

[21] D. S. Hochba, "Approximation algorithms for np-hard problems," *ACM SIGACT News*, vol. 28, no. 2, pp. 40–52, 1997.

[22] H. Bräsel, A. Herms, M. Mörig, T. Tautenhahn, J. Tusch, and F. Werner, "Heuristic algorithms for open shop scheduling to minimize mean flow time, part i: Constructive algorithms," 2008.

[23] M. Mastrolilli, M. Queyranne, A. S. Schulz, O. Svensson, and N. A. Uhan, "Minimizing the sum of weighted completion times in a concurrent open shop," *Operations Research Letters*, vol. 38, no. 5, pp. 390–395, 2010.

[24] R. Duan and H.-H. Su, "A scaling algorithm for maximum weight matching in bipartite graphs," in *SODA*. SIAM, 2012, pp. 1413–1424.

[25] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*. ACM, 2010, pp. 267–280.

[26] S. Bojja Venkatakrishnan, In-Person Discussions at Sigmetrics Conference, June. 2017.

[27] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, 1962.

[28] T. Inukai, "An efficient SS/TDMA time slot assignment algorithm," *IEEE Trans. Commun.*, vol. 27, no. 10, pp. 1449–1455, 1979.

[29] B. Wu and K. L. Yeung, "Nxg05-6: Minimum delay scheduling in scalable hybrid electronic/optical packet switches," in *GLOBECOM*. IEEE, 2006, pp. 1–5.

[30] I. Gopal and C. Wong, "Minimizing the number of switchings in an ss/tdma system," *IEEE Trans. Commun.*, vol. 33, no. 6, pp. 497–501, 1985.

[31] X. Li and M. Hamdi, "On scheduling optical packet switches with reconfiguration delay," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 7, pp. 1156–1164, 2003.

[32] S. Fu, B. Wu, X. Jiang, A. Pattavina, L. Zhang, and S. Xu, "Cost and delay tradeoff in three-stage switch architecture for data center networks," in *HPSR*. IEEE, 2013, pp. 56–61.

[33] C.-H. Wang, T. Javidi, and G. Porter, "End-to-end scheduling for all-optical data centers," in *INFOCOM*. IEEE, 2015, pp. 406–414.

[34] C.-H. Wang, S. T. Maguluri, and T. Javidi, "Heavy traffic queue length behavior in switches with reconfiguration delay," *arXiv preprint arXiv:1701.05598*, 2017.

[35] T. Gonzalez and S. Sahni, "Open shop scheduling to minimize finish time," *J. ACM*, vol. 23, no. 4, pp. 665–679, 1976.

[36] T. Gonzalez, O. H. Ibarra, and S. Sahni, "Bounds for lpt schedules on uniform processors," *SIAM J. Comput*, vol. 6, no. 1, pp. 155–166, 1977.