HYBRID AND OPTICAL SWITCHING SCHEDULING ALGORITHMS IN DATA CENTER NETWORKS

A Dissertation
Presented to
The Academic Faculty

By

Liang Liu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2020

HYBRID AND OPTICAL SWITCHING SCHEDULING ALGORITHMS IN DATA CENTER NETWORKS

Approved by:

Dr. Jun (Jim) Xu, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Mostafa Ammar
School of Computer Science
Georgia Institute of Technology

Dr. Ellen W. Zegura
School of Computer Science
Georgia Institute of Technology

Dr. Lance Fortnow
School of Computer Science
Illinois Institute of Technology

Dr. Mohit Singh
School of Industrial and Systems Engineering
Georgia Institute of Technology

Date Approved: March 27, 2020

The best preparation for tomorrow is doing your best today

H. Jackson Brown, Jr.

ACKNOWLEDGEMENTS

I would like to convey my deepest gratitude to my advisor Jun (Jim) Xu, for giving me the opportunity to be a member in his group and continuous supports on my research. His strong passion on research and broad and deep knowledge inspired, guided, and helped me overcome many difficulties. He is also very considerate and helped me get through many tough times.

I would like to express my sincere gratitude to my other thesis committee members, Dr. Mostafa Ammar, Dr. Ellen W. Zegura, Dr. Lance Fortnow, and Dr. Mohit Singh for being great teachers and mentors, and for their interests in my graduate work and their valuable and insightful comments.

My thanks also goes to the lab members, Sen Yang, Long Gong, Yimeng Zhao, Tarun Mangla, Jingfan Meng, Huayi Wang, and others. Thank you for the help, insights, and discussions. I am also indebted to my friends, Qiang Hu, Shan Chen, Bobin Deng, Tongzhou Sun, Zidong Zhou, and others. Thank you for creating an unforgettable memory for me in the past few years.

I owe my utmost gratitude to my family, especially my parents, Guang Liu and Lihua Zhang. Their endless love, support, and encourage are always the source of my strength and motivation. This dissertation is dedicated to them.

This work is supported in part by US NSF through award CNS-1909048.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Hybrid-switched data center networks have received considerable research attention recently. A hybrid-switched data center network employs a much faster optical switch that is reconfigurable with a nontrivial cost, and a much slower packet switch, to interconnect its racks of servers. A fundamental research (optimization) problem in such a hybrid-switched data center network is, how to properly schedule the optical switch to both fully utilize its high bandwidth and minimize its reconfiguration cost. However, this optimization problem, in various forms, has been proved to be NP-hard. Almost all existing solutions are mostly greedy heuristics that either achieve poor throughput performance, or require an extremely high computational complexity.

The objective of this dissertation research is to design high-performance low-complexity scheduling algorithms for optical switches in hybrid-switched data center networks to boost the throughput performance.

We proposed three different scheduling algorithms that exploit different methodologies towards our objective. The first work, 2-hop Eclipse [1], extends the state-of-the-art solution Eclipse [2] from a direct routing scheduling algorithm to an algorithm that supports both direct routing and 2-hop indirect routing. The second work, Quantized Birkhoff-von Neummann Decomposition (QBvND) [3], builds upon a classic framework for packet switch scheduling. Although adapting this framework to optical switch scheduling straightforwardly incurs high reconfiguration cost, we manage to overcome this problem by combining the framework with the quantization technique. The third work, Best First Fit (BFF), is the first hybrid switching solution that exploits *partial reconfigurability* of optical switches, which leads to a new form of the optimization problem. This new optimization problem can be viewed as a special case of the well-known Open Shop Scheduling (OSS) problem. BFF, adapted from an existing OSS solution, not only significantly outperforms, but also has much lower computational complexity than the state-of-the-art solutions.

Another closely related research problem about the optical and hybrid switching scheduling that we investigated is, when the racks of servers are connected by multiple *independent* (i.e., parallel) optical switches, how to split the traffic demand matrix into sub-workload matrices and give them to the parallel optical switches as their respective workloads. This problem induces a general mathematical problem: How to split a nonnegative matrix D into s nonnegative matrices $D_1, D_2, ..., D_s$ such that (1) each of them accounts for $1/s$ the row sum and the column sum of D, and (2) the total number of nonzero entries in these s matrices is minimized. Here the first condition aims for workload load-balancing, and the second condition aims for minimizing the overall reconfiguration cost. We develop a combinatorial solution for this matrix split problem that achieves both perfect load-balancing (i.e., satisfies the first condition) and near-optimal (minimum) reconfiguration costs. Our evaluation results show that, using this matrix split algorithm, the parallel optical switches deliver balanced and ideal throughput performance under various system parameter settings and various traffic demands.

CHAPTER 1
INTRODUCTION

1.1 Motivation and Background

Fueled by the phenomenal growth of cloud computing, data center networks (DCN) continue to grow relentlessly both in size, as measured by the number of racks of servers a DCN has to interconnect, and in speed, as measured by the amount of traffic a DCN has to transport per unit of time to/from each rack [4]. A traditional data center network (DCN) 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 [5]. 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 cost-effective approach to this scalability problem, called hybrid DCN architecture, has received considerable research attention in recent years [6, 7, 8, 9, 2].

In a hybrid data center, shown in Figure 1.1, n racks of computers on the left hand side (LHS) are connected by both an optical (circuit) switch and a packet switch to n racks on the right hand side (RHS). Each switch transmits data from input ports (racks on the LHS) to output ports (racks on the RHS) according to the configuration of the switch (a bipartite matching) at the moment. The optical switch has higher bandwidth than the packet switch, typically by an order of magnitude, but 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 [6, 10, 11, 12, 13]. Note that racks on the LHS is an identical copy of those on the RHS; however

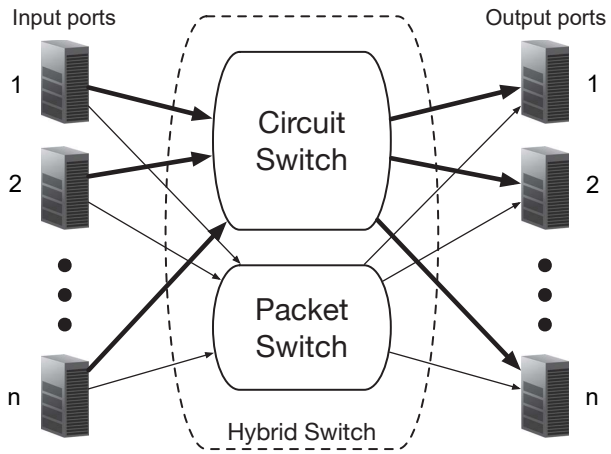

Figure 1.1: Hybrid Optical and Packet Switch

we restrict the role of the former to only transmitting data and restrict the role of the latter to only receiving data. This purpose of this duplication (of racks) and role restrictions is that the resulting hybrid data center topology can be modeled as a bipartite graph.

The following optimization problem is the focus of most of the research works, including ours, on hybrid switching: Given a traffic demand matrix D from the input ports to the output ports, how should we schedule the optical switch to best (*i.e.*, in the shortest makespan or equivalently with the highest throughput) meet the demand? A typical schedule for the optical switch consists of a sequence of configurations (matchings) and their time durations (M_1, α_1), (M_2, α_2), $\cdots$, (M_K, α_K). An ideal schedule should let the optical switch remove (*i.e.*, transmit) the vast majority of the traffic demand from D by a small number of configurations (i.e., small reconfiguration cost), so that every row or column sum of the remaining demand matrix is small enough for the packet switch to handle. Since the problem of computing the optimal schedule (i.e., in the shortest makespan) for hybrid switching, in various forms, is NP-hard [14], almost all existing solutions are greedy heuristics.

1.2 Research Objective and Main Contributions

The objective of this dissertation research is to design high-performance low-complexity scheduling algorithms for hybrid-switched data center networks. Such algorithms should achieve good throughput/makespan performance and be scalable (i.e., low computational complexity) when the switch size (i.e., number of input/output ports) increases. Existing hybrid switching algorithms either provide poor makespan performance (e.g., Solstice [9]), or require an extremely high computational complexity (e.g., Eclipse [2], Albedo [15]).

The main contribution of this dissertation research is, we proposed three different scheduling algorithms, 2-hop Eclipse [1], Quantized Birkhoff-von Neumann Decomposition (QBvND) [3], and Best First Fit (BFF) [16], for optical switch scheduling that investigated different methodologies towards the objective. Another contribution is, we investigated a closely related research problem about the optical and hybrid switching scheduling, that is, when the racks of servers are connected by multiple *independent* (i.e., parallel) optical switches, how to properly split the traffic demand into sub-workloads and feed them to the parallel optical switches as their respective workloads, so that each of the parallel optical switches can achieve high throughput (or short makespan) performance when independently scheduling their respective workloads by our proposed scheduling algorithms.

1.2.1 Scheduling algorithms for single optical switch

In this section, we briefly introduce the three scheduling algorithms. The first work, 2-hop Eclipse [1], extends the state-of-the-art solution Eclipse [2] from a direct routing scheduling algorithm to an algorithm that supports both direct routing and 2-hop indirect routing. More specifically, in each iteration, Eclipse tries to extract and subtract a matching (with its duration) from the traffic demand matrix that has the largest *cost-adjusted utility*, which is defined as the quotient of the amount of the traffic demand the matching can serve and the duration of the matching. 2-hop Eclipse, on the other hand, also tries to extract and

subtract a matching from the traffic demand matrix that has the largest *cost-adjusted utility* in each iteration. The difference is, the cost-adjust utility here accounts for not only the traffic that can be transmitted under direct routing, but also that can be indirectly routed over all possible 2-hop paths. 2-hop Eclipse harvests a considerate throughput gain from 2-hop indirect routing while its asymptotical computational complexity is the same as that of Eclipse.

The second work, QBvND [3] builds on a framework in packet switching scheduling [17], which is to stuff the traffic matrix into a scaled doubly stochastic matrix first and then decompose the stuffed matrix into a sequence of configurations (permutation matrices) with durations (weights of the matrices) as the schedule using Birkhoff-von Neumann decomposition (BvND) [18]. Note that this packet switching method itself however performs poorly in optical switches, since it would result in a quadratic number of configurations that leads to a huge reconfiguration and computation cost. We overcome this problem by quantizing the demand matrix and substitute the standard BvND algorithm with the max-min BvND algorithm [19], which significantly reduces the total number of configurations both theoretically and empirically.

The third work, Best First Fit (BFF), is the first hybrid switching algorithm that exploits *partial reconfigurability* of optical switches for hybrid data center networks [16]. 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 [20, 21]), 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. By exploiting this partial reconfigurability, BFF not only significantly outperforms but also has much lower computational complexity than the state of the art solutions.

1.2.2 Scheduling problem in parallel-optical-switched networks

Another closely related research problem about the optical and hybrid switching scheduling problem is, when the racks of servers are connected by multiple *independent* (i.e., parallel) optical switches (say s optical switches), how to split the traffic demand D into sub-workload matrices $D_1, D_2, ..., D_s$ and give them to the parallel optical switches as their respective workloads. This approach makes perfect systems sense, as the scheduling of each sub-workload matrix can be computed independently of each other (using a different processor).

We focus on a Matrix Split and Balance (MSB) problem that lies at the heart of this divide-and-conquer approach: How to split the traffic demand D so that the resulting sub-workload matrices $D_1, D_2, ..., D_s$ lead to near-optimal switching performance yet the computation of this split is not NP-hard or otherwise extremely expensive? One intuitive notion of optimality is to minimize the worst-case (i.e., longest) makespan of the s schedules resulting from $D_1, D_2, ..., D_s$ respectively. This optimality notion is however computationally infeasible since even to minimize the makespan of a single schedule is usually NP-hard [14]. Hence, we instead impose the following two milder conditions on this split that can work *toward this optimality*.

The *first condition* is that the total traffic demand in every row of D, which corresponds to that originating at an input port, should be evenly split among $D_1, D_2, ..., D_s$, and so should every column of D, which corresponds to that destined for an output port; we call this condition *line-even* as each row or column is a straight line through the matrix. The ra-

tionale for such a split is that, since every optical switch receives roughly the same amount of workload to its $2n$ input and output ports, these s switches hopefully can finish their respective workloads in similar amounts of time (i.e., similar makespans), leading to a short *overall makespan* (the maximum among the s makespans).

Although letting $D_1 = D_2 = \cdots = D_s = D/s$ trivially satisfies this condition, this naive split is far from ideal because every nonzero entry in D is cut into s identical pieces, each of which incurs a nontrivial reconfiguration delay δ. Hence we impose the *second condition* that $\sum_{k=1}^{s} \|D_k\|_0$, the total number of nonzero entries in these s matrices, is to be minimized, which we call the *sparsity* condition. Here $\|M\|_0$ denotes the number of nonzero entries in a matrix M.

To summarize, our MSB problem is to split D into $D_1, D_2, ..., D_s$ under the *constraint* that every row or column sum of D is evenly split and with the objective of *minimizing* the total number of nonzero entries in $D_1, D_2, ..., D_s$. Unfortunately, this relatively easier constrained minimization problem is still NP-hard [22]. We propose LESS (Line-Even Sparse Split) [23], an approximation algorithm that provides the strong theoretical guarantee (a bounded gap between the total number of nonzero entries split by LESS and the theoretical optimal) on this constrained minimization problem. The LESS algorithm, based on linear programming (LP), is conceptually straightforward. Its execution time is however a bit too long. We reduce this execution time by converting the LP computation problem to a graph computation problem that is much less expensive.

CHAPTER 2
LITERATURE SURVEY

In this chapter, we summarize related works on hybrid switching and standalone optical switching algorithms respectively in section 2.1 and section 2.2. Then we introduce some other works that proposed different datacenter network architectures using optical switches. We defer to describe Eclipse [2], the state of the art hybrid switching algorithm, in section 3.2, since 2-hop Eclipse, the hybrid switching algorithm that we will propose in chapter 3, builds upon Eclipse.

2.1 Hybrid Switching Algorithms

Liu et al. [9] 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. The Solstice [9] 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 [20, 21]. Their problem formulations are a bit different than that in [2, 9], and so are their solution approaches. In [20], 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 [24]. 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 [21]

are mostly in the aspect of systems building and are not on matching algorithm designs.

To the best of our knowledge, Albedo [15] is the only other indirect routing solution for hybrid switching, besides Eclipse++ [2]. 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.

2.2 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 [25, 13] or infinity [26, 27, 28]. Further studies, like DOUBLE [26], ADJUST [14] and other algorithms such as [27, 29], take the actual reconfiguration delay into consideration.

Towles et al. [26] 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) [30]. They tried to adapt List scheduling (LIST) [31, 32, 33], 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. LIST gains no advantages from partial reconfiguration, because in this circumstance (*i.e.*, no packet switch), all traffic including the demand from large number of short VOQs has to be transmitted by the optical switch, and that incurs a massive cost for reconfiguration inevitably.

Recently, a solution called Adaptive MaxWeight (AMW) [34, 35] 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. [26] 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) [30]. They tried to adapt List scheduling (LIST) [31, 32, 33], 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. [36] proposed a solution called adaptive open-shop algorithm (AOS), for scheduling partially reconfigurable optical switches. It essentially runs an optimal preemptive strategy [37] and a non-preemptive LIST strategy [30, 38] in a dynamic and flexible fashion to find a good schedule. However, the computational complexity of the optimal preemptive strategy [37] alone is $O(n^4)$.

Bianco et al. [39] considered a different aspect of the lifetime of optical switch and proposes the fatigue-aware scheduling approach that takes both throughput and fatigue costs into consideration. The main idea of reducing the fatigue cost in [39] is sorting configurations properly, so that the number of variations (*i.e.*, connection setup and breakdown) occurring in the sequence of matchings is minimized.

2.3 Other optical datacenter networks

There are also a class of optical switch scheduling algorithms for specific optical datacenter networks. RotorNet [40] proposed an optical datacenter network design that uses rotor switches rotating through a set of pre-defined matchings. This (traffic) demand-unaware design makes the computational complexity of the scheduling operation extremely low ($O(1)$ per frame). However, it gives up on the opportunity to boost network (throughput) performance using workload information as were done in recent work [9, 2, 1, 16, 3].

For instance, while demand-aware networks can achieve an outstanding throughput performance (e.g., $> 80\%$) given sparse and skewed traffic demand matrices [9, 2, 1, 16, 3], RotorNet can only achieve $< 50\%$ throughput under such traffic demands.

The Petabit switch fabric [41] adopts a three-stage Clos network to build a large-scale optical switch fabric that includes interconnected Arrayed Waveguide Grating Router (AWGRs) and tunable wavelength converters (TWCs). It does not use any buffers inside the switch fabric and thus avoids the power hungry E/O and O/E conversion. Instead, the congestion management is performed using electronic buffers in the Line cards that are connected to the input ports. Although the optical fabric performs highspeed switching with nanosecond-level reconfiguration overhead, the reconfiguration time is not negligible compared to the packet length. To reduce the impact of switching (i.e., reconfiguration) time, the Petabit switch adopt frame-based switching, which assembles packets into fixed size frames in the ingress of the line cards and disassembled at the egress of the line cards. However, besides the reconfiguration time, a guard time needs to be inserted between consecutive frames to compensate the propagation delay difference, since the optical switch fabric is bufferless and switch modules are reconfigured synchronously. The reconfiguration time and guard time could significantly decrease the throughput of the Petabit switch.

In [42], Shay et al. extended the classic hybrid switching architecture by introducing one-to-many and many-to-one composite paths between the circuit switch and the packet switch to improve the performance of hybrid switch under one-to-many and/or many-to-one traffic workloads. They proposed a simple technique to transform the problem of scheduling such a special hybrid switch into that of scheduling a standard hybrid switch. The transformed problem is then solved using existing solutions such as Solstice [9] and Eclipse [2]. However, no new solutions to the standard hybrid switching problem were proposed in [42].

Mellette et al. [43] proposed a new beam-steering optical switch, "partially configurable" optical switch, for data centers with the switching time in the order of less than

$150\,\mu s$. Here "partial configurability" means only a subset of output ports are directly connected with each input port, which is very different from the "partial reconfigurability" in the thesis. In an $n \times n$ "partially configurable" optical switch, the connectivity of each input port is limited by $k << n$. This reduction in the number of optical states significantly reduces the aperture and tilt requirements of the micromirror, allowing it to be redesigned for higher speed operation. Here "partial configurable" means only a subset of port-mappings (matchings) are achievable. Although this phrase is quite similar with our "partial reconfigurable" optical switch, they are totally different concepts.

2.4 Flow scheduling algorithms in DCN

There is another group of DCN load-balancing algorithms on flow routing scheduling. The seminal work [44, 45] on flow routing uses Equal Cost MultiPath (ECMP), which equally splits the traffic among available shortest paths by hashing, to do load-balancing. However, ECMP can perform poorly since it may hash large flows to the same path and cause congestion. And ECMP also perform poorly in asymmetry data center architectures (such as JellyFish [46]). There has been much research on flow scheduling algorithms [47, 48, 49, 50, 51, 52, 53, 54] that addresses the shortcoming of ECMP and improves its load-balancing performance. Our problem scope is different from that of these prior works, which are focused on load-balancing among different switches over the whole DCN topology (*e.g.,* FatTree [48], JellyFish [46], *etc*). Hybrid switching on the other hand is designed to do load-balancing within a switch for local traffic.

CHAPTER 3

2-HOP ECLIPSE: ADAPT INDIRECT ROUTING IN HYBRID SWITCH SCHEDULING

3.1 System Model and Problem Statement

In this section, we formulate the problem of hybrid circuit and packet switching precisely. We first specify the aforementioned traffic demand traffic D precisely. Borrowing the term virtual output queue (VOQ) from the crossbar switching literature [55], we refer to, the set of packets that arrive at input port i and are destined for output j, as VOQ(i, j). In some places, we refer to a VOQ also as an input-output flow. The demand matrix entry $D(i, j)$ 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 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 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 2-hop Eclipse 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), 2-hop Eclipse 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 [9]: 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 thesis. An alter-

native formulation, used in [2], 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.

A schedule of the circuit switch consists of a sequence of circuit switch configurations and their durations: (M_1, α_1), (M_2, α_2), $\cdots$, (M_K, α_K). Each M_k is an $n \times n$ permutation (matching) matrix that denotes the k^{th} switch configuration. $M_k(i,j) = 1$ if input i is connected to output j and $M_k(i,j) = 0$ otherwise. α_k denotes its duration. Note that the circuit switch takes a reconfiguration delay δ between any two sequential switch configurations, and it transmits no traffic during this period. The total transmission time of the above schedule is $K\delta + \sum_{k=1}^{K} \alpha_k$, where δ is the reconfiguration delay, K is the total number of configurations in the schedule.

Since computing the optimal circuit switch schedule *alone* (*i.e.*, when there is no packet switch), in its full generality, is NP-hard [14], almost all existing solutions are greedy heuristics. Indeed, the typical workloads we see in data centers exhibit two characteristics that are favorable to such greedy heuristics: sparsity (the vast majority of the demand matrix elements have value 0 or close to 0) and skewness (few large elements in a row or column account for the majority of the row or column sum) [9].

3.2 Background on Eclipse and Eclipse++

Since our 2-hop Eclipse algorithm builds upon Eclipse [2], we provide here a more detailed description of Eclipse. Eclipse iteratively chooses a sequence of configurations, one per iteration, according to the following greedy criteria: In each iteration, Eclipse tries to extract and subtract a matching from the demand matrix D that has the largest *cost-adjusted utility*, defined as follows. For a configuration (M, α) (using a permutation matrix M for a duration of α), its utility $U(M, \alpha)$, before adjusting for cost, is $U(M, \alpha) \triangleq \| \min(\alpha M, D_{\text{rem}}) \|_1$, where D_{rem} denotes what remains of the traffic demand (matrix) D after we subtract from it the amounts of traffic to be served by the circuit switch according to the previous match-

ings, *i.e.*, those computed in the the previous iterations. The $\|\cdot\|_1$ denotes the entrywise L_1 matrix norm, which is the summation of the absolute values of all matrix elements. Note that $U(M, \alpha)$ is precisely the total amount of traffic the configuration (M, α) would remove from D. The cost of the configuration (M, α) is modeled as $\delta + \alpha$, which accounts for the reconfiguration delay δ. The cost-adjusted utility is simply their quotient $\frac{U(M,\alpha)}{\delta+\alpha}$.

Algorithm 1: Eclipse

Input: Traffic demand D;
Output: Sequence of schedules $(M_k, \alpha_k)_{k=1,\ldots,K}$;

```
sch ← {};                                              ▷ schedule
D_rem ← D;                                             ▷ remaining demand
t_c ← 0;                                               ▷ transmission time of circuit switch
while ∃ any row or column sum of D_rem > r_p t_c do
    (M, α) ← argmax_{M∈𝓜, α∈ℝ+} ‖min(αM, D_rem)‖_1 / (δ+α);
    sch ← sch ∪ {(M, α)};
    t_c ← t_c + δ + α;
    D_rem ← [D_rem − α · M]^+;
end
```

Although the problem of maximizing this cost-adjusted utility is very computationally expensive, it was shown in [2] that an computationally efficient heuristic algorithm solution exists that empirically produces the optimal value most of time. This solution, invoking the scaling algorithm for computing *maximum weighted matching* (MWM) [56] $O(\log n)$ times, has a relatively low computational complexity of $O(n^{5/2} \log n \log B)$, where B is the value of the largest element in D. Hence the computational complexity of Eclipse is $O(Kn^{5/2} \log n \log B)$, shown in Table 3.1, where K is the total number of matchings (iterations) used.

However, restricting the solution strategy space to only direct routing algorithms may leave the circuit switch underutilized. For example, a connection (edge) from input i_0 to output j_0 belongs to a matching M that lasts 50 microseconds, but at the start time of this matching, there is only 40 microseconds worth of traffic left for transmission from i_0 to j_0, leaving 10 microseconds of "slack" (*i.e.*, *residue capacity*) along this connection. The

existence of connections (edges) with such "slacks" makes it possible to perform indirect (*i.e.,* multi-hop) routing of remaining traffic via one or more relay nodes through a path consisting of such edges.

Eclipse++ [2] explored indirect routing using these "slacks". It was shown in [2] that optimal indirect routing using such "slacks", left over by a direct routing solution such as Eclipse, can be formulated as the maximum multi-commodity flow over a "slack graph", which is NP-complete [57, 58, 59, 60]. Eclipse++ is a greedy heuristic that converts, with "precision loss" (otherwise P = NP), this multi-commodity flow computation to a large set of shortest-path computations. The computational complexity of Eclipse++ is still extremely high, since it involves a large number of shortest-path computations. Both us and the authors of [2] found that Eclipse++ is roughly three orders of magnitude more computationally expensive than Eclipse [61] for a data center with $n = 100$ racks.

Table 3.1: Comparison of time complexities

Algorithm	**Time Complexity**
Eclipse	$O(Kn^{5/2} \log n \log B)$
2-hop Eclipse	$O(Kn^{5/2} \log n \log B + \min(K, n)Kn^2)$
Eclipse++	$O(WKn^3(\log K + \log n)^2)$

3.3 Overview

Our first algorithm, called 2-hop Eclipse [1], builds upon and significantly improves the throughput of the state of the art solution Eclipse/Eclipse++ [2]. 2-hop Eclipse is a slight, but very subtle, modification of Eclipse. On one hand, its asymptotical computational complexity is the same as, and its execution time only slightly higher than, that of Eclipse. On the other hand, given the same workloads (the traffic demand matrices) used in [2], 2-hop Eclipse harvests, from indirect routing, much more performance gain (over Eclipse) than Eclipse++, which is three orders of magnitude more computationally expensive.

3.4 2-hop Eclipse

Unlike Eclipse, which considers only direct routing, 2-hop Eclipse considers both direct routing and 2-hop indirect routing in its optimization. More specifically, 2-hop Eclipse iteratively chooses a sequence of configurations that maximizes the cost-adjusted utility, just like Eclipse, but the cost-unadjusted utility $U(M, \alpha)$ here accounts for not only the traffic that can be transmitted under direct routing, but also that can be indirectly routed over all possible 2-hop paths.

We make a qualified analogy between this scheduling of the circuit switch and the scheduling of "flights". We view the connections (between the input ports and the output ports) in a matching M_k as "disjoint flights" (those that share neither a source nor a destination "airport") and the residue capacity on such a connection as "available seats". We view Eclipse, Eclipse++, and 2-hop Eclipse as different "flight booking"algorithms. Eclipse books "passengers" (traffic in the demand matrix) for "non-stop flights" only. Then Eclipse++ books the "remaining passengers" for "flights with stops" using only the "available seats" left over after Eclipse does its "bookings". Different than Eclipse++, 2-hop Eclipse "books passengers" for both "non-stop" and "one-stop flights" early on, although it does try to put "passengers" on "non-stop flights" as much as possible, since each "passenger" on a "one-stop flight" costs twice as many "total seats" as that on a "non-stop flight".

It will become clear shortly that the sole purpose of this qualified analogy is for us to distinguish two types of "passengers" in presenting the 2-hop Eclipse algorithm: those "looking for a non-stop flight" whose counts are encoded as the remaining demand matrix D_{rem}, and those "looking for a connection flight" to complete their potential "one-stop itineraries", whose counts are encoded as a new $n \times n$ matrix I_{rem} that we will describe shortly. We emphasize that this analogy shall not be stretched any further, since it would otherwise lead to absurd inferences, such as that such a set of disjoint "flights" must span

the same time duration and have the same number of "seats" on them.

3.4.1 The Pseudocode

The pseudocode of 2-hop Eclipse is shown in Algorithm 2. It is almost identical to that of Eclipse [2]. The only major difference is that in each iteration (of the "while" loop), 2-hop Eclipse searches for a matching (M, α) that maximizes $\frac{\|\min(\alpha M, D_{\text{rem}}+I_{\text{rem}})\|_1}{\delta+\alpha}$, whereas Eclipse searches for one that maximizes $\frac{\|\min(\alpha M, D_{\text{rem}})\|_1}{\delta+\alpha}$. In other words, in each iteration, 2-hop Eclipse first performs some preprocessing to obtain I_{rem}, which denotes the "possible 2-hop indirect routing traffic demand" using the previous slacks. Then substitute the parameter D_{rem} by $D_{\text{rem}} + I_{\text{rem}}$ in making the "argmax" call (Line 7) to jointly optimize the cost-adjusted utility on both direct routing and 2-hop indirect routing. The "while" loop of Algorithm 2 terminates when every row or column sum of D_{rem} is no more than $r_p t_c$, where r_p denotes the (per-port) transmission rate of the packet switch and t_c denotes the total transmission time used so far by the circuit switch, since the remaining traffic demand can be transmitted by the packet switch (in t_c time). Note there is no occurrence of r_c, the (per-port) transmission rate of the circuit switch, in Algorithm 2, because we normalize r_c to 1.

3.4.2 The Matrix I_{rem}

Just like D_{rem}, the value of I_{rem} changes after each iteration. We now explain the value of I_{rem}, at the beginning of the k^{th} iteration ($k > 1$). To do so, we need to first introduce another matrix R. As explained earlier, among the edges that belong to the matchings (M_1, α_1), (M_2, α_2), $\cdots$, (M_{k-1}, α_{k-1}) computed in the previous $k-1$ iterations, some may have residue capacities. These residue capacities are captured in an $n \times n$ matrix R as follows: $R(l, i)$ is the total residue capacity of all edges from input l to output i that belong to one of these (previous) $k-1$ matchings. Under the qualified analogy above, $R(l, i)$ is the total number of "available seats on all previous flights from airport l to airport i". We

Algorithm 2: 2-hop Eclipse

Input: Traffic demand D;
Output: Sequence of schedules $(M_k, \alpha_k)_{k=1,\ldots,K}$;

sch $\leftarrow \{\}$; ▷ schedule
$D_{\text{rem}} \leftarrow D$; ▷ remaining demand
$R \leftarrow \mathbf{0}$; ▷ residue capacity
$t_c \leftarrow 0$; ▷ transmission time of circuit switch
while $\exists$ *any row or column sum of* $D_{rem} > r_p t_c$ **do**
  Construct I_{rem} from (D_{rem}, R); ▷ 2-hop demand matrix
  $(M, \alpha) \leftarrow \underset{M \in \mathcal{M}, \alpha \in \mathbb{R}_+}{\arg\max} \frac{\|\min(\alpha M, D_{\text{rem}} + I_{\text{rem}})\|_1}{\delta + \alpha}$;
  sch $\leftarrow$ sch $\cup \{(M, \alpha)\}$;
  $t_c \leftarrow t_c + \delta + \alpha$;
  Update D_{rem};
  Update R;
end

refer to R as the *(cumulative) residue capacity matrix* in the sequel.

Now we are ready to define I_{rem}. Consider that, at the beginning of the k^{th} iteration, $D_{\text{rem}}(l, j)$ "local passengers" (*i.e.*, those who are originated at l) who need to fly to j remain to have their "flights" booked. Under Eclipse, they have to be booked on either a "non-stop flight" or a "bus" (*i.e.*, through the packet switch) to j. Under 2-hop Eclipse, however, there is a third option: a "one-stop flight" through an intermediate "airport". 2-hop Eclipse explores this option as follows. For each possible intermediate "airport" i such that $R(l, i) > 0$ (*i.e.*, there are "available seats" on one or more earlier "flights" from l to i), $I^{(l)}_{\text{rem}}(i, j)$ "passengers" will be on the "speculative standby list" at "airport" i, where

$$I^{(l)}_{\text{rem}}(i, j) \triangleq \min\big(D_{\text{rem}}(l, j), R(l, i)\big). \tag{3.1}$$

In other words, up to $I^{(l)}_{\text{rem}}(i, j)$ "passengers" could be booked on "earlier flights" from l to i that have $R(l, i)$ "available seats", and "speculatively stand by" for a "flight" from i to j that might materialize as a part of matching M_k.

The matrix element $I_{\text{rem}}(i, j)$ is the total number of "nonlocal passengers" who are originated at all "airports" other than i and j and are on the "speculative standby list" for a

possible "flight" from i to j. In other words, we have

$$I_{\text{rem}}(i,j) \triangleq \sum_{l \in [n] \setminus \{i,j\}} I^{(l)}_{\text{rem}}(i,j). \tag{3.2}$$

Recall that $D_{\text{rem}}(i,j)$ is the number of "local passengers" (at i) that need to travel to j. Hence at the "airport" i, a total of $D_{\text{rem}}(i,j) + I_{\text{rem}}(i,j)$ "passengers", "local or nonlocal", could use a "flight" from i to j (if it materializes in M_k). We are now ready to precisely state the different between Eclipse and 2-hop Eclipse: Whereas $\| \min(\alpha M, D_{\text{rem}})\|_1$, the cost-unadjusted utility function used by Eclipse, accounts only for "local passengers", $\| \min(\alpha M, D_{\text{rem}} + I_{\text{rem}})\|_1$, that used by 2-hop Eclipse, accounts for both "local" and "nonlocal passengers".

Note that the term $D_{\text{rem}}(l,j)$ appears in the definition of $I^{(l)}_{\text{rem}}(i,j)$ (Formula (3.1)), for all $i \in [n] \setminus \{l,j\}$. In other words, "passengers" originated at l who need to travel to j could be on the "speculative standby list" at multiple intermediate "airports". This is however not a problem (*i.e.*, will not result in "duplicate bookings") because at most one of these "flights" (to j) can materialize as a part of matching M_k.

3.4.3 Update D_{rem} and R

After the schedule (M_k, α_k) is determined by the "argmax call" (Line 7 in Algorithm 2) in the k^{th} iteration, the action should be taken on "booking" the right set of "passengers" on the "flights" in M_k, and updating D_{rem} (Line 10) and R (Line 11) accordingly. Recall that we normalize r_c, the service rate of the circuit switch, to 1, so all these flights have $\alpha_k \times 1 = \alpha_k$ "available seats". We only describe how to do so for a single "flight" (say from i to j) in M_k; that for other "flights" in M_k is similar. Recall that $D_{\text{rem}}(i,j)$ "local passengers" and $I_{\text{rem}}(i,j)$ "nonlocal passengers" are eligible for a "seat" on this "flight". When there are not enough seats for all of them, 2-hop Eclipse prioritizes "local passengers" over "nonlocal passengers", because the former is more resource-efficient to serve

than the latter, as explained earlier. There are three possible cases:

(I) $\alpha \leq D_{\text{rem}}(i,j)$. In this case, only a subset of "local passengers" (directly routed traffic), in the "amount" of α_k, are booked on this "flight", and $D_{\text{rem}}(i,j)$ is hence decreased by α. There is no "available seat" on this "flight" so the value of $R(i,j)$ is unchanged.

(II) $\alpha \geq D_{\text{rem}}(i,j) + I_{\text{rem}}(i,j)$. In this case, all "local" and "nonlocal passengers" are booked on this "flight". After all these "bookings", $D_{\text{rem}}(i,j)$ is set to 0 (all "local passengers" traveling to j gone), and for each $l \in [n] \setminus \{i,j\}$, $D_{\text{rem}}(l,j)$ and $R(l,i)$ each is decreased by $I^{(l)}_{\text{rem}}(i,j)$ to account for the resources consumed by the indirect routing of traffic demand (*i.e.*, "nonlocal passengers"), in the amount of $I^{(l)}_{\text{rem}}(i,j)$, from l to j via i. Finally, $R(i,j)$ is increased by $\alpha - \big(D_{\text{rem}}(i,j) + I_{\text{rem}}(i,j)\big)$, the number of "available seats" that remain on this flight after all these "bookings".

(III) $D_{\text{rem}}(i,j) < \alpha < D_{\text{rem}}(i,j) + I_{\text{rem}}(i,j)$. In this case, all "local passengers" are booked on this "flight", so $D_{\text{rem}}(i,j)$ is set to 0. However, different from the previous case, there are not enough "available seats" left on this "flight" to accommodate all $I_{\text{rem}}(i,j)$ "nonlocal passengers", so only a proper "subset" of them can be booked on this "flight". We allocate this proper "subset" proportionally to all origins $l \in [n] \setminus \{i,j\}$. More specifically, for each $l \in [n] \setminus \{i,j\}$, we book $\theta \cdot I^{(l)}_{\text{rem}}(i,j)$ "nonlocal passengers" originated at l on one or more "earlier flights" from l to i, and also on this "flight", where $\theta \triangleq \frac{\alpha - D_{\text{rem}}(i,j)}{I_{\text{rem}}(i,j)}$. Similar to that in the previous case, after these "bookings", $D_{\text{rem}}(l,j)$ and $R(l,i)$ each is decreased by $\theta \cdot I^{(l)}_{\text{rem}}(i,j)$. Finally, $R(i,j)$ is unchanged as this "flight" is full.

We restrict indirect routing to most 2 hops in 2-hop Eclipse because the aforementioned "duplicate bookings" could happen if indirect routing of 3 or more hops are allowed, making its computation not "embeddable" into the Eclipse algorithm and hence much more computationally expensive. This restriction is however by no means punitive: 2-hop indirect routing appears to have reaped most of the performance benefits from indirect routing, as shown in subsection 3.5.3.

3.4.4 Complexities of 2-hop Eclipse

Each iteration in 2-hop Eclipse has only a slightly higher computational complexity than that in Eclipse. This additional complexity comes from Lines 6, 10, and 11 in Algorithm 2. We need only to analyze the complexity of Line 6 (for updating $I^{(l)}_{\text{rem}}$), since it dominates those of others. For each k, the complexity of Line 6 in the k^{th} iteration is $O(kn^2)$ because there were at most $(k-1)n$ "flights" in the past $k-1$ iterations, and for each such flight (say from l to i), we need to update at most $n-2$ variables, namely $I^{(l)}_{\text{rem}}(i,j)$ for all $j \in [n] \setminus \{l,i\}$. Hence the total additional complexity across all iterations is $O(\min(K,n)Kn^2)$, where K is the number of iterations actually executed by 2-hop Eclipse. Adding this to $O(n^{5/2}\log n \log B)$, the complexity of Eclipse, we arrive at the complexity of 2-hop Eclipse: $O(Kn^{5/2}\log n \log B + \min(K,n)Kn^2)$ (see Table 3.1). We found that the execution times of 2-hop Eclipse are only roughly 20% to 40% longer than that of Eclipse, for the instances (scheduling scenarios) used in our evaluations. Also shown in Table 3.1, the computational complexity of Eclipse++ is much higher than those of both Eclipse and 2-hop Eclipse. Here W denotes the maximum row/column sum of the demand matrix. Finally, it is not hard to check that the space (memory) complexity of 2-hop Eclipse is $O(\max(K,n)n)$, which is empirically only slightly larger than $O(n^2)$, that of Eclipse. This $O(Kn)$ additional space is needed to store the residue capacities (of no more than n links in each schedule) induced by each schedule (M_k, α_k).

3.5 Evaluation

In this section, we evaluate the performance of 2-hop Eclipse and compare it with those of Eclipse and Eclipse++, under various system parameter settings and traffic demands. We do not however have Eclipse++ in all performance figures because its computational complexity is so high that it usually takes a few hours to compute a schedule. We do not compare our solutions with Solstice [9] in these evaluations, since Solstice was shown

in [2] to perform worse than Eclipse in all simulation scenarios. For all these comparisons, we use the same performance metric as that used in [9]: the total time needed for the hybrid switch to transmit the traffic demand D.

For our simulations, we use the same traffic demand matrix D as used in other hybrid scheduling works [9, 2]. 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 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

$$D = (\sum_{i=1}^{n_L} \frac{c_L}{n_L} P_i + \sum_{i=1}^{n_S} \frac{c_S}{n_S} P'_i + \mathcal{N}_1) \times 90\% + \mathcal{N}_2 \tag{3.3}$$

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

The parameters c_L and c_S control the 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 [9, 2], 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 (3.3), 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 [9, 2]. 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 [9, 2], 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 [9, 2], 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 [62] in the sense the total traffic volume of these "mice flows" is not insignificant. To incorporate this heavy-tail behavior 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 Equation (3.3).

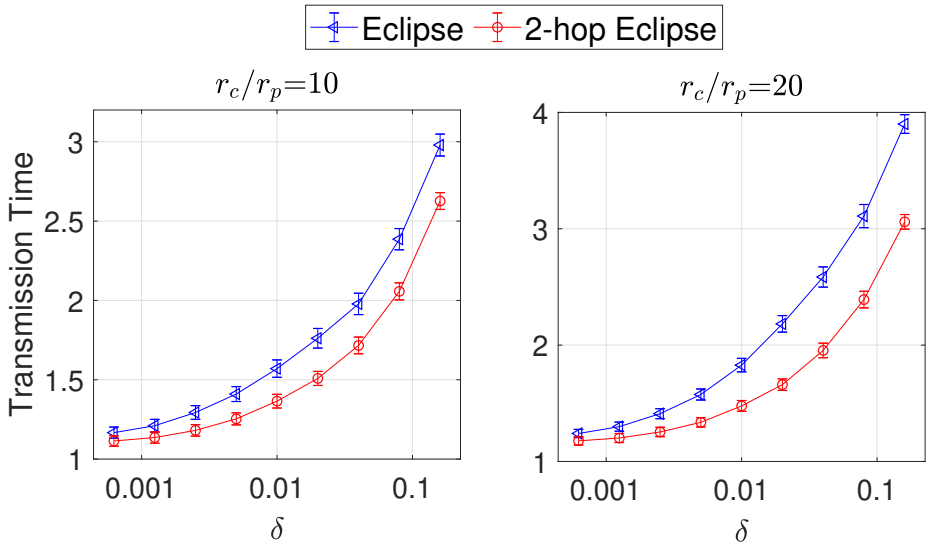

Figure 3.1: Performance comparison under different system settings (varying δ)

3.5.1 Performances under Different System Parameters

We evaluate the performances of Eclipse and 2-hop Eclipse for different value combinations of δ and r_c/r_p under the traffic demand matrix with the default parameter settings.

For each scenario, we perform 100 simulation runs, and report the average transmission time and the 95% confidence interval (the vertical error bar) in Figure 3.1 and Figure 3.2. They show that 2-hop Eclipse performs better than Eclipse, especially when reconfiguration delay δ and rate ratio r_c/r_p are large. For example, when $\delta = 0.01, r_c/r_p = 10$ (default setting), the average transmission time under 2-hop Eclipse is approximately 13% shorter than that under Eclipse, and when $\delta = 0.04, r_c/r_p = 20$, that under 2-hop Eclipse is 23% shorter. The performance of 2-hop Eclipse is also less variable than Eclipse: In all these scenarios, the confidence intervals (of the transmission time) under 2-hop Eclipse are slightly smaller than that under Eclipse.

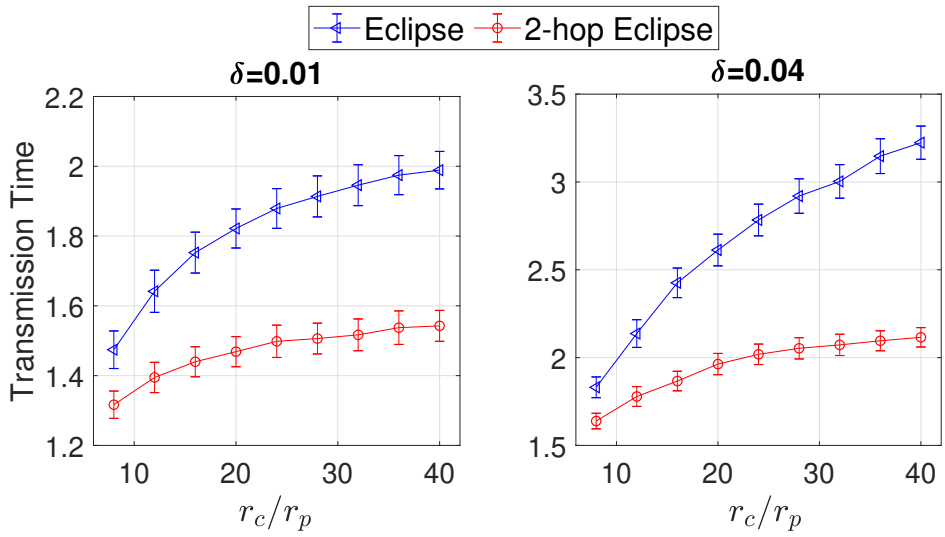

Figure 3.2: Performance comparison under different system settings (varying r_c/r_p)

3.5.2 Performances under Different Traffic Demands

In this section, we evaluate the performance robustness of our algorithm 2-hop Eclipse 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

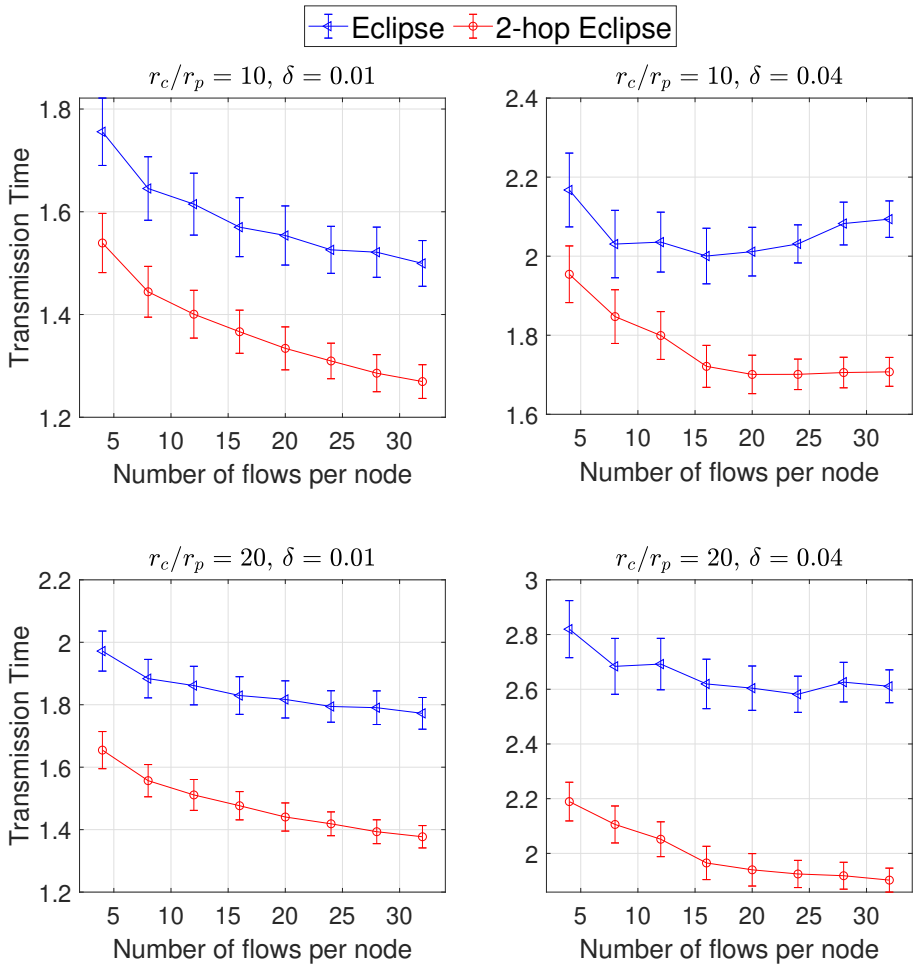

Figure 3.3: Performance comparison while varying sparsity of demand matrix

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 3.3 compares the transmission time of 2-hop Eclipse 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 3.4 compares the transmission time of 2-hop Eclipse 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 $(= 4 + 12)$.

Both Figure 3.3 and Figure 3.4 show that 2-hop Eclipse performs better than Eclipse

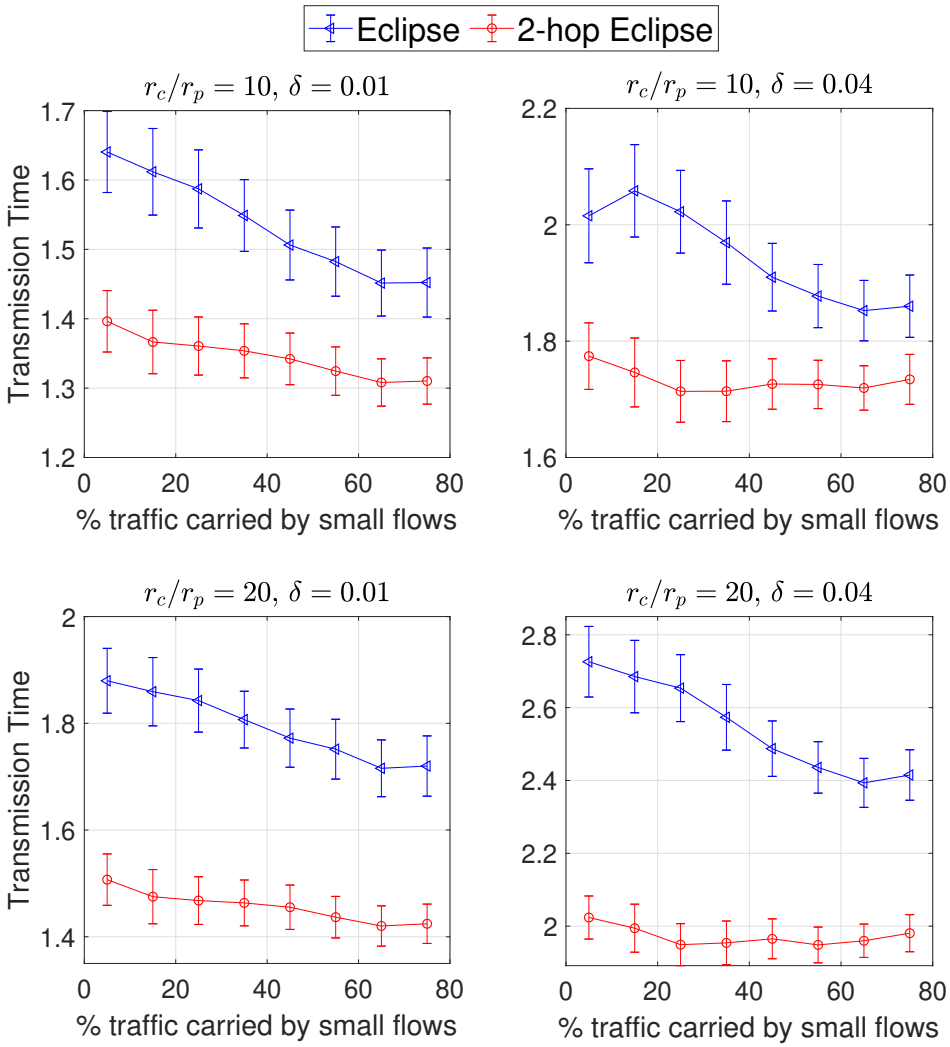

Figure 3.4: Performance comparison while varying skewness of demand matrix

under various traffic demand matrices, especially when the traffic demand matrix becomes dense. This shows that 2-hop indirect routing can reduce transmission time significantly under a dense traffic demand matrix. This is not surprising: Dense matrix means smaller matrix elements, and it is more likely for a small matrix element to be transmitted entirely by indirect routing (in which case there is no need to pay a large reconfiguration delay for the direct routing of it) than for a large one.

3.5.3 Compare 2-hop Eclipse with Eclipse++

In this section, we compare the performances of 2-hop Eclipse and Eclipse++, both indirect routing algorithms, under the default parameter settings.

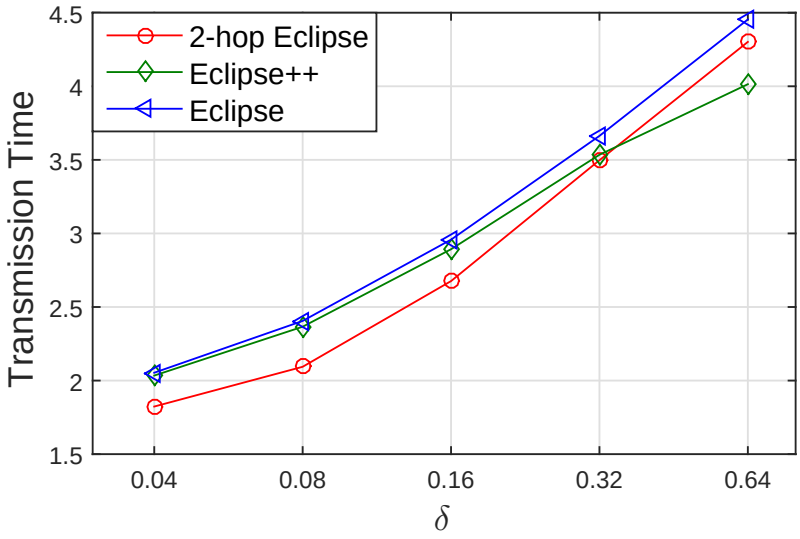

Figure 3.5: Performance comparison of Eclipse, 2-hop Eclipse and Eclipse++

Since Eclipse++ has a very high computational complexity, we perform only 50 simulation runs for each scenario. The results are shown in Figure 3.5. They show that Eclipse++ slightly outperforms 2-hop Eclipse only when the reconfiguration delay is ridiculously large ($\delta = 0.64$ unit of time); note that, as explained earlier, the idealized transmission time is 1 (unit of time)! In all other cases, 2-hop Eclipse performs much better than Eclipse++, and Eclipse++ performs only slightly better than Eclipse.

CHAPTER 4

QUANTIZED BIRKHOFF-VON NEUMANN DECOMPOSITION (QBVND)

In this section, we introduce QBvND, a scheduling algorithm that works for both hybrid switching and standalone optical switching (i.e., no packet switch).

4.1 System Model and Problem Formulation

In this section we describe the formal model for the optical and hybrid switching problems. In both cases we are given an $n \times n$ traffic demand matrix D, and each matrix entry $D(i, j)$ is the amount of traffic that originates at input port (rack) i and is destined for output port (rack) j, within a short (e.g., 3 milliseconds long) scheduling epoch of the recent past (e.g., from 4 milliseconds ago to 1 millisecond ago). Our optical or hybrid switching algorithm needs to meet this demand in the next scheduling epoch. We assume full knowledge of the precise and complete demand matrix D (in this recent past epoch), as do most prior works on hybrid switching and on optical switching.

4.1.1 The Optical Switching Problem

Given a traffic demand matrix D, we aim to compute a schedule that minimizes the *transmission time*, the amount of time for the optical switch to transmit D. An alternative formulation, used in Eclipse [2], would maximize the effective throughput, i.e., the amount of traffic that the optical or hybrid switch can transmit within a scheduling epoch of fixed duration. These two formulations are roughly equivalent, as mathematically the latter is roughly the dual of the former. So do the two corresponding metrics: shorter transmission time implies higher effective throughput and vice versa. Hence we use transmission time as the metric in all performance evaluation plots, but effective throughput as the metric in interpreting these plots.

A schedule of the optical switch consists of a sequence of configurations (matchings) and their durations: (P_1, α_1), (P_2, α_2), $\cdots$, (P_K, α_K). Each configuration P_k is an $n \times n$ permutation (matching) matrix that denotes the k^{th} switch configuration, where $P_k(i,j) = 1$ if input i is connected to output j and $P_k(i,j) = 0$ otherwise. α_k denotes its duration. The transmission time of the above schedule is $\sum_{k=1}^{K}(\alpha_k + \delta)$, where δ is the reconfiguration delay, and K is the number of configurations in the schedule. The optical switching problem aims to minimize this transmission time, under the constraint that the configurations P_k with their respective durations α_k can "sweep clean" the traffic matrix D, or mathematically

$$\min\{K\delta + \sum_{k=1}^{K} \alpha_k\} \text{ such that } D \leq \sum_{k=1}^{K} \alpha_k P_k \tag{4.1}$$

Since this optimization problem is NP-hard [14], almost all optical switching solutions use heuristics.

4.1.2 The Hybrid Switching Problem

In a hybrid switching system, the packet switch is typically an electronic switch, which is an order of magnitude or more slower than the optical switch. For example, the optical and packet switches might operate at the respective rates of 100 Gbps and 10 Gbps per port. However, unlike the optical switch, the packet switch does not incur a reconfiguration delay when its configuration (matching) changes from one switching cycle to the next.

In hybrid switching, the optical switch is allowed to leave a small residue matrix $R \triangleq (D - \sum_{k=1}^{K} \alpha_k P_k)^+$ for the packet switch to handle. Suppose the per-port rate of the packet switch is r_p. Then no row sum or column sum of the reside matrix R can exceed r_p, as otherwise the corresponding input or output port would be given a workload larger than its its capacity. Mathematically, this constraint can be written as

$$\mathbb{1}^T \cdot R \leq r_p \cdot \mathbb{1}^T \text{ and } R \cdot \mathbb{1} \leq r_p \cdot \mathbb{1} \tag{4.2}$$

where $\mathbb{1}$ is a column vector with n scalars that all have value 1, and T stands for transpose. Hence the mathematical formulation of the hybrid switching problem is the same as that of optical switching, except that the constraint in (4.1) is changed to (4.2) above. This optimization problem is also NP-hard, because optical switching is a special case of it (where $r_p = 0$).

4.2 Background on Birkhoff-von Neumann Decomposition

Since QBvND and many optical and hybrid switching solutions are based on Birkhoff-von Neumann Decomposition(BvND), here we provide a brief introduction on it.

4.2.1 Preliminaries

We say that a nonnegative $n \times n$ matrix M is *doubly stochastic* (or doubly sub-stochastic) if every row or column sum of M is equal to 1 (or no larger than 1). The Birkhoff-von Neumann Theorem [18] states that a doubly stochastic matrix M can be expressed as a linear combination of permutation matrices. More precisely, we have

$$M = \sum_{k=1}^{K} \alpha_k P_k \tag{4.3}$$

where $\sum_{k=1}^{K} \alpha_k = 1$ and $P_1, P_2, ..., P_K$ are *permutation matrices*, in which each row or column has exactly one non-zero entry with value 1.

Algorithm 3: BvND

Input : Doubly stochastic matrix M;
Output: BvND of M: $M = \sum_{k=1}^{K} \alpha_k P_k$;

$k \leftarrow 1$;
while M *is not a zero matrix* **do**
  Find a perfect matching P_k in graph M;
  $\alpha_k \leftarrow min\{$weights of the edges $\in P_k\}$;
  $M \leftarrow M - \alpha_k P_k$;
  $k \leftarrow k + 1$;
end

The standard BvND algorithm, which is used in the constructive proof of the Birkhoff-von Neumann Theorem, is shown in algorithm 3. In this algorithm, the matrix M is viewed also as a weighted bipartite graph, with n vertices, denoted as I_1, I_2, ..., I_n, that correspond to the n rows of M in one partite, and another n vertices, denoted as O_1, O_2, ..., O_n that correspond to the n columns of M in the other partite; the bipartite graph contains an edge between I_i and O_j if and only if the (current) value of the matrix element m_{ij} is nonzero, in which case the weight of the edge is m_{ij}.

In each (say k^{th}) iteration, algorithm 3 first finds a perfect matching, which corresponds to a permutation matrix P_k, in the bipartite graph, and then subtracts $\alpha_k P_k$ from M. Here the coefficient α_k is set to the smallest value among the weights of the edges in the perfect matching, so that after the subtraction, at least one previously nonzero matrix element will become zero; once a matrix element becomes zero, the corresponding edge is deleted from the bipartite graph in performing the subsequent iterations. Hence at most n^2 iterations are needed to zero out the matrix M, and algorithm 3 has a total computational complexity of $O(n^{4.5})$, when the classical $O(n^{2.5})$ maximum cardinality matching algorithm [63] is used to find a perfect matching in each iteration.

Let M be a doubly stochastic matrix. We call any uM, where $u > 0$ is a scaling factor, a scaled doubly stochastic matrix. Clearly, any scaled doubly stochastic matrix can also be expressed as a linear combination of permutation matrices, and this decomposition can also be computed using algorithm 3. In this case, the sum of the linear coefficients $\sum_{k=1}^{K} \alpha_k$ is equal to u instead of 1.

4.2.2 The Stuffed BvND Algorithm

A BvND-based algorithm was proposed in [17] for packet switching when the traffic arrival rate matrix Λ is constant or very slowly varying. It consists of two steps, namely stuffing and BvND, so we call it stuffed BvND in the sequel. In the stuffing step, the matrix Λ, which is in general not scaled doubly stochastic, is stuffed into a scaled doubly stochastic

matrix Λ', using the von Neumann stuffing algorithm [64]. In the BvND step, algorithm 3, the standard BvND algorithm, is used to decompose the scaled doubly stochastic matrix Λ' into $\sum_{k=1}^{K} \alpha_k P_k$. To service the incoming traffic, the packet switch simply repeats the following schedule forever (or until Λ changes): use permutation matrix P_1 as the switch configuration for a duration of α_1, P_2 for a duration of α_2, ..., and P_K for a duration of α_K. Since this schedule is precomputed for repeated use, we refer to such an operation as precomputed packet switching in the sequel.

The stuffed BvND algorithm in general works for neither optical nor hybrid switching, because the resulting decomposition would in general consist of $K = O(n^2)$ different configurations, where n is the number of racks, and hence incur a prohibitively high reconfiguration delay cost; it works well for packet switching because the reconfiguration delay there is zero. For this reason, although many existing optical and hybrid switching solutions are based on BvND, they all have to somehow reduce this number K.

4.3 Quantize BvND

In this section, we first provide an overview of QBvND in subsection 4.3.1, then describe the max-min BvND step of QBvND in subsection 4.3.2, and finally discuss how to tune a critical parameter of QBvND in subsection 4.3.3.

4.3.1 Pseudocode of QBvND

In this section, we first provide an overview of QBvND in the context of optical switching, and then describe, toward the end, how to modify it slightly for hybrid switching.

Algorithm 4: QBvND

1 Quantize D to an s-integer matrix $D^{(Q)}$;
2 Stuff $D^{(Q)}$ to a scaled doubly stochastic matrix $D^{(QS)}$;
3 Max-min BvND of $D^{(QS)}$ into $\sum_{k=1}^{K} \alpha_k P_k$;

The pseudocode of our solution, called Quantized BvND or QBvND in short, is shown

in algorithm 4. As its name suggests, QBvND prepends a quantization step before the two other steps described earlier, namely stuffing and BvND. In the quantization step, an appropriate quantization unit $s > 0$ is first chosen and fixed, and then every element D_{ij} in a doubly sub-stochastic traffic demand matrix $D \triangleq (D_{ij})_{n\times n}$ is rounded up to $\lceil \frac{D_{ij}}{s} \rceil \cdot s$, the nearest s-integer, defined as an integral multiple of s. The resulting s-integer matrix is denoted as $D^{(Q)} \triangleq (D^{(Q)}_{ij})_{n\times n}$.

Due to the positive rounding error (no larger than s) added to each matrix element of $D^{(Q)}$ in the rounding-up process, the maximum row or column sum of $D^{(Q)}$ could exceed 1 (i.e., the matrix may not be doubly sub-stochastic any more), but is clearly upper-bounded by $1 + ns$. This maximum row or column sum corresponds to the minimum net (excluding the reconfiguration delays) transmission time $\sum_{k=1}^{K} \alpha_k$ required to serve traffic matrix $D^{(Q)}$. Hence this step is not slack-free in the sense the minimum net transmission time in general increases after the quantization step.

In the second step, the s-integer matrix $D^{(Q)}$ is carefully stuffed into another s-integer matrix $D^{(QS)}$ that is at least as large (i.e, $D^{(QS)} \geq D^{(Q)}$) and is scaled doubly stochastic. This problem, known as matrix stuffing, has been studied in [64, 9, 65]. Among many different such algorithms, we use for our solution QBvND a greedy heuristic algorithm, called QuickStuff, that was used in the Solstice algorithm [9] for hybrid switching, because it possesses two desirable properties that others generally don't. The first property is that, if the input M is an s-integer matrix (hence ϕ is an s-integer), the output computed by QuickStuff is also an s-integer matrix. Preserving this "s-integrality" during the stuffing step is necessary for the next step (BvND) to cap K, the number of configurations (matchings) in the resulting schedule, at $O(n)$. The second property is that, QuickStuff avoids, to the extent possible, converting a zero element to a nonzero element and thereby decreasing the sparsity of the matrix. It is desirable for the traffic demand matrix to be sparse, since the BvND of a sparse matrix generally contains a smaller number of configurations than that of a dense one, resulting in a smaller total reconfiguration delay. Finally, we note that all

stuffing algorithms, including QuickStuff, guarantees to be slack-free in the sense the maximum row or column sum of $D^{(QS)}$ is the same as that of $D^{(Q)}$ (so the net transmission time remains unchanged after the stuffing). We refer readers to [9] for a detailed description of QuickStuff.

The last step is the BvND of $D^{(QS)}$. Now since $D^{(QS)}$ is an s-integer scaled doubly stochastic matrix, it can be expressed as a linear combination of at most $O(n)$ (instead of $O(n^2)$) permutation matrices, whose corresponding durations are s-integers, as follows. Suppose for the moment that algorithm 3 is used for the decomposition. Then all coefficients α_k, $k = 1, 2, \cdots, K$, are positive s-integers for the following reason. In the first iteration of algorithm 3, the coefficient α_1 computed from line 4 must be a positive s-integer, as it is the minimum of a set of s-integer edge weights, so $D^{(QS)}$ remains an s-integer matrix after the subtraction of $\alpha_1 P_1$ from it. Hence α_2 is also a positive s-integer, and so on. Since each $\alpha_k P_k$ takes away at least weight s from any row or column in $D^{(QS)}$, whose sum is upper-bounded by $1 + ns$ as explained earlier, there are at most $(ns + 1)/s$ configurations in the BvND of $D^{(QS)}$. Since s is set to $\Omega(1/n)$ or larger, the number of configurations K is at most $O(n)$.

To further reduce the total number of configurations K, we use the max-min BvND algorithm [19], instead of algorithm 3, to decompose $D^{(QS)}$. The resulting K is also $O(n)$, but with a much smaller constant factor. The computational complexity of the last step (max-min BvND) is $O(n^{3.5})$. Since it dominates the complexities of the other two steps, which are both $O(n^2)$, the overall complexity of QBvND is $O(n^{3.5})$.

Finally, to use QBvND for hybrid switching, only the following slight modification needs to be made to its last step (max-min BvND): the iterative process of searching for max-min matchings can stop as soon as what remains of the quantized stuffed matrix $D^{(QS)}$ can be handled by the packet switch.

4.3.2 A Modified Max-Min BvND Algorithm

In this section, we describe the last step of our QBvND solution: max-min BvND [19]. The max-min BvND algorithm differs from algorithm 3 only in that, in each iteration (say k^{th}), whereas the latter finds an arbitrary full matching P_k (Line 4), the former finds a P_k that maximizes the value of the corresponding α_k. Such a matching is called max-min matching because α_k is the minimum among the weights of the edges in P_k, and we are trying to maximize this minimum. As a result, max-min BvND can extract most or all of the traffic from the traffic demand matrix using much fewer configurations than almost all other BvND algorithms.

Now we describe the algorithm, proposed in [66], for computing a max-min matching P in the weighted bipartite graph that corresponds to a nonnegative matrix M. With a slight abuse of notation, we denote this bipartite graph also as M. This algorithm can be best explained using the following alternative formulation of this computation problem. Consider the pruning of the graph M according to a threshold $\epsilon > 0$ as follows: an edge is removed from the graph M if and only if its weight is less than ϵ. We denote the resulting pruned graph as M_ϵ. This computing problem can be restated as finding the maximum ϵ for M_ϵ to contain a full matching (which is exactly the max-min matching P we are looking for). This algorithm is simply to binary-search for this ϵ in the interval $[0, \epsilon_{max}]$, where ϵ_{max} is the value of the largest element in the matrix M, as follows: at each binary search point ϵ', the search is considered successful if a full matching can be found using the classical $O(n^{2.5})$ maximum cardinality matching (MCM) algorithm [63]. The computational complexity of finding a single max-min matching is $O(n^{2.5} \log n)$, since $\log(\epsilon_{max})$ binary search steps are needed, each of which involves a MCM computation, and $\log(\epsilon_{max})$ is $O(\log n)$ in our context.

Using this max-min matching algorithm, the total complexity of the max-min BvND step in QBvND would be $O(n^{3.5} \log n)$, since it needs to run this algorithm at most $O(n)$ times thanks to the quantization. In QBvND, by doing away with the binary search, we

reduce this complexity further by a factor of $O(\log n)$ as follows. The modified algorithm conducts a linear search, starting from the value of the largest matrix element in $D^{(QS)}$ (also denoted as ϵ_{max}), which is an s-integer no larger than $1 + s$ (since the original traffic demand matrix D is sub-stochastic), for all $O(n)$ max-min matchings. In other words, it searches for and extracts full matchings *exhaustively* (i.e., until a full matching can no longer be found) at each of the $O(n)$ arithmetically progressing graph-pruning thresholds ϵ_{max}, $\epsilon_{max} - s$, $\epsilon_{max} - 2s$, $\cdots$, $2s$, and s.

Since the modified algorithm conducts at most $O(n)$ successful searches, each of which results in a max-min matching and has complexity $O(n^{2.5})$, and at most $O(n)$ unsuccessful searches (once at each of the $O(n)$ graph-pruning thresholds), its total complexity is $O(n^{3.5})$. In QBvND, we also significantly reduce the constant factor inside this big-O, by increasing the unit step of the linear search (i.e., the arithmetic progression) from s to $5s$. In doing so, we observe only a negligible degradation in the qualities (i.e., the durations and the number) of the resulting configurations (matchings). Finally, we note this linear search trick can be used here only because the matrix $D^{(QS)}$ is s-integral. Otherwise, the algorithm would have to linearly search through all distinct values, in the decreasing order, among the nonzero matrix elements (in general $O(n^2)$ of them), resulting in a total complexity of $O(n^{4.5})$.

4.3.3 Theoretical Analysis and Quantization Unit Selection

In this section, we explain how to approximately maximize the throughput performance (or equivalently minimize the transmission time) of QBvND by tuning the quantization unit s. The (gross) transmission time of an optical switch schedule can be split into two parts: the *net* transmission time $\sum_{k=1}^{K} \alpha_k$ and the reconfiguration cost $K\delta$. As explained in subsection 4.3.1, in optical switching, the first part is upper-bounded by $1 + ns$, and K is upper-bounded by $(1 + ns)/s$, so the second part is upper-bounded by $(1 + ns)\delta/s$. Hence the (gross) transmission time is upper-bounded by $1 + ns + \frac{1+ns}{s}\delta = 1 + n\delta +$

$ns + \delta/s$, which reaches the minimum when $s = \sqrt{\delta/n}$. Although this upper bound is not exactly the objective function we would like to minimize, it is reasonable to use this optimal parameter setting $\sqrt{\delta/n}$ "asymptotically", that is, to set s to $\beta\sqrt{\delta/n}$ where $\beta > 0$ is a tunable parameter.

It remains to explain how to set this β. In optical switching, the second part $K\delta$ (upper-bounded by $(1 + ns)\delta/s$) is quite large so we would like to make s larger to reduce it. In this case, we set β to a value larger than 1 (e.g., set to $\sqrt{2}$ in section 7.2). In hybrid switching, however, the second part becomes smaller since the packet switch can absorb a residue matrix that would otherwise have to be swept clean by a fairly large number of configurations with short durations. Hence, it becomes less important to reduce the second part by increasing s. In this case, we choose $\beta < 1$ (e.g., set to 0.3 in section 7.2).

4.4 Closely Related Works

In this section, we describe several (precomputed) packet, optical, and hybrid switching algorithms that are most related to QBvND and we will evaluate QBvND against in section 7.2. We will also compare their computational complexities against that of QBvND in subsection 4.4.3.

4.4.1 Precomputed Packet Switching Algorithms

We are aware of two other quantized BvND algorithms, namely RQ (Rate Quantization) [67] and FBD (Frame-Based Decomposition) [68]. Both were proposed, more than a decade ago, for precomputed packet switching (PPS). They both address the following problem with Chang et. al's original solution (i.e., the stuffed BvND algorithm described in subsection 4.2.2) [17] for PPS: among the numerous (more specifically $O(n^2)$) configurations $\{(P_k, \alpha_k)\}_{k=1}^{K}$ contained in the resulting decomposition, most of them are very short in duration (i.e., with a tiny α_k), and moreover much shorter than a switching cycle of the packet switch (called a frame in FBD). However, each of these short configurations

still has to be covered by a full switching cycle, resulting in a poor utilization of the packet switch bandwidth.

In both RQ and FBD, the sole purpose of quantization is to reduce the number of such severely underutilized switching cycles. Although as an unintended side benefit of the quantization, both algorithms also reduce the number of configurations in the resulting decomposition, this number is still too large for bandwidth-efficient optical switching, as we will show in subsection 4.5.4. Note this is not a shortcoming of either algorithm: there is no incentive to reduce the number of configurations in packet switching, where the reconfiguration delay is zero.

FBD uses the same quantization method (of rounding up to an s-integer where s is set to the length of a switching cycle in FBD) as QBvND, but performs the decomposition differently. Whereas QBvND greedily finds and extracts, in each iteration, a fully utilized configuration with maximum possible duration (i.e., a max-min matching), FBD sets the duration of every configuration to the minimum value: one switching cycle (i.e., s). Hence a (switch) schedule generated by FBD typically contains several times more configurations than a schedule generated by QBvND, as we will show in subsection 4.5.4, and consequently incurs a much higher reconfiguration cost if used for optical switching. The motivation for FBD to use the same minimum duration s for each configuration, as stated in [68], is that the resulting computation problem can be modeled as edge coloring of a bipartite multigraph, which has much faster algorithmic solutions (e.g., with $O(n^2 \log n)$ complexity using [69]), than BvND (roughly $O(n^{3.5})$ with quantization).

RQ uses a slightly different quantization method than QBvND. It splits the traffic demand matrix into an s-integer scaled doubly stochastic matrix and an "s-fractional" residue matrix, the value of each element in which is a nonnegative value less than s. It then performs the standard BvND of the former matrix using algorithm 3 and "covers" the latter using n configurations (e.g., the n cyclic shifts of a permutation matrix) each with duration s. However, this n is already several times larger than the number of configurations needed

by QBvND.

4.4.2 Optical Switching Algorithms

Scheduling of optical switch alone has been studied for decades. In early works the reconfiguration delay is often assumed to be either zero [25, 26, 13] or infinity (a very large value) [26, 27, 28]. In later works, such as DOUBLE [26], ADJUST [14] and [27, 29], the reconfiguration delay is usually assumed to be finite and nonzero. Towles et al. [26] proposed three different scheduling algorithms, EXACT, MIN, and DOUBLE, for the optical switches with zero, infinite, and nonzero finite reconfiguration delays respectively. EXACT is identical to the stuffed BvND algorithm described in subsection 4.2.2, which is very bandwidth-inefficient for optical switching when the reconfiguration delay is high, due to the very large number ($O(n^2)$ in general) of configurations it has to use.

MIN is proposed to significantly reduce the number of configurations to at most n. Its algorithm is identical to algorithm 3 except that in line 4, α_k is set to the maximum (instead of the minimum in algorithm 3) among the weights of edges in P_k. This way, after the subtraction of $\alpha_k P_k$, n matrix elements become zero. Hence only n configurations are needed to "cover" the matrix. The flip side of the coin however is that most of the n input-output connections in such a configuration can be severely underutilized (i.e., contain considerable slack), as their corresponding edge weights can be much smaller than this maximum.

DOUBLE is a compromise between the two extremes EXACT and MIN. Each schedule computed by DOUBLE uses at most $2n$ configurations, each of which has the same duration $1/n$, to "cover" a doubly sub-stochastic traffic demand matrix. The algorithm is named DOUBLE because the total duration of these configurations is bounded at 2 ($= 2n * 1/n$), and any set of configurations that "cover" a doubly sub-stochastic matrix has a total duration of at least 1. In DOUBLE, each matrix element D_{ij} is rounded down to the closest $1/n$-integer $\lfloor nD_{ij} \rfloor / n$ called the quotient, and their difference is called the residue. This

way, the matrix D is split into a quotient matrix and a residue matrix. Each matrix can be covered by at most n configurations, each of which has duration $1/n$. The residue matrix, like in RQ, can be covered by at most n configurations, that are cyclic shifts of one another, and each has the same duration $1/n$. The quotient matrix, viewed as a bipartite multigraph like in FBD, is decomposed into at most n configurations, each also with duration $1/n$, using an edge-coloring algorithm such as [69].

ADJUST [14] is different than DOUBLE only in that ADJUST sets the value of quantization unit s to $\sqrt{\delta/n}$ (like we did in subsection 4.3.3) to strike the optimal compromise between EXACT and MIN. ADJUST is the same as DOUBLE when the configuration delay δ is equal to $1/n$, since both are using the same quantization unit $\sqrt{\delta/n} = 1/n$ in this case.

4.4.3 Computational Complexity Comparisons

Table 4.1: Complexities of various algorithms

	Algorithms	Complexities
Optical Switching	DOUBLE [26]	$O(n^2 \log n)$
	ADJUST [14]	$O(n^2 \log n)$
	MIN [26]	$O(n^{3.5})$
	EXACT [26]	$O(n^{4.5})$
Hybrid Switching	Solstice [9]	$O(Kn^{2.5})$
	Eclipse [2]	$O(Kn^{2.5} \log^2 n)$
Precomputed Packet Switching	BvND [17]	$O(n^{4.5})$
	FBD [68]	$O(n^2 \log n)$
	RQ [67]	$O(n^{4.5})$

Table 4.1 summarizes the computational complexities of the optical and the hybrid switching algorithms that QBvND will be compared against in section 7.2. In the complexities of Solstice and Eclipse, K denotes the total number of configurations in the schedule. Although K is not too large (roughly $O(n)$) in our simulation scenarios, it could be as large as $O(n^2)$ in the worst case. Hence the worst case complexities of Solstice and Eclipse are both higher than that of QBvND, which is $O(n^{3.5})$. Compared to optical switching

algorithms, although QBvND has a higher computational complexity than DOUBLE and ADJUST, it outperforms all four of them, in terms of throughput performance, by a wide margin, as will be shown in section 7.2.

4.5 Evaluation

Since QBvND works for both hybrid switching and standalone optical switching, we will evaluate the performance of QBvND in both of these two circumstances and compare it with the state of the art hybrid and standalone optical scheduling algorithms, under various system parameter settings and traffic demands.

4.5.1 Traffic Demand Matrix D

As shown in [9, 2], the typical workloads we see in data centers exhibit two characteristics: sparsity (the vast majority of the demand matrix elements have value 0 or close to 0) and skewness (few large elements in a row or column account for the majority of the row or column sum). Hence, for our simulations, we use the same set of sparse and skewed demand matrices as used in [9, 2]. In each such matrix D, 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 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. Hence n_L and n_S control the sparsity, and c_L and c_S control the skewness, of the traffic demand, respectively. Roughly speaking, we have

$$D = \sum_{i=1}^{n_L} \frac{c_L}{n_L} P_i + \sum_{i=1}^{n_S} \frac{c_S}{n_S} P'_i + \mathcal{N} \tag{4.4}$$

where each P_i and each P'_i is an $n \times n$ random permutation matrix.

Same as in [9, 2], in our simulation studies, the default values of the sparsity parameters n_L and n_S are set to 4 and 12 respectively and the default values of c_L and c_S are set to 0.7

(*i.e.*, 70%) and 0.3 (*i.e.*, 30%) respectively. In other words, in each row (or column) of the demand matrix, by default the 4 large flows account for 70% of its total traffic demand, and the 12 medium flows account for the rest 30%. We will also vary the sparsity parameters n_L and n_S and skewness parameters c_L and c_S in our evaluations. In Equation (4.4), before a noise matrix $\mathcal{N}$ (described next) is added to it, each such D is doubly stochastic, that is, the sum of each row or column of it is 1. This normalized workload 1 is defined as the amount of traffic that an *established* (after paying for the reconfiguration cost) optical switch connection/link, which we assume to have a normalized rate also of 1, can transmit in 1 unit of time, defined as the length of a scheduling epoch (e.g., 3 milliseconds).

As shown in Equation (4.4), we also add a noise matrix term $\mathcal{N}$ to D, like in [9, 2]. Each nonzero element in $\mathcal{N}$ is a Gaussian random variable that is added to a traffic demand matrix element that was nonzero before the noise added. Each nonzero (noise) element here in $\mathcal{N}$ has a standard deviation, which is equal to 0.3% of the normalized workload 1.

4.5.2 System Parameters

In this section, we introduce the system parameters, for both optical and hybrid switching, used in our simulations.

Network size: By default, both the optical switch and the packet switch (if applicable) has $n = 100$ input/output ports (i.e., 100 racks of servers in the data center) although we will vary n in subsection 4.5.6. Other reasonably large (say ≥ 32) switch sizes produce similar results.

Reconfiguration delay of the optical switch δ: In both optical and hybrid switching, the larger this reconfiguration delay is, the more time the optical switch has to spend on reconfigurations, and hence the higher the transmission time is. By default, $\delta = 0.01$ (i.e., $1/100$ of the scheduling epoch), although we will vary δ in our simulation studies.

Optical switch per-port rate $r_c = 1$ 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. The

higher this ratio is, the higher percentage of traffic should be transmitted by the optical switch. This ratio varies from 8 to 40 in our simulations. As explained earlier, we normalize r_c to 1.

The largest row or column sum in a (random) demand matrix D generated using Equation 4.4, also a random variable, has an expectation that is roughly equal to 1.0325, where the fractional part 0.0325 comes from the noise matrix $\mathcal{N}$. In optical switching, since $r_c = 1$, even with perfect scheduling and zero reconfiguration delay, the total transmission time is at least 1.0325. In hybrid switching, the transmission time could be smaller than 1, thanks to the switching and transmission capacity of the packet switch, although that never materialized in our simulation studies, likely due to the nontrivial reconfiguration delays.

In the rest of section 7.2, every point in every plot in every figure is the sample mean averaged from 100 simulation runs, so is every number in Table 4.2 and Table 4.3.

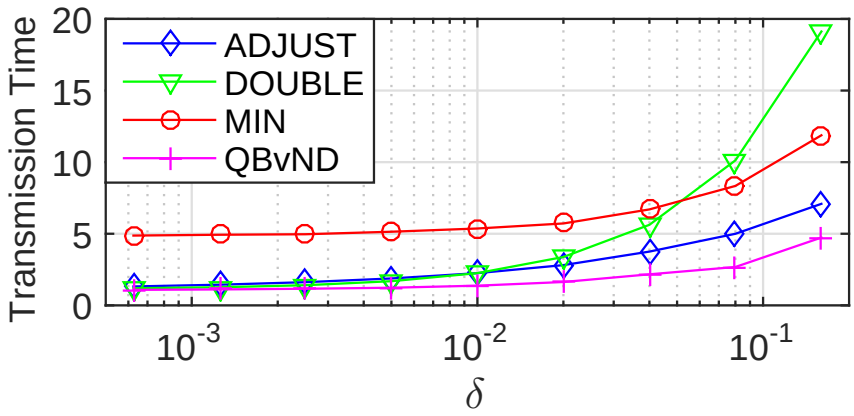

Figure 4.1: Comparison while varying the reconfiguration delay (optical)

4.5.3 QBvND vs. Others for Optical Switching

In this section, we compare QBvND, as an standalone optical switching solution, with three state of the art optical switching algorithms: ADJUST [14], DOUBLE [26], and MIN [26]. Note that when $\delta = 0.01$ (i.e., $\delta = 1/n$ since $n = 100$), ADJUST is the same as DOUBLE, as we explained in subsection 4.4.2. Hence there is a combined plot for ADJUST/DOUBLE in Figure 4.2.

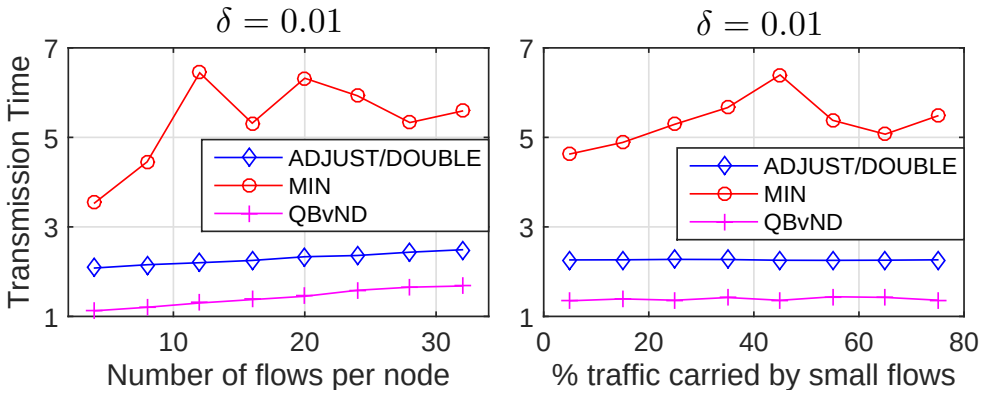

Figure 4.2: Comparison under various demand matrices (optical)

The simulation results, shown in Figure 4.1 and Figure 4.2, demonstrate that the schedules generated by QBvND are consistently better (*i.e.*, shorter transmission times) than those generated by ADJUST, DOUBLE, and MIN. More specifically, when $\delta = 0.01$ (default setting) and $\delta = 0.04$, the average transmission times of the schedules generated by QBvND are roughly 40% shorter than those of schedules generated by ADJUST, the best among others.

4.5.4 An "Anatomic" Comparison of Transmission Time

To better understand the reason why the QBvND outperforms all other optical switching algorithms by a wide margin, we split the (gross) transmission time into the aforementioned two components: the net transmission time $\sum_{k=1}^{K} \alpha_k$ and the reconfiguration cost $K\delta$. Table 4.2 separately compares these two components in schedules computed by different algorithms under the default setting (4 large flows and 12 small flows accounting for roughly 70% and 30% of the total traffic demand into each input port, $\delta = 0.01$). Besides ADJUST, DOUBLE and MIN, we compare QBvND also with three other algorithms: FBD [68], RQ [67], and EXACT [26]. Note that FBD and RQ are proposed originally for precomputed packet switching, but here we view them as optical switching algorithms and impose the reconfiguration delay on them. ADJUST is not in the table since it is equivalent to DOUBLE under this parameter setting. The simulation results are summarized in

Table 4.2.

Table 4.2: Transmission time comparison of optical switching algorithms

Algorithm	$K\delta$	$\sum_{k=1}^{K} \alpha_k$	$K\delta + \sum_{k=1}^{K} \alpha_k$
DOUBLE	1.1245	1.1245	2.2490
MIN	0.4403	4.8337	5.2740
EXACT	12.98	1.0325	14.0125
FBD	0.8329	1.1697	2.0026
RQ	1.7252	2.0185	3.7437
QBvND	0.2294	1.1457	1.3751

As expected, the average net transmission time $\sum_{k=1}^{K} \alpha_k$ under EXACT is equal to 1.0325, the aforementioned theoretical minimum, since it is precisely the stuffed BvND algorithm (described in subsection 4.2.2). However, its reconfiguration cost is humongous (12.98), due to the large number of configurations in the resulting BvND. The net transmission time under QBvND (1.1457) is only slightly larger than the theoretical minimum 1.0325 and smaller than that of all other algorithms. Its reconfiguration cost (0.2294) is smaller, by at least 70%, than that of all other algorithms except MIN, whose net transmission time (4.8337) is extremely high.

4.5.5 QBvND vs. Solstice and Eclipse for Hybrid Switching

In this section, we compare QBvND, as a hybrid switching solution, with the two state of the art algorithms, Eclipse [2] and Solstice [9].

The simulation results demonstrate that the schedules generated by QBvND are better than those generated by Solstice, especially when the reconfiguration delay δ or the rate ratio r_c/r_p is large. More specifically, as shown in Figure 4.3(left), when $\delta = 0.04, r_c/r_p = 10$, QBvND results in 8% shorter average transmission time than Solstice; as shown in Figure 4.3(right), when $\delta = 0.01, r_c/r_p = 32$, QBvND results in 11% shorter average transmission time than Solstice. On the other hand, QBvND results in slightly longer (approximately 5% longer) average transmission times in both cases than Eclipse.

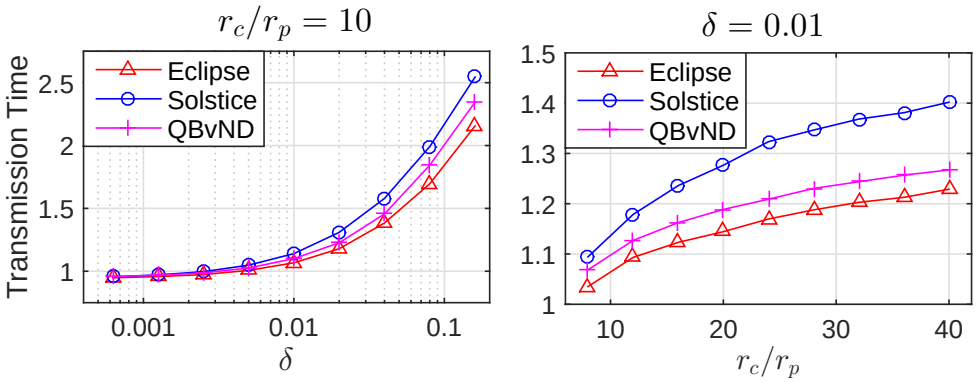

Figure 4.3: Comparison under different system settings (hybrid)

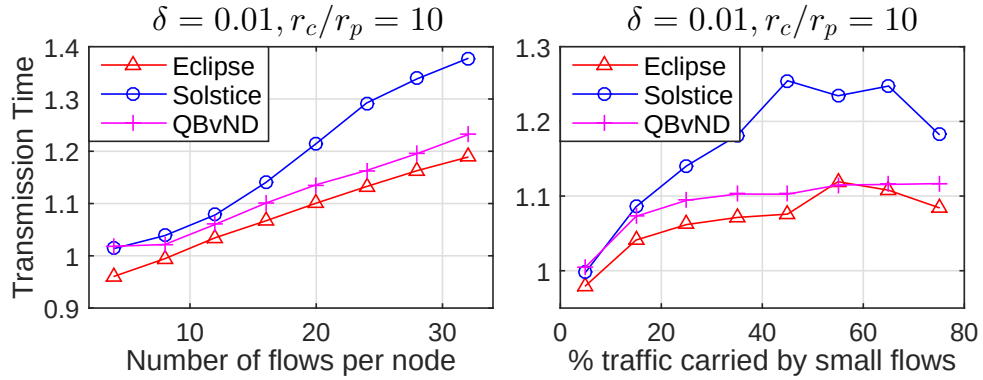

Figure 4.4: Comparison under various demand matrices (hybrid)

4.5.6 Execution Time Comparison

In this section, we present the execution times of Eclipse, Solstice, and QBvND (all implemented in C++) for three different δ values (0.0025, 0.01, and 0.04), 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 to 10 in each scenario. These execution time measurements, shown in Table 4.3, are performed on a Dell Precision Tower 3620 workstation equipped with an Intel Core i7-6700K CPU @4.00GHz processor with 16GB RAM, and running Windows 10 Professional.

As shown in Table 4.3, the average execution time of QBvND is roughly 20 times smaller than that of Eclipse under the default setting ($n = 100$). As n increases to 200, QBvND outperforms Eclipse even more (e.g., 100 times when $\delta = 0.01$) in terms of exe-

Table 4.3: Comparison of average execution time

	δ	Eclipse	QBvND	Solstice
$n = 50$	0.0025	$152.3ms$	$52.75ms$	$33.23ms$
	0.01	$141.5ms$	$24.90ms$	$33.83ms$
	0.04	$128.8ms$	$19.45ms$	$25.95ms$
$n = 100$	0.0025	$1.430s$	$111.67ms$	$76.07ms$
	0.01	$1.282s$	$55.65ms$	$72.67ms$
	0.04	$0.943s$	$48.15ms$	$59.56ms$
$n = 200$	0.0025	$17.57s$	$263.6ms$	$182.8ms$
	0.01	$12.37s$	$139.1ms$	$165.8ms$
	0.04	$7.384s$	$111.2ms$	$134.0ms$

cution time. Meanwhile, QBvND's average execution time is only slightly larger than that of Solstice when $\delta = 0.0025$. When $\delta = 0.01, 0.04$, QBvND is even faster than Solstice.

CHAPTER 5

BEST-FIRST-FIT (BFF): TOWARDS PARTIALLY RECONFIGURABLE HYBRID SWITCHING FOR DATA CENTERS

5.1 System Model and Problem Formulation

In this work, we study this problem of hybrid switch scheduling under the following standard formulation that was introduced in [9]: to minimize the *transmission time* for the circuit and the packet switches working together to transmit a given traffic demand matrix D.

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.

5.2 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 [20, 21]), 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.

For an optical switch that has partial reconfigurability, its schedule is no longer restricted to a sequence of configurations with durations (M_1, α_1), (M_2, α_2), $\cdots$, (M_K, α_K). Instead, it can be represented by an $n \times n$ matrix process $S(t) = s_{ij}(t)$, where for any give 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 port 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.

5.3 Open Shop Scheduling Problem

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). 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 [70], so only heuristic or approximate solutions to it [31, 32, 33] exist that run in polynomial time.

5.4 LIST: A Family of Heuristics

LIST (list scheduling) is a well-known family of polynomial-time heuristic OSS algorithms [31, 32, 33]. 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 [33] 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 [26]. In particular, it was shown in [26] 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.

5.5 Best-First-Fit (BFF)

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

Algorithm 5: 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**:**
    Output_Seek_Pairing(j, τ);
    Input port i reconfigures during $[\tau, \tau + \delta]$;
    Input_Seek_Pairing$(i, \tau + \delta)$;
Procedure *Output_Seek_Pairing*(j, t)
    Update D_{rem};
    if $\{l \in I_a \mid D_{rem}(l, j) > 0\} \neq \phi$ **then**
        $l = \arg\max_u D_{\text{rem}}(u, j)$;
        Connect output j with input l;
    else
        $O_a \leftarrow O_a \cup \{j\}$;
    end
Procedure *Input_Seek_Pairing*(i, t)
    Update D_{rem};
    if $\{j \in O_a \mid D_{rem}(i, j) > 0\} \neq \phi$ **then**
        $j = \arg\max_v D_{\text{rem}}(i, v)$;
        Connect input i with output j;
    else
        $I_a \leftarrow I_a \cup \{i\}$;
    end

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 5. 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 "Out-

put_Seek_Pairing(j, τ)" (Line 2 in Algorithm 5). 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 4 in Algorithm 5). 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 4 in Algorithm 5) so that the actual transmission can start at time $\tau + \delta$.

In Algorithm 5, 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 6 through 12) 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 14 through 20). 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 [56]) 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 5.1). Empirically, BFF runs about three orders of magnitude faster than Eclipse when $n = 100$, as will be shown in subsection 5.6.4.

Wondering whether allowing indirect routing can bring further performance improve-

Table 5.1: 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	$O(Kn^2 + n^{5/2} \log B)$

ments, 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.

5.6 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 [9] in these evaluations, since Solstice was shown in [2] to perform worse than Eclipse in all simulation scenarios. For all these comparisons, we use the same performance metric as that used in [9]: the total time needed for the hybrid switch to transmit the traffic demand D.

For our simulations, we use the same traffic demand matrix D as used in other hybrid scheduling works [9, 2]. 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 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

$$D = (\sum_{i=1}^{n_L} \frac{c_L}{n_L} P_i + \sum_{i=1}^{n_S} \frac{c_S}{n_S} P'_i + \mathcal{N}_1) \times 90\% + \mathcal{N}_2 \tag{5.1}$$

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 [9, 2], 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 (5.1), 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 [9, 2]. 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 [9, 2], 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 [9, 2], 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 [62] 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 (5.1).

5.6.1 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.

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: Reconfiguration delay is another important parameter of hybrid switch. In general, the smaller this reconfiguration delay is, the less time the circuit switch has to spend on reconfigurations and the better performance hybrid switch achieves. Hence, given a traffic demand matrix, the transmission time should increase as δ increases.

5.6.2 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 5.1 and Figure 5.2.

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 5.1 and Figure 5.2, 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.

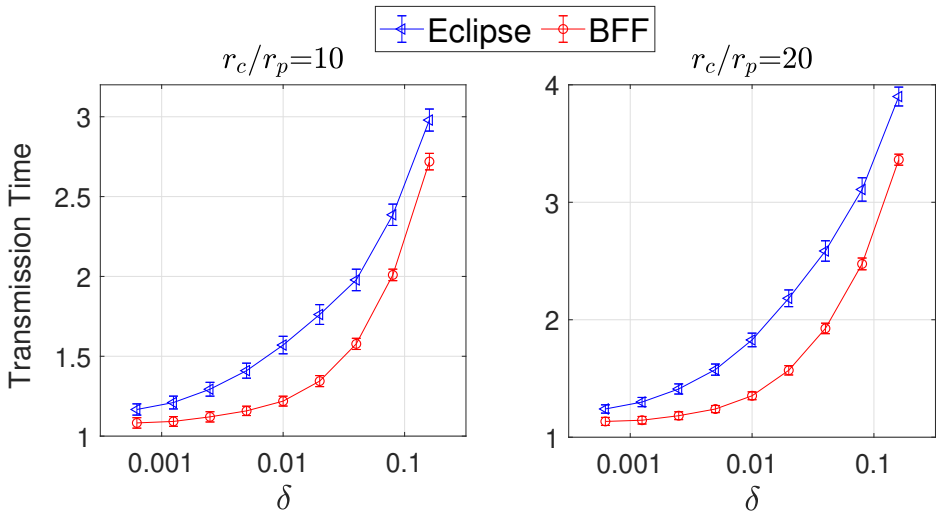

Figure 5.1: Performance comparison under different system settings (varying δ)

5.6.3 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

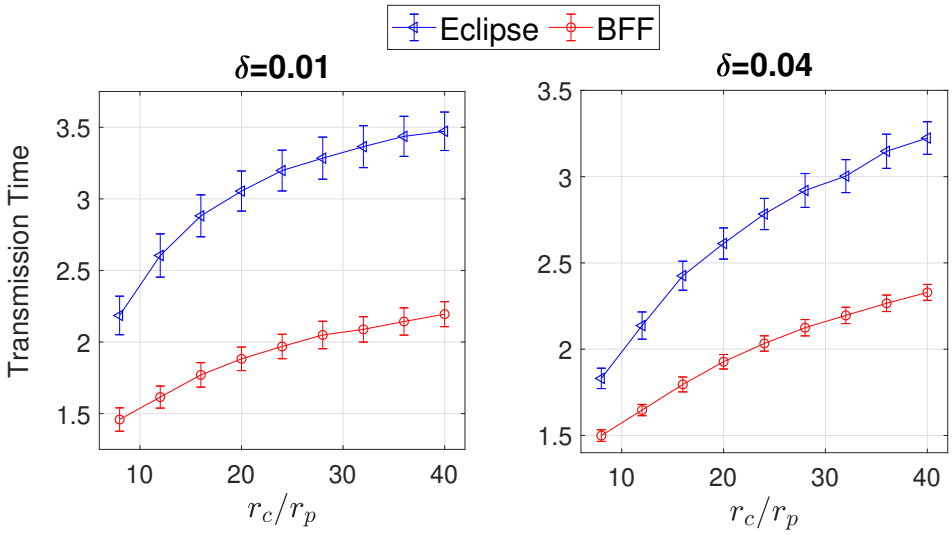

Figure 5.2: Performance comparison under different system settings (varying r_c/r_p)

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 5.3 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 5.4 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 5.3 and Figure 5.4 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.

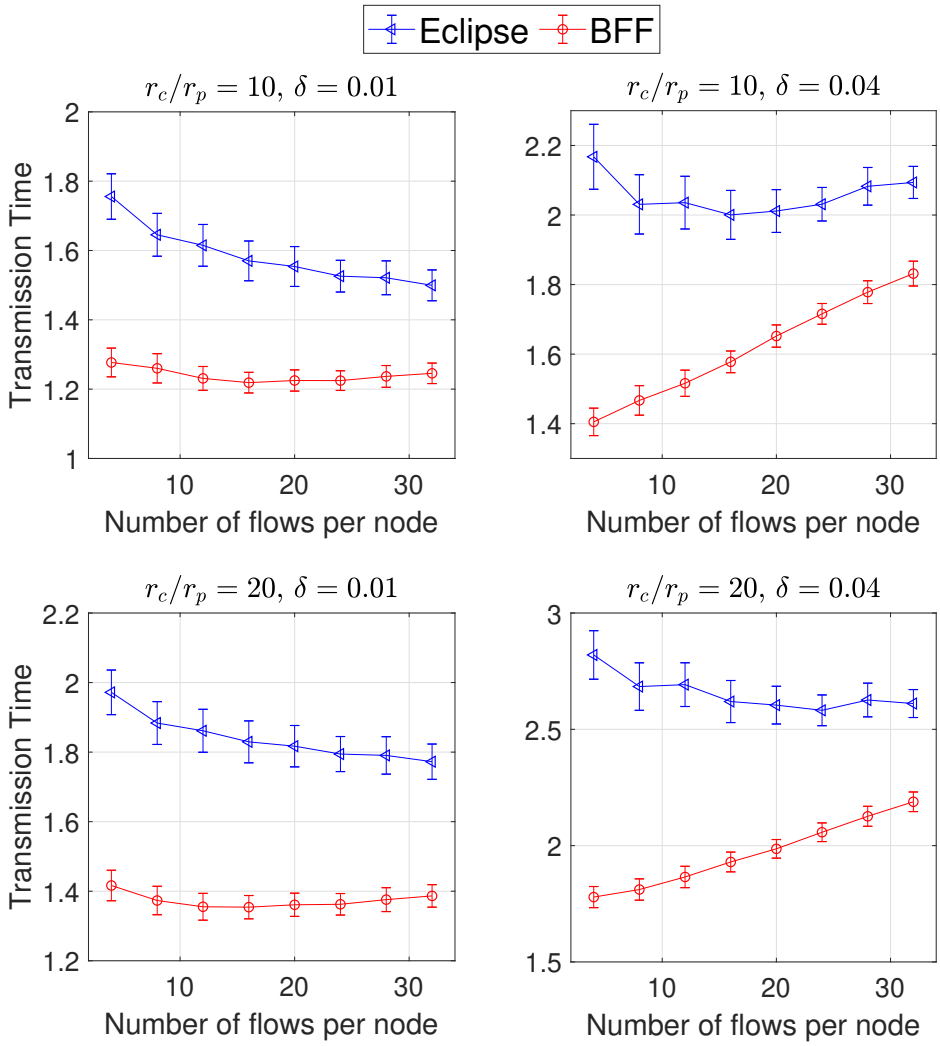

Figure 5.3: Performance comparison while varying sparsity of demand matrix

The results, presented in Figure 5.1, Figure 5.2, Figure 5.3, and Figure 5.4, 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.

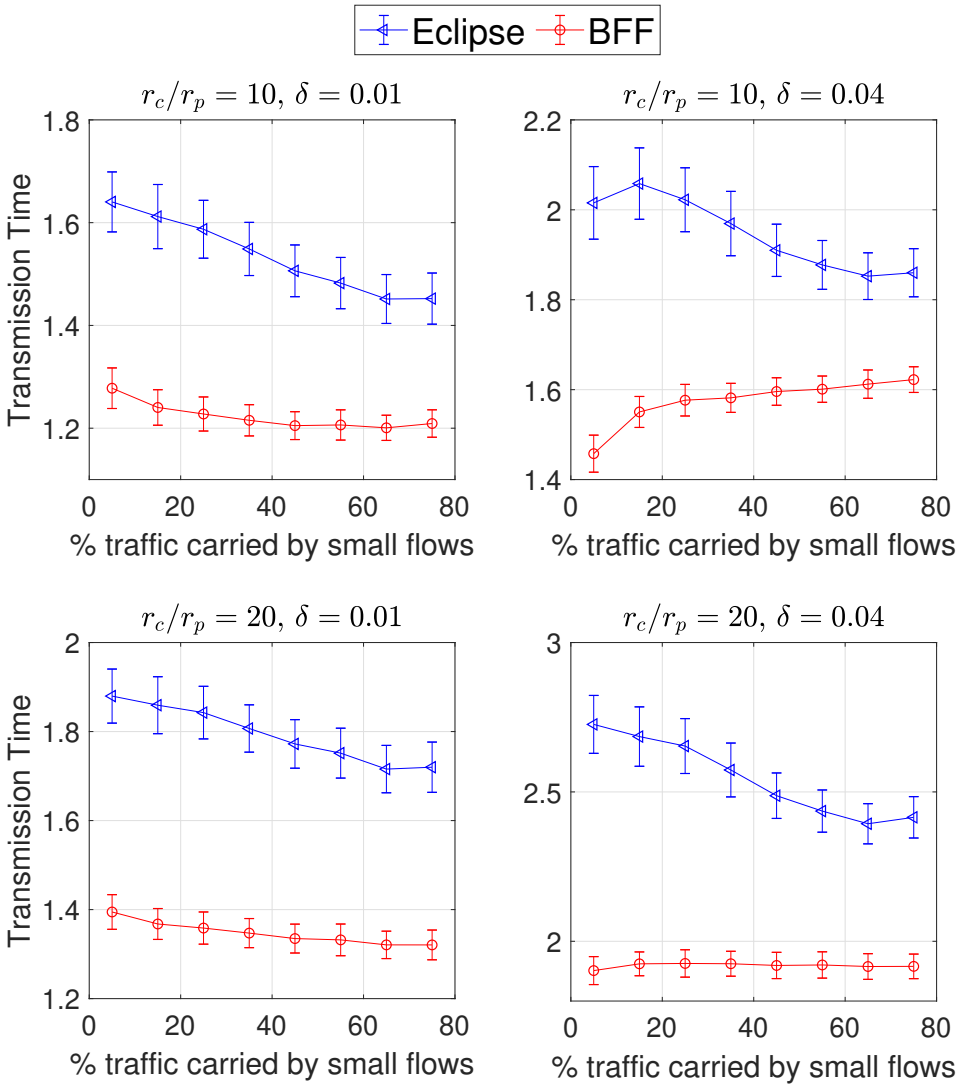

Figure 5.4: Performance comparison while varying skewness of demand matrix

5.6.4 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 5.2.

Table 5.2: 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 5.2, 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 [61] was made by the first author of [2] (the Eclipse/Eclipse++ paper).

CHAPTER 6

SUMMARY OF THE THREE ALGORITHMS

In the previous three sections, we have shown that each of the proposed three algorithms (2-hop Eclipse, QBvND, and BFF) outperforms the state-of-the-art solution Eclipse [2] on one or several metrics (i.e., transmission time performance, execution time performance, etc). However, their respective advantages over Eclipse are quite different from each other, since the designs of them are focused on different metrics improvement. In this section, we summarize their respective advantages of the three algorithms, by comparing the following four metrics of them:

- Applicable conditions;
- Computational complexity;
- Theoretical guarantee on transmission time performance;
- Empirical transmission time performance;

6.1 Applicable condition, computational complexity, and theoretical guarantee

The first three metrics can be compared straightforwardly and we summarize the results in Table 6.1.

Applicable condition. Only BFF requires the optical switch to be partially reconfigurable, whereas 2-hop Eclipse and QBvND apply for all types of optical switches.

Computational complexity. The respective computational complexities of 2-hop Eclipse, QBvND, and BFF are shown in Table 6.1. Here B is the largest entry in the demand matrix, which can be considered as $O(1)$, n is the number of racks (scale of the demand matrix), and K is the number of average number of reconfigurations per port (rack), which

Table 6.1: Applicable condition, computational complexity, and theoretical guarantee comparisons of 2-hop Eclipse, QBvND, and BFF

Algorithm	2-hop Eclipse	QBvND	BFF
Applicable condition	$\emptyset$	$\emptyset$	Partially Reconfigurable
Computational Complexity	$O(Kn^{5/2}\log n \log B + \min(K,n)Kn^2)$	$O(n^{3.5})$	$O(Kn^2+n^{5/2}\log B)$
Theoretical guarantee on transmission time performance	$\emptyset$	$(1+\sqrt{n\delta})^2$	$2 \times OPT$

is typically on the order of $O(n)$. Hence, the computational complexities of 2-hop Eclipse, QBvND, and BFF can be viewed as $O(n^4)$, $O(n^{3.5})$, and $O(n^3)$ respectively. Obviously, BFF has the lowest computational complexity and that of 2-hop Eclipse is the highest.

Theoretical guarantee on transmission time performance. Eclipse [2] considers a different objective, that is, to maximize the throughput given a fixed period of time. This objective, as we mentioned earlier, is roughly the dual of our objective (minimize the total transmission time). Eclipse [2] exploited the *submodularity* [71] structure of their optimization problem and proved a theoretical guarantee that is $(1 - 1/e) \approx 63.2\%$ times the theoretically optimal (maximum) throughput. This property, however, does not apply to our 2-hop Eclipse algorithm. QBvND has a theoretical guarantee that the total transmission time is upper bounded by $(1 + \sqrt{n\delta})^2$, where both the maximum row / column sum of the traffic demand matrix and the per-port transmission rate of the optical switch are normalized to 1. This upper bound applies for both the hybrid switching problem and the standalone optical switching problem. BFF also has a theoretical guarantee of $2 \times OPT$, 2 times of the optimal transmission time, which is derived from the property of LIST scheduling [72].

In the next section, we will compare the transmission time (throughput) performances of 2-hop Eclipse, QBvND, and BFF, in hybrid switching under various system settings and various traffic demand matrices.

6.2 Transmission Time Performance Comparison

6.2.1 Using constructed traffic demand matrices

For our simulations, we use the same traffic demand matrix D as used in [1, 16]. 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 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

$$D = (\sum_{i=1}^{n_L} \frac{c_L}{n_L} P_i + \sum_{i=1}^{n_S} \frac{c_S}{n_S} P_i' + \mathcal{N}_1) \times 90\% + \mathcal{N}_2 \tag{6.1}$$

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

Recall that the parameters c_L and c_S control the skewness (few large elements in a row or column account for the majority of the row or column sum) of the traffic demand. 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. Same as in [1, 16], two noise matrix terms $\mathcal{N}_1$ and $\mathcal{N}_2$ are added 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. Each nonzero (noise) element here in $\mathcal{N}_1$ has a standard deviation of $1/5$ of the value of the demand matrix element it is to be added to. We also add a positive Gaussian noise – with standard deviation equal to 0.3% of 1 – to 50% of the zero entries of the demand matrix.

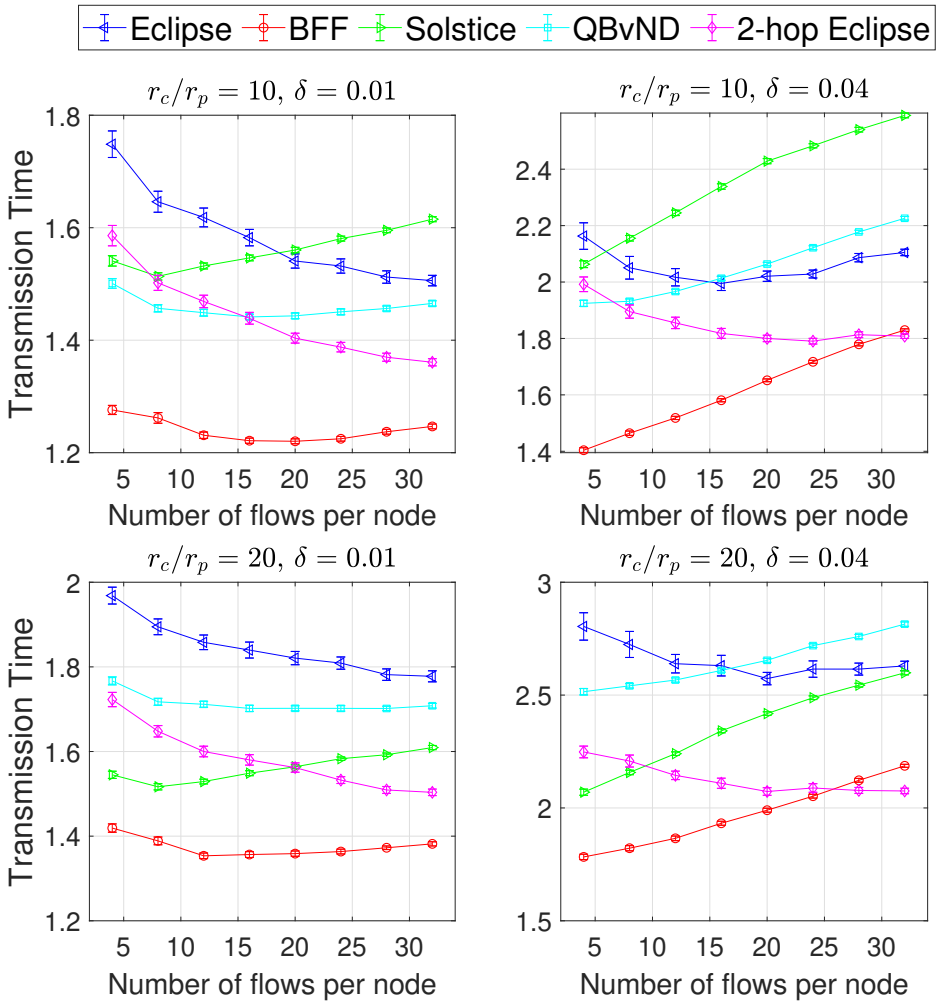

Figure 6.1: Performance comparison while varying sparsity of demand matrix

Transmission Time Performances

In this section, we evaluate the performance robustness of the three algorithms, 2-hop Eclipse, QBvND, and BFF, 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) $\delta = 0.04, r_c/r_p = 20$.

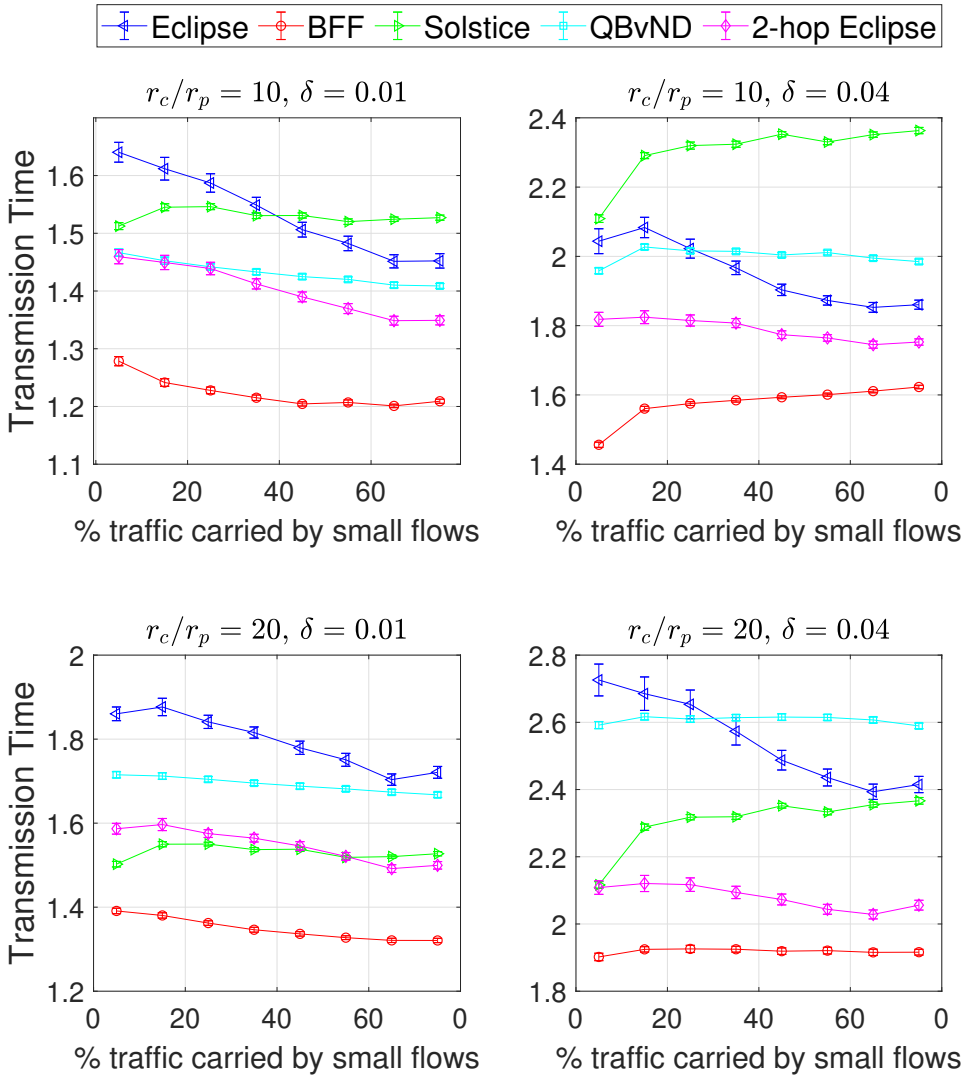

Figure 6.2: Performance comparison while varying skewness of demand matrix

Recall that δ is the reconfiguration delay of the optical switch, r_c $(= 1)$ and r_p are the transmission rate of the optical switch and the packet switch respectively. Figure 6.1 compares the transmission time of the three algorithms with Solstice 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 6.2 compares the transmission time of the three algorithms with Solstice 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 $(= 4 + 12)$.

Both Figure 6.1 and Figure 6.2 show that BFF performs the best (i.e., shortest transmission times) in almost all the circumstances among all these algorithms. This is not surprising: Recall that BFF applies for optical switches that are partially reconfigurable. As we described in section 5.2, this partial reconfigurability prevents unnecessary reconfigurations and therefore increases the throughput of the optical switches. Comprehensively speaking, 2-hop Eclipse performs the second best among them. It even outperforms BFF in the right-side two figures of Figure 6.1 when the number of flows per node is large (i.e., when the traffic matrix is dense). This result is also reasonable: 2-hop Eclipse is an indirect routing algorithm, which can serve more VOQs (i.e., more entries in the demand matrix) under a fixed number of configurations. In other words, when the traffic matrix is dense (i.e., has large number of nonzero entries), 2-hop Eclipse is able to use fewer number of configurations to "cover" more nonzero entries and therefore reduces reconfiguration times. When the reconfiguration delay δ is large, this smaller number of reconfiguration times induces significant savings on the reconfiguration overhead. QBvND, on the other hand, does not perform very well comparing with Eclipse and Solstice. It is mainly because the design of QBvND is targeting the standalone optical switching problem, in which all traffic demands are transmitted by the optical switch. The advantage of QBvND is at "swiping" out the small entries using only a few configurations, which does not help much on the hybrid switching problem.

6.2.2 Using recovered traffic demand matrices from real traces

For our simulations, we also use traffic demand matrices that are "recovered" (generated) from real traces.

Dataset

We use the open-source traces provided by Facebook that describe Facebook's data center traffic [73]. The traces contain three different data center clusters that represent different application types: Database, Web servers, and Hadoop servers. More than 300 million packets are sampled from each of the three cluster traces. We note that the dataset is highly sampled from the outbound link at a rate of $1 : 30,000$.

Inter-rack intra-pod traffic demand

Each sampled packet in the traces contains several attributes. Among them, we only concern the following four attributes:

- Timestamp;
- Anonymized src/dst rack;
- Anonymized src/dst pod;
- Packet length

Here "pod" is a standard "unit of network" in Facebook data center fabric, which consists of dozens of top of rack (TOR) switches and a few fabric switches to connect them. Using the four attributes above, we can obtain all the inter-rack traffic in a pod.

Traffic recovery process

Recall that the dataset is highly sampled from the outbound link at a rate of $1 : 30,000$. In other words, the traces only provide a tiny portion of traffic demand. Hence, we cannot obtain traffic demand matrices (say within a $3ms$ interval) straightforwardly using the traces. It naturally arises the following question: How to generate traffic matrices using the traces? A. Chen *et al* [74] provides a traffic recovery method that is implemented on the same trace. Figure 6.3 shows how it works.

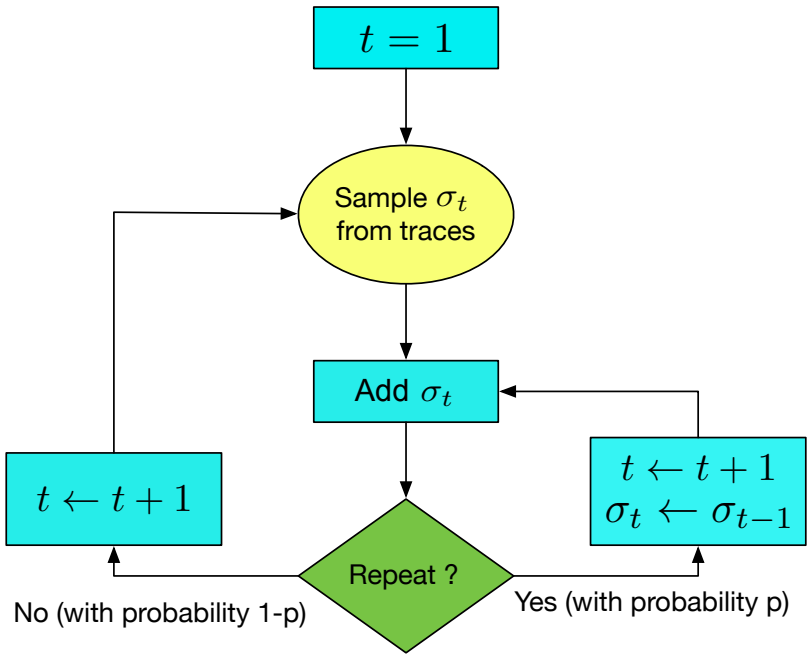

Figure 6.3: Traffic Generation Model

The traffic matrix is generated by iteratively sampling packets from the trace. Once a packet is sampled from the trace, it will be selected again in the next iteration with a *repeating probability* p, the same packet is sampled again; otherwise, a new packet is sampled from the trace. Here p is the repeating probability of a packet. It determines how bursty the generated traffic demand is. The larger p we use, more busty the traffic demand we generate. Note that the number of repeating times of each packet follows the geometry distribution with probability p. Each packet should be repeatedly sampled $1/(1-p)$ (the expectation of the geometry distribution) times on average.

To generate traffic matrices using the above method, we have two parameters to determine: (1) The repeating probability p; and (2) The total number of packets to sample for each demand matrix. Normally, we should use derived statistics from the trace to determine the two parameters. However, for the repeating probability p, we cannot obtain any useful statistics from the trace, since it is highly sampled. For this reason, we try to cover the range of all meaningful p values by selecting various values of p for matrix genera-

tion. More precisely, we select p from the following four values $\{0, 0.8, 0.9, 0.95\}$, which corresponds to an average repeating times (i.e., $1/(1-p)$) of $\{1, 5, 10, 20\}$ respectively.

On the other hand, since the optical switches have much larger bandwidth than packet switches, the traffic demand within a small time interval (say $3ms$) that is currectly being handled by a packet switch is too "easy" for an optical switch to handle. For instance, for the data center cluster used for database, the average number of sampled packets per second is 14.7, which induces the average number of total packets per second to be $14.7 \times 30,000 = 441000$ and the average number of total packets per $3ms$ to be $441000 \times 0.003 = 1323$. If we generate a traffic matrix that accounts for only 1323 (or several thousands) sampled packets, the traffic demand would be too small for an optical switch to handle (a packet switch with much lower bandwidth is sufficient to handle the demand). In other words, the resulting transmission time would be only a few dozens of microseconds, under a typical optical switch configuration, which is much shorter than the length of the time interval ($3ms$). For this reason, we scale the number of sampled packets per traffic demand matrix to be a large number (say tens of thousands), so that the traffic demand is sufficiently large, and it is worthy to apply an optical switch to transmit it.

Optical switch configuration

For our simulation, we borrow the configuration of the optical switch proposed in [75]:

- Reconfiguration delay δ: $11.5\mu s$;
- Transmission rate per port r_c: $10Gbps$;

Note that each pod contains roughly 150 TOR switches, so the selected optical switch must supports multiple input / output ports. We choose this particular optical switch, since to the best of our knowledge, it has the smallest reconfiguration delay (in microseconds level) among the optical switches that can support multiple (the prototype proposed in [75] supports 24 ports) input / output ports. Similar to the previous works on hybrid switching,

we assume a packet switch that has a transmission rate $r_p = 0.1 \times r_c$ ($1/10$ transmission rate of the optical switch) is combined with this optical switch to schedule each traffic demand matrix.

Transmission Time Performances

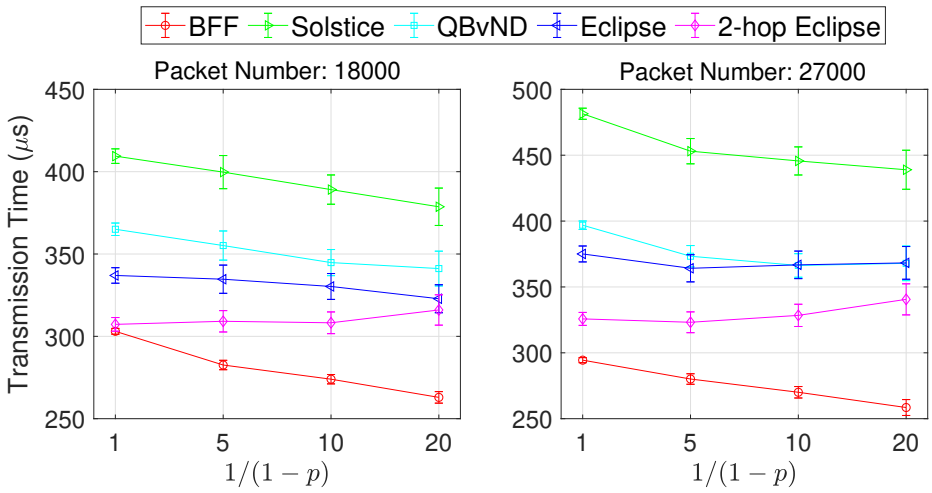

Figure 6.4: Transmission time performances under generated traffic matrices

Figure 6.4 shows the transmission time performances under generated traffic matrices. Similar to the previous results (of using constructed traffic matrices), BFF performs the best (i.e., shortest transmission times) in all the circumstances among all these algorithms, 2-hop Eclipse performs the second best, and QBvND does not perform very well comparing with Eclipse and Solstice. The underlining reason behind this result has been thoroughly discuss earlier in subsection 6.2.1.

Recall that p, the repeating probability of packet sampling process, controls "how bursty the traffic demand is". When p is larger, each sampled packet will be reused for more times, which induces a more bursty traffic pattern. Note that the burstiness implies the skewness of the demand matrix. When the traffic pattern is bursty, a few large flows would account for the majority of the traffic demand of the corresponding input port and output port. In the context of the traffic matrix, the large flows are mapped to a fewer number of large elements

in the traffic matrix, each of them accounts for the a large portion of the row or column sum. Therefore, the resulting demand matrix under larger p (i.e., large $1/(1-p)$) would be more skewed. The burstiness also implies the sparsity of the demand matrix. When the traffic pattern is bursty, the sampled packets would be concentrated in a few flows, each of which contributes at most 1 nonzero entry in the demand matrix. Therefore, the resulting demand matrix under larger p (i.e., large $1/(1-p)$) would also be more sparse.

Recall that the two characteristics, skewness (few large elements in a row or column account for the majority of the row or column sum) and sparsity (the vast majority of the demand matrix elements have value 0 or close to 0), are favorable for scheduling algorithms to perform well. Figure 6.4 shows that the transmission time decreases as $1/(1-p)$ increases (i.e., p increases) for almost all algorithms, except for 2-hop Eclipse. The performance of 2-hop Eclipse is however counterintuitive: its transmission time slightly increases as $1/(1-p)$ increases.

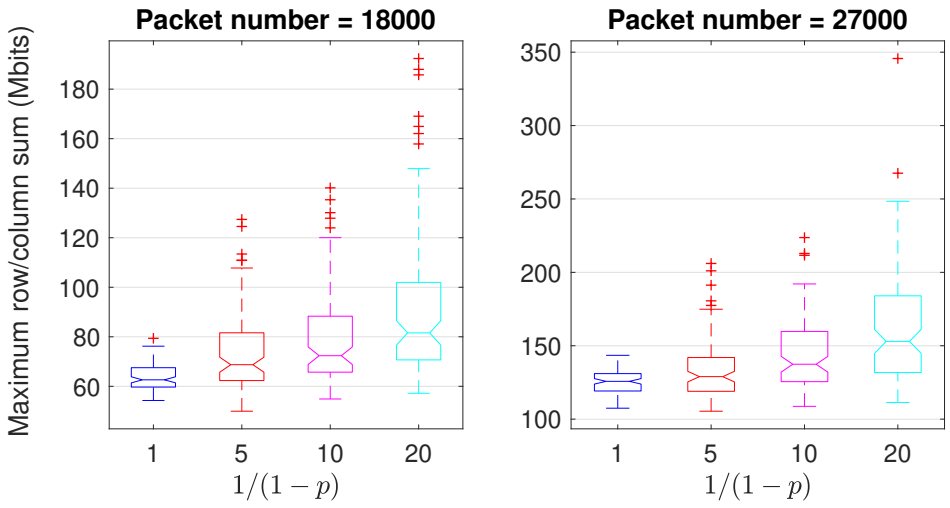

Figure 6.5: Maximum row/column sums of the generated traffic matrices

To explain this counterintuitive phenomenon, we further investigate the effect of burstiness: We note that burstiness also incurs another "skewness" on the traffic demands among the input (output) ports. More specifically, due to the burstiness, some input (output) port

may have much larger traffic demands than the other ports. In fact, it would cause the transmission time to be longer, since the "heaviest" (i.e., having the largest amount of traffic demand) input (output) port itself requires a long transmission time. To confirm our hypothesis, we compute the maximum row and column sum of each generated traffic matrix, which corresponds to the amount of traffic demand of the "heaviest" input (output) ports. The results are shown in Figure 6.5. It shows that amount indeed increases as $1/(1-p)$ increases (i.e., p increases), which confirms our hypothesis.

6.3 From batch scheduling algorithm to batch scheduling process

Almost all previous works on hybrid switch scheduling, including ours, perform only *a "snapshot" of batch scheduling*, that is, given the traffic demand matrix D, a schedule of the optical and the packet switches is computed only for transmitting traffic in D. In reality, a scheduling algorithm needs to handle traffic demands that are continuously arriving. In this section, we will elaborate how to extend a batch scheduling algorithm to a batch scheduling process.

6.3.1 Batch Scheduling Process

Let us introduce a naive batch scheduling process using an arbitrary batch scheduling algorithm. Let $\{[t_k, t_{k+1}]\}_{k=0}^{\infty}$ denotes the sequence of scheduling epochs. At time t_1, the traffic demand accumulated (within $[t_0, t_1]$), denoted as D_1, is measured (by traffic demand collector) and fed into the hybrid switch scheduler as the input. The scheduler then takes a scheduling epoch $[t_1, t_2]$ to compute the batch schedule of D_1 (using the batch scheduling algorithm). This schedule will be operated within epoch $[t_2, t_3]$. Similarly, the schedule of the traffic demand accumulated at time t_2, denoted as D_2, is computed in $[t_2, t_3]$ and then operated within $[t_3, t_4]$. Generally speaking, the schedule of the traffic demand accumulated at time t_k, denoted as D_k, is computed in $[t_k, t_{k+1}]$ and then operated within $[t_{k+1}, t_{k+2}]$.

This naive batch scheduling process has two flaws: (1) The computation of a schedule

that is going to be operated within $[t_{k+1}, t_{k+2}]$ only considers the traffic demand that is accumulated before time t_k. The new arrival traffic demand within $[t_k, t_{k+1}]$ is completely dismissed; (2) Normally, the computed schedule only occupies a portion of the scheduling epoch, which leaves the rest of the epoch underutilized. Nevertheless, in the next section, we will propose a solution that kills both birds (flaws) with one stone. The general idea is very simple: We "expand" the computed schedule to use the whole scheduling epoch. The "expansion" of the computed schedule, in turn, generates "slacks" (i.e., residue capacity) that can carry new arrival traffic demand.

6.3.2 Optical Switch Schedule Scaling

Now we describe how to "expand" a schedule to make full use of a scheduling epoch in detail. We start with a simple case, where the optical switch is not partial reconfigurable. As we described before, a schedule S of such an optical switch consists of a sequence of optical switch configurations and their durations: (M_1, α_1), (M_2, α_2), $\cdots$, (M_K, α_K). Each M_k is an $n \times n$ permutation (matching) matrix that denotes the k^{th} switch configuration. $M_k(i, j) = 1$ if input i is connected to output j and $M_k(i, j) = 0$ otherwise. α_k denotes its duration. Recall that the optical switch takes a reconfiguration delay δ between any two sequential switch configurations, and it transmits no traffic during this period. The total transmission time of the above schedule is $T_S \triangleq K\delta + \sum_{k=1}^{K} \alpha_k$, where δ is the reconfiguration delay, K is the total number of configurations in the schedule. Let $T\ (> T_S)$ denotes the length of the scheduling epoch.

Our "expansion" algorithm scales up all of the durations $\alpha_1, \alpha_2, \cdots, \alpha_K$ by a universal factor $c\ (> 1)$, such that the scaled schedule, denoted as S', expressed as $(M_1, c\alpha_1)$, $(M_2, c\alpha_2)$, $\cdots$, $(M_K, c\alpha_K)$, makes full use of the scheduling epoch. Mathematically, we have

$$K\delta + c\sum_{k=1}^{K} \alpha_k = T \tag{6.2}$$

It is easy to derive that Equation 6.2 holds if and only if $c = (T - K\delta)/(T_S - K\delta)$, which would be used as the scaling factor.

This scaling algorithm has several merits: First, the scaling algorithm has an extremely low computational complexity. Once a schedule S (before scaling) is computed (say using QBvND), the scaling algorithm can scale S almost instantly and obtain the scaled schedule $\tilde{S}$; Second, note that the traffic demand patterns between two successive schedule epochs should have some similarities, since some data flows may exist during both scheduling epochs. As a result, the optical switch configurations used in the last scheduling epoch should be able to transmit a considerate portion of the traffic demands arrived in the current scheduling epoch. Hence, this scaling algorithm should be able to transmit a considerate portion of new arrival traffic demand; Third, this scaling algorithm does not incur additional reconfiguration cost (i.e., the total number of reconfigurations K does not increase). Note that the switching (reconfiguration) times is closely related to the lifetime of an optical switch [39]. This scaling algorithm minimizes the fatigue costs of the optical switch.

Scale the schedule of a partially reconfigurable optical switch

When the optical switch is partially reconfigurable, scaling a schedule S of it becomes more difficult, since the reconfigurations happening at different input (output) ports are not synchronized. Recall that for a partially reconfigurable optical switch, its schedule 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 output ports at time t.

For describing the scaling algorithm, we consider the schedule S in a different way: Note that the schedule of each input port (i.e., transmitter) can be viewed as a sequence of successive intervals. There are three types of intervals in this context: (1) transmission intervals; (2) reconfiguration intervals; and (3) idle intervals. Figure 6.6 shows an example of such a schedule

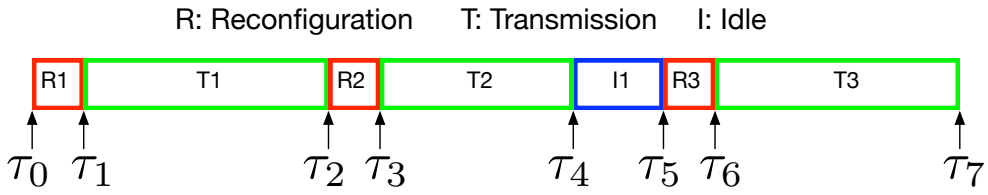

Figure 6.6: An example of the schedule of an input port (transmitter)

The schedule of an input port must satisfy the following properties: (1) The end time of a reconfiguration interval is the start time of a transmission interval (i.e., τ_1, τ_3, τ_6); (2) The end time of an idle interval is the start time of a reconfiguration interval (i.e., τ_5); and (3) The end time of a transmission interval could be either the start time of a reconfiguration interval (i.e., τ_2), or the start time of a idle interval (i.e., τ_4).

Now we describe the scaling algorithm using the above example. Given a scaling factor c (we will describe how to determine c shortly), we first scale all the timings $\tau_1, \tau_2, ..., \tau_7$ by multiplying c. Then all of the reconfiguration intervals, transmission intervals, and idle intervals are scaled c times respectively. Second, we split each scaled reconfiguration interval (with duration $c\delta$) into two parts, a normal reconfiguration interval (with duration δ) plus an idle interval (with duration $(c-1)\cdot\delta$).

The scaling factor c is assigned to be T/T_{max}, where T_{max} denotes the maximum finish time of all input ports (i.e., the finish time of the input port in the example is τ_7), and T is the length of the scheduling epoch. Note that a necessary condition of a schedule $S(t)$ to be *feasible* is that $S(t)$ is a sub-matching matrix at any time t (i.e., each input port connects to at most one output port at the same time, and each output port connects to at most one input port at the same time). It is easy to verify that the scaled schedule, denoted as $\tilde{S}(t)$, using this algorithm is feasible, since we have $\tilde{S}(t) = S(t/c)$ at any time t.

6.3.3 Evaluation Results

In this section, we evaluate the performance of the above scaling algorithm. More specifically, we measure the amount of additional traffic demand that can be transmitted under

various scaling factors. We use QBvND as the scheduling algorithm. Other scheduling algorithms (e.g., BFF) show similar results.

Demand matrix

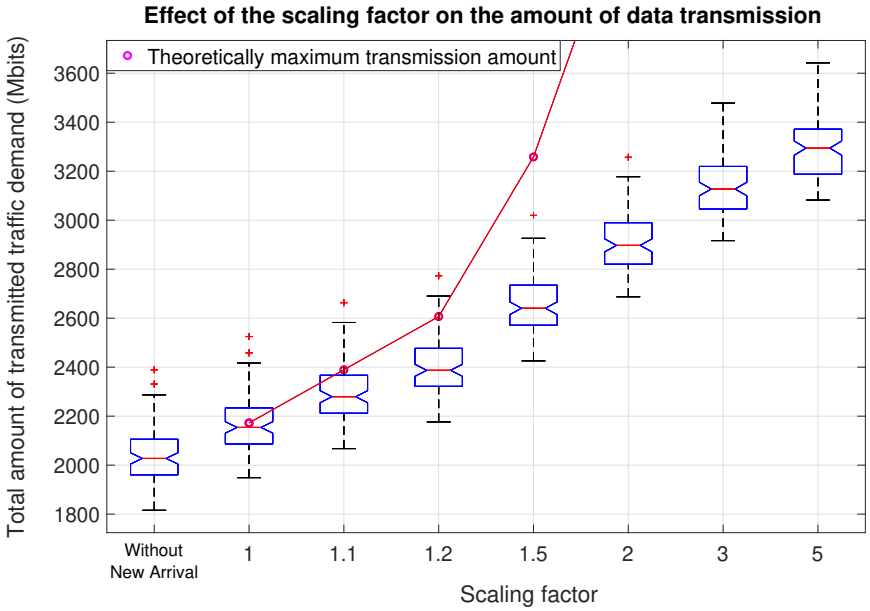

Figure 6.7: Effect of the scaling factor on the total amount of data transmission

We use the generated demand matrices used in subsection 6.2.2. We assume the generated demand matrices to be the arrival traffic demands for a sequence of successive scheduling epochs. Each traffic matrix is generated by sampling $27,000$ packets, and the repeating probability p is selected to be 0.95%. The generated traffic matrices correspond to the last column of the right-hand-side of Figure 6.4, where $1/(1-p) = 20$. Other generated traffic matrices using a different number of sampled packets or a different repeating probability show similar results. We continue to assume the reconfiguration delay of the optical switch is $11.5\mu s$, the optical switch has 156 input ports and 156 output ports respectively, and the transmission rate per port is $10Gbps$. We measure the total amount of transmitted traffic

demand under different scaling factors and concludes the results in Figure 6.7.

The first column of Figure 6.7 (i.e., without new arrival) is the amount of traffic transmission when new arrival traffic demand is not in consideration. The rest columns consider the new arrival traffic demand and use different scaling factors. It is not surprising that a larger scaling factor induces a larger amount of transmitted traffic demand, as shown in Figure 6.7. For instance, when the schedule is scaled by a scaling factor of 1.2, the amount of data transmission increases roughly 12% comparing to that of the original schedule (without scaling); when the schedule is scaled by a scaling factor of 1.5, the amount of data transmission increases roughly 24%.

The red line in Figure 6.7 represents the *theoretically maximum amount of data transmission after scaling*: Note that the duration of each configuration is scaled up by a factor of c. Theoretically, the amount of data transmission using the scaled schedule is at most c times of that using the original (not scaled) schedule. Figure 6.7 shows that when the scaling factor is small (i.e., $c \leq 1.2$), the amount of data transmission is close to the theoretical maximum. When the scaling factor is large (i.e., $c \geq 2$), that amount is far less than the theoretical maximum.

CHAPTER 7

LINE-EVEN SPARSE SPLIT (LESS): TRAFFIC DEMAND SPLIT IN PARALLEL-OPTICAL-SWITCHED NETWORKS

7.1 MSB Problem and LESS Solution

In this section, we first formally formulate the Matrix Split and Balance (MSB) problem in subsection 7.1.1. Then in subsection 7.1.2, we introduce our solution, Line-Even Sparse Split (LESS), and elaborate how to reduce LESS for a s-way MSB problem to LESS for a *2*-way MSB problems. Then we describe in details how LESS solves a *2*-way MSB problems by two different methods, a straightforward but slower LP-based method in subsection 7.1.3, and a faster combinatorial method in subsection 7.1.4.

7.1.1 Matrix Split and Balance (MSB)

In this section, we formally formulate the MSB problem of splitting the demand matrix D into s sub-workload matrices $D_1, D_2, ..., D_s$. It is a constrained optimization problem with the objective of minimizing the total number of nonzero entries (Formula Equation 7.1), or equivalently of maximizing the sparsity of the split. There are four sets of constraints shown in Formulae Equation 7.2 through Equation 7.5 respectively. In them we define $[s] \triangleq \{1, 2, \cdots, s\}$ and $[n] \triangleq \{1, 2, \cdots, n\}$.

The first set of constraints (Equations Equation 7.2) state that, for each column j in each matrix D_k, the sum of entries in the j^{th} column of D_k must be equal to $1/s$ of the sum of entries in the j^{th} column of D. These constraints ensure that the j^{th} output port of every switch is given the same amount of workload that is equal to $1/s$ of the traffic to be received by the rack j. The second set of constraints (Equations Equation 7.3) state the same for each row i in each matrix D_k. These constraints ensure that the i^{th} input port of every switch

is given the same amount of workload that is equal to $1/s$ of the traffic to be transmitted by rack i. These two sets of constraints correspond to the aforementioned "line-even" (i.e., identical row or column sum) requirement. The fourth (Inequalities Equation 7.5) and the third (Equations Equation 7.4) sets state respectively that $D_1, D_2, ..., D_s$ are *nonnegative* matrices and that their total is D.

$$\text{minimize} \quad \sum_{k=1}^{s} \|D_k\|_0 \tag{7.1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} D_k(i,j) = \frac{1}{s} \sum_{i=1}^{n} D(i,j), \forall j \in [n], k \in [s] \tag{7.2}$$

$$\sum_{j=1}^{n} D_k(i,j) = \frac{1}{s} \sum_{j=1}^{n} D(i,j), \forall i \in [n], k \in [s] \tag{7.3}$$

$$\sum_{k=1}^{s} D_k(i,j) = D(i,j), \forall i,j \in [n] \tag{7.4}$$

$$0 \leq D_k(i,j) \leq D(i,j), \forall i,j \in [n] \tag{7.5}$$

Although all the constraints are linear, the objective function $\sum_{k=1}^{s} \|D_k\|_0$ is not, so this constrained optimization problem is not a Linear Programming (LP) problem. In fact, it has been proved to be NP-hard [22], so only heuristic or approximate solutions to it exist that run in polynomial time. Our solution LESS is a $(1 + \frac{(2n-1)(s-1)}{m})$-approximation algorithm, where $m = \|D\|_0$ is the number of nonzero entries in D. In practice, it can be considered a $(1+\epsilon)$-approximation algorithm, since typically $n = o(m)$, s is a small constant (typically < 10), and n can grow to hundreds (of racks) in real-world datacenter networks.

7.1.2 Line-Even Sparse Split (LESS)

The design of LESS is based on the following insight: The linear constraints Equation 7.2 through Equation 7.5 define a polytope within which any point satisfies the line-even condition and any extreme point of this polytope corresponds to a fairly sparse split in the sense

of Equation 7.6, which we will prove shortly. Hence our LESS algorithm is simply to find an extreme point of this polytope. This can be done by replacing the nonlinear objective function in Equation 7.1 by a dummy linear objective function such as "Minimizing 0" in the constrained optimization problem above, and solving the resulting LP problem using an LP solver such as Gurobi [76].

$$\sum_{k=1}^{s} \|D_k\|_0 \leq \|D\|_0 + (s-1)(2n-1) \tag{7.6}$$

Reduction from s-way Split to 2-way Splits

Throughout this section, whenever we use the term *split*, we mean to split (the matrix) in the LESS manner. In other words, such a split always corresponds to an extreme point of the corresponding polytope. Now we show that, for any $s > 2$, we can reduce an s-way split (i.e., D into D_1, D_2, ..., D_s) to a "binary tree" of $s-1$ recursive 2-way splits. This reduction will not only significantly simplify our presentation of the resulting linear programming (LP) problem and solution, but also allow for the use of parallel processing (to be elaborated next) to speedup its computation. Intuitively, this reduction is straightforward when s is a power of 2. For example, when $s = 8$, D is split first into "two halves", then into "four quarters", and finally into eight sub-workload matrices D_1, D_2, ..., D_8, each of which accounts for exactly $1/8$ of the total workload contained in D. The corresponding "binary tree" is a complete binary tree of height 3 with a total of $s - 1 = 7$ internal nodes, each of which corresponds to a 2-way split.

We now explain how this reduction can be done when s is not a power of 2. Due to the recursive nature of the splits, we need only to explain what the very first 2-way split of a matrix D' should be, when D' needs to *eventually* be split into s' pieces. There are only two cases to consider. In the first case where s' is an even number, the 2-way split has weights $(1/2, 1/2)$ in the sense each row or column sum of D'_1 is equal to $1/2$ of the corresponding row or column sum of D'. Matrices D'_1 and D'_2 will then be split further

into $s'/2$ pieces each. In the second case when s' is an odd number, the 2-way spit has weights $(\frac{s'-1}{2s'}, \frac{s'+1}{2s'})$ in the sense each row or column sum of D'_1 is equal to $\frac{s'-1}{2s'}$ of the corresponding row or column sum of D'. Matrices D'_1 and D'_2 will then be split further into $(s'-1)/2$ and $(s'+1)/2$ pieces respectively. For example, when $s'=7$, the 2-way split has weights $(3/7, 4/7)$. The resulting D'_1 and D'_2 need to be split further into 3 and 4 pieces respectively.

Now that any s-way split can be reduced to $s-1$ 2-way splits, we will only describe how a 2-way split is performed in the sequel. Furthermore, we will only consider weights $(1/2, 1/2)$ because the 2-way LESS algorithm works with any (pair of) weights (as parameters) in the same manner. When describing the 2-way LESS algorithm with weights $(1/2, 1/2)$ in the next section, we will also prove that each 2-way split increases the number of nonzero entries $\sum_{k=1}^{s} \|D_k\|_0$ by at most $2n-1$. This implies that any s-way split increases $\sum_{k=1}^{s} \|D_k\|_0$ by at most $(s-1)(2n-1)$ (Inequality Equation 7.6), since it can be reduced to $s-1$ 2-way splits.

Parallelization

As explained earlier, this reduction from *s*-way split to *2*-way splits allows for the speedup of its computation using parallelization. We now illustrate how to parallelize the computation in the aforementioned simple case of $s=8$, where there are seven instances of 2-way split computations over three rounds: split D first into "two halves" (one instance) in the first round , then into "four quarters" (two instances) in the second round, and finally into eight sub-workload matrices D_1, D_2, ..., D_8 (four instances) in the third round. Clearly, four (more generally $s/2$) parallel processors (or cores) can compute this 8-way split in three rounds of time, that is, roughly three (more generally $\log_2 s$) times the amount of time needed to compute a 2-way split instance. In comparison, serial execution takes roughly seven (more generally $s-1$) rounds of time. Finally, we do not advocate further pipelining these three (more generally $\log_2 s$) rounds of computations because although it

increases the "throughput" of this computation, it does not reduce the "delay", which is what matters in real-world operations.

Comparison with Naive Solution

Although the naive MSB solution of splitting D evenly (i.e., $D_1 = D_2 = \cdots = D_s = D/s$) satisfies all the constraints Equation 7.2 through Equation 7.5, it maximizes, rather than minimizes, the objective function $\sum_{k=1}^{s} \|D_k\|_0$. This leads to much higher reconfiguration costs for the naive solution, as we will show in section 7.2. As a result, LESS outperforms the naive solutions under most of the realistic parameter settings.

7.1.3 LP-based 2-way LESS

In this section, we describe the 2-way split of a matrix D' into two matrices D'_1 and D'_2 with weights $(1/2, 1/2)$. For convenience of presentation, we drop the apostrophe character from D', D'_1, and D'_2 and write them as D, D_1, and D_2 respectively. We emphasize this (new) D could be the original demand matrix or any internal node of the aforementioned "binary tree" of 2-way splits.

This 2-way split corresponds to finding an extreme point of the polytope defined by the following equations and inequalities using the LESS algorithm. Here Equation 7.7, Equation 7.8, and Equation 7.9 correspond to the special case of Equation 7.2, Equation 7.3, Equation 7.4, and Equation 7.5 when s is set to 2. And the weight for D_1 is the term $\frac{1}{2}$ in Equation 7.7 and Equation 7.8. This term will have a different value if D_1 has a different weight (e.g., $3/7$ in the "odd split" example in subsubsection 7.1.2). Note that we only need to compute D_1, since $D_2 = D - D_1$.

$$\sum_{j=1}^{n} D_1(i,j) = \frac{1}{2}\sum_{j=1}^{n} D(i,j), \forall i \in [n] \tag{7.7}$$

$$\sum_{i=1}^{n} D_1(i,j) = \frac{1}{2}\sum_{i=1}^{n} D(i,j), \forall j \in [n] \tag{7.8}$$

$$0 \leq D_1(i,j) \leq D(i,j), \forall i,j \in [n] \tag{7.9}$$

Note that Equation 7.7 corresponds to n equations (also called *tight constraints* below in Lemma 1), one for each row i, and so does Equation 7.8. Out of these $2n$ tight constraints, only $2n-1$ of them are linearly independent, because the sum of n row sums of D_1 has to be equal to the sum of n column sums of D_1. According to Lemma 1 below, any *extreme point solution* (defined precisely below in Definition 1) of this LP problem has at most $2n-1$ *variables*. In this context, a *variable* corresponds to a matrix entry $D_1(i,j)$ that is not on the boundary of Equation 7.9, or in other words $0 < D_1(i,j) < D(i,j)$. Clearly, each such variable $D_1(i,j)$ increases the number of nonzero entries from one (namely $D(i,j)$) before the split to two (namely $D_1(i,j)$ and $D_2(i,j)$) after the split. This proves the following proposition.

Proposition 1. *A 2-way split of D under constraints Equation 7.7 through Equation 7.9 increases the total number of nonzero entries by at most $2n-1$.*

Lemma 1 (**Rank Lemma**, Lemma 1.2.3 in [77])**.** *Let $P = \{x : Ax \geq b, x \geq 0\}$, and let x be an extreme point solution of P such that $x_i > 0$ for each i. Then any maximal number of linearly independent tight constraints of the form $A_i x = b_i$ for some row i of A equals the number of variables.*

Definition 1 (Definition 1.2.1 in [77])**.** *Let $P = \{x : Ax \geq b, x \geq 0\} \subseteq \mathbb{R}^n$. Then $x \in P$ is an extreme point solution of P if there does not exist a nonzero vector $y \in \mathbb{R}^n$ such that $x + y, x - y \in P$.*

Such an extreme point solution can be computed using a LP solver such as Gurobi [76]. However, when n is large, this LP computation is very slow. For example, when $n = 100$ (racks), it takes the Gurobi, which is the quickest among LP solvers by our experience, hundreds of milliseconds to compute a 2-way split of D. In the next section, we describe a non-LP-based LESS algorithm that performs the same LP computation, but in a combinatorial manner. For $n = 100$, it runs an order of magnitude faster than LP-based LESS, as we will show in subsection 7.2.5.

Algorithm 6: The pseudocode of combinatorial 2-way LESS

Input : D;
Output: D_1;
Initialize $D_1(i, j) \leftarrow D(i, j)/2, \forall i, j \in [n]$;
Convert D_1 to G;
while *An alternating cycle σ is found in G* **do**
    Increase and decrease the weights of edges in σ in an alternating manner by the same value η so that all edge weights remain within their "legal ranges" and one or more edges become tight;
    Remove tight edges from G;
end
Return D_1 that is converted back from G;

7.1.4 Combinatorial 2-way LESS

The combinatorial LESS algorithm performs the same LP (solving) operation as before: Starting with the aforementioned naive solution of $D_1 = D/2$, the algorithm iteratively modifies D_1 within the solution space to push it towards one of its extremal points. However, it does so by modeling D_1 as a bipartite graph and converting this LP (solving) operation into a graph computation problem.

Conversion to Graph Computation

To describe this conversion, we need the following definition.

Definition 2. *We call a matrix entry $D_1(i, j)$ tight if $D_1(i, j) = 0$ or $D_1(i, j) = D(i, j)$. In*

other words, $D_1(i,j)$ is tight if it is on the boundary of the constraint $0 \leq D_1(i,j) \leq D(i,j)$ (as a part of Equation 7.9). We call $D_1(i,j)$ loose otherwise (i.e., when $0 < D_1(i,j) < D(i,j)$).

In the combinatorial LESS algorithm, the matrix D_1 is modeled as a bipartite graph $G(U \cup V, E)$ whose edge set E evolves when the values of its entries are changed by the execution of the algorithm. In this bipartite graph, one partite (vertex set) U contains n vertices $u_1, u_2, \cdots, u_n$, in which each u_i, $1 \leq i \leq n$, corresponds to row i of D_1. The other partite V also contains n vertices $v_1, v_2, \cdots, v_n$ in which each v_j, $1 \leq j \leq n$, corresponds to column j of D_1. A weighted edge exists between u_i and v_j, or in other words $(u_i, v_j) \in E$, if and only if $D_1(i,j)$ is loose. The weight of this edge is set to $D_1(i,j)$.

Pseudocode of Combinatorial 2-way LESS

The pseudocode of the combinatorial (graph) algorithm is shown in algorithm 6. The design of the algorithm is based on the following fact: If the bipartite graph G contains a cycle σ (line 3), then we can modify the weights of the matrix entries (of D_1) that correspond to the edges in σ so that one or more such matrix entries become tight (line 4). Once such a matrix entry becomes tight, its corresponding edge is removed from the bipartite graph (line 5), according to the definition of the edge set E above. In line 3 of algorithm 6, the *depth-first search (DFS)* procedure is used to find a cycle.

algorithm 6 terminates only when no cycle exists in the bipartite graph. The resulting cycle-free graph, which has only $2n$ vertices, can have no more than $2n - 1$ edges (loose entries), since otherwise it cannot be cycle-free. Since each loose entry increases the number of nonzero entries by 1 as explained earlier, each 2-way split increases this number by at most $2n - 1$. Hence this combinatorial view offers another proof of Proposition 1.

Now we explain why and how we can make one or more edges (rather the corresponding matrix entries) tight in each such cycle σ, as stated in line 4 of algorithm 6. Since G is

bipartite, σ must contain an even number of edges. Like in the graph algorithm literature, we call σ an *alternating cycle* for a similar reason: "Walking around" the cycle starting at an arbitrary vertex on the σ and following a direction chosen arbitrarily (from the two possible directions), we will increase or decrease the weights of the edges traversed by the walk in an alternating manner, by the same amount η. In other words, we will increase the weight of the first edge by η, decrease the weight of the second edge by η, increase the weight of the third edge by η, and so on. As we will explain shortly using a toy example, this amount η is decided such that after the weight modifications, the weights of all edges (matrix entries) in σ remain within their "legal ranges" (i.e., $0 \leq D_1(i,j) \leq D(i,j)$ for any $D_1(i,j)$ in σ), and at least one of them becomes tight.

Since the starting point and the direction of each such walk are chosen arbitrarily, which can be implemented as being chosen randomly in practice, it appears that algorithm 6 is "equal-opportunity" in the sense its logic is inherently not biased for or against D_1 (equivalently against or for D_2) in distributing the up to $2n-1$ new nonzero entries to D_1 and D_2. This behavior explains why the resulting s sub-workload matrices from an s-way split have similar sparsities, as will be shown in subsection 7.2.4.

A Toy Example

We now illustrate the concept of alternating cycle and the process of weight modification by an example shown in Figure 7.1 and Figure 7.2. The first 4×4 matrix to the left in Figure 7.1 is the demand matrix D, which has 9 nonzero entries. As shown in Figure 7.1, D_1 is initialized to $\frac{D}{2}$ (second 4×4 matrix to the left), so it also has 9 nonzero entries. All of them are loose to start with so they are all underlined. The bipartite graph corresponding to D_1 at this moment is shown in Figure 7.2. As explained earlier, vertices u_1, u_2, u_3, and u_4 correspond to rows 1, 2, 3, and 4 of D_1 respectively and vertices v_1, v_2, v_3, and v_4 correspond to columns 1, 2, 3, and 4 of D_1 respectively. The graph G has 9 edges, corresponding respectively to the 9 loose entries of D_1. For example, the edge $u_1 \rightarrow v_2$

corresponds to the loose (underlined) matrix entry $D_1(1,2)$.

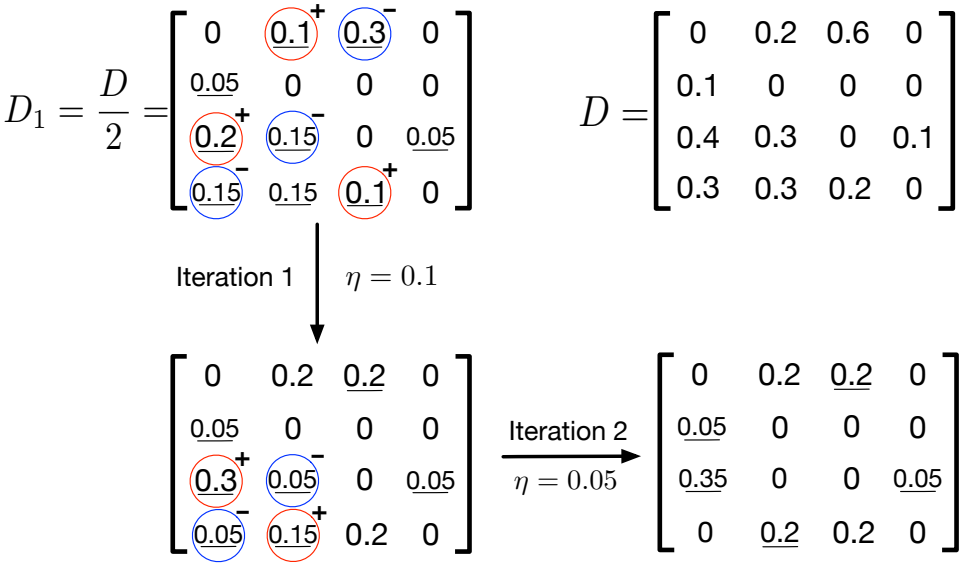

Figure 7.1: Example of cycle cancellation

In the first iteration, the alternating cycle $u_1 \to v_2 \to u_3 \to v_1 \to u_4 \to v_3 \to u_1$, highlighted in Figure 7.2 in alternating red and blue colors, is discovered. We start the cycle traversal at u_1. As specified in line 4, we increase $D_1(1,2)$, $D_1(3,1)$, and $D_1(4,3)$, which correspond to the red edges in Figure 7.2 and are hence circled in red with a superscript '+' in Figure 7.1, and decrease $D_1(3,2)$, $D_1(4,1)$, $D_1(1,3)$, which correspond to the blue edges in Figure 7.2 and are hence circled in blue with a superscript '-' in Figure 7.1, all by the same value η.

This alternating increase and decrease by the same value has a desirable property: Any row and column sum of D_1 remains the same after the weight modifications, because an increase to a matrix entry in any row or column is always accompanied by a decrease to another entry in the same row or column, and vice versa. For example, an increase to $D_1(1,2)$ is accompanied by a decrease to $D_1(1,3)$. Due to this desirable property, the final D_1 output by algorithm 6 satisfies Equation 7.7 and Equation 7.8.

We now explain how this η is determined using this example. The three matrix entries

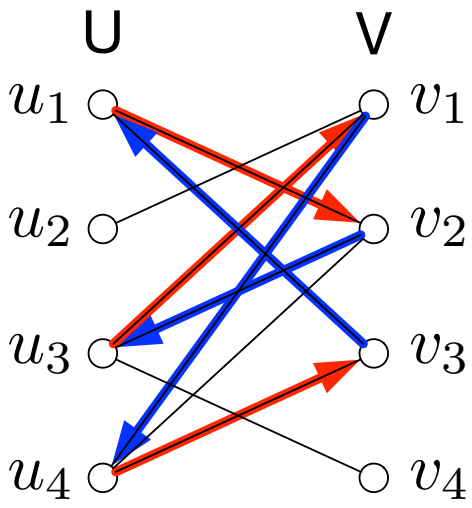

Figure 7.2: Alternating cycle mapping

to be increased and their current values are $D_1(1,2) = 0.1$, $D_1(3,1) = 0.2$, and $D_1(4,3) = 0.1$ respectively. Their respective upper bounds are $D(1,2) = 0.2$, $D(3,1) = 0.4$, and $D(4,3) = 0.2$. The three respective differences are $D(1,2) - D_1(1,2) = 0.1$, $D(3,1) - D_1(3,1) = 0.2$, and $D(4,3) - D_1(4,3) = 0.1$. So η cannot exceed 0.1, the minimum of the three. Similarly, η cannot exceed 0.15, since the three matrix entries to be decreased are $D_1(3,2) = 0.15$, $D_1(4,1) = 0.15$, and $D_1(1,3) = 0.3$ and their lower bounds are all 0. Then η is set to the minimum of these two upper bounds, which in this case is $min\{0.1, 0.15\} = 0.1$. After these increases and decreases by $\eta = 0.1$, the new values of these matrix entries are $D_1(1,2) = 0.2$, $D_1(3,1) = 0.3$, $D_1(4,3) = 0.2$, $D_1(3,2) = 0.05$, $D_1(4,1) = 0.05$, and $D_1(1,3) = 0.2$. Among them, the values of $D_1(1,2)$ and $D_1(4,3)$ have reached their respective upper bounds (both due to an increase) $D(1,2)$ and $D(4,3)$, so both of them become tight. Hence the two corresponding edges are removed from G (so no longer underlined in the third 4×4 matrix to the left in Figure 7.1) after the first iteration. In Figure 7.2, algorithm 6 stops after two iterations, when no more alternating cycle exists.

Computational Complexity of algorithm 6

In the following analysis, we assume the $n \times n$ matrix D is not extremely sparse in the sense $n = o(m)$ where $m = \|D\|_0$ is the number of nonzero entries in D. In this case, the "while" loop in algorithm 6 runs at most $m - (2n - 1) = O(m)$ iterations because there are m edges in G to start with, each iteration removes at least one edge from G, and algorithm 6 terminates when G contains no more than $2n - 1 = o(m)$ edges. Hence, the computational complexity of algorithm 6 is in theory $O(m^2)$ since, in each iteration, detecting a cycle using DFS has complexity $O(m)$, and so are computing η and updating edge weights. In practice, however, cycles are usually quite short for a real-world workload D, so empirically the complexity "feels more like" $O(m^{1.5})$ or less.

7.2 Evaluation

In this section, we evaluate the efficacy of LESS, and compare it with that of the naive algorithm (denoted in the figures and called "Naive" in the sequel) of simply dividing D by s. We do so by feeding the sub-workload matrices resulting from LESS and Naive respectively to the s optical switches, each of which is scheduled by BFF [16]. We denote these these two resulting schedulers as LESS+BFF and Naive+BFF respectively. In this comparison, we use the overall makespan, defined as the maximum among the makespans of the schedules of the s switches, as the performance metric.

7.2.1 Simulation Parameters and Setup

Traffic demand matrix D: It was shown in [9, 2] that typical traffic workloads in real-world data centers exhibit two characteristics: sparsity (the vast majority of the demand matrix elements have value 0 or close to 0) and skewness (few large elements in a row or column account for the majority of the row or column sum). Hence, for our simulations, we use the same set of sparse and skewed demand matrices as used in [9, 2]. In each

such matrix D, 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 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. Hence n_L and n_S control the sparsity, and c_L and c_S control the skewness, of the traffic demand, respectively. Roughly speaking, we have

$$D = \sum_{i=1}^{n_L} \frac{c_L}{n_L} P_i + \sum_{i=1}^{n_S} \frac{c_S}{n_S} P'_i + \mathcal{N} \tag{7.10}$$

where each P_i and each P'_i is an $n \times n$ random permutation matrix.

Same as in [9, 2], in our simulation studies, the default values of the sparsity parameters n_L and n_S are set to 4 and 12 respectively and the default values of c_L and c_S are set to 0.7 (*i.e.*, 70%) and 0.3 (*i.e.*, 30%) respectively. In other words, in each row (or column) of the demand matrix, by default the 4 large flows account for 70% of its total traffic demand, and the 12 medium flows account for the rest 30%. We will also vary the sparsity parameters n_L and n_S and skewness parameters c_L and c_S in our evaluations. In Equation (7.10), before a noise matrix $\mathcal{N}$ (described next) is added to it, each such D is doubly stochastic. As shown in Equation (7.10), we also add a noise matrix term $\mathcal{N}$ to D, like in [9, 2]. Each nonzero element in $\mathcal{N}$ is a Gaussian random variable that is added to a traffic demand matrix element that was nonzero before the noise added. Each nonzero (noise) element here in $\mathcal{N}$ has a standard deviation, which is equal to 0.3% of the normalized workload 1.

Reconfiguration delay of the optical switch δ: The larger δ is, the more time the optical switch has to spend on reconfigurations, and hence the higher the resulting makespan is. By default, $\delta = 0.04$ (i.e., 4% of the scheduling epoch), although we will vary δ in our simulation studies. Here we use a larger default value of δ than that in [9, 2], which is $\delta = 0.01$ (i.e., 1% of the scheduling epoch), because the former is closer to the δ values of real-world large (e.g., 100×100) optical switches that range mostly from hundreds of μs to milliseconds [78] (after being normalized by the typical epoch length of 3 milliseconds).

Although δ values as small as $12\mu s$ were mentioned in both [20] and [78], they apply only to a small switch (e.g., 4×4) or an optical transmitter-receiver pair with tiny rotation-angle distance.

Simulation Setup: In the rest of section 7.2, every point in every plot in every figure is the sample mean averaged from 100 simulation runs, so is every number in Table 7.1 and Table 7.2.

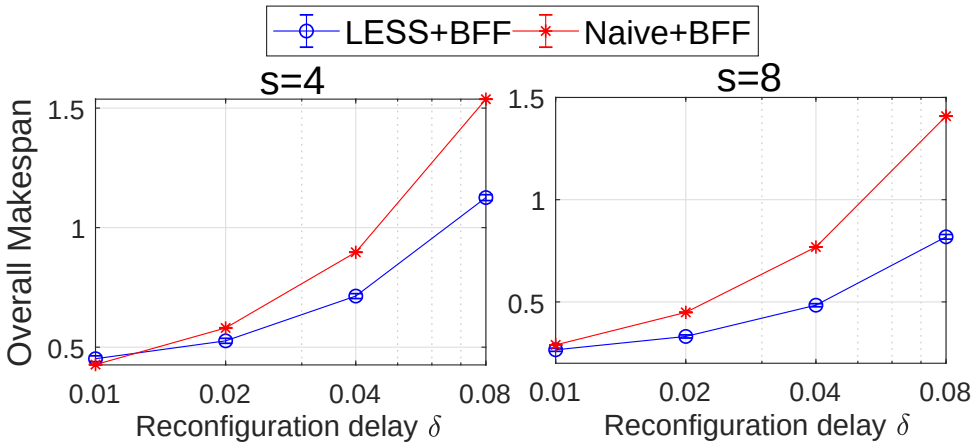

Figure 7.3: LESS+BFF vs. Naive+BFF while varying δ

7.2.2 Under Different System Parameters

In this section, we compare the overall makespan performances of LESS+BFF and Naive+BFF for different value combinations of s (number of parallel switches) and δ (reconfiguration delay), under traffic demand D with the default parameter settings described above (4 large flows and 12 small flows accounting for roughly 70% and 30% of the total traffic demand into each input port). The simulation results, presented in Figure 7.3, show that LESS+BFF outperforms Naive+BFF, as indicated by shorter overall makespans, when the reconfiguration delay δ is large (say $\delta \geq 0.02$). More specifically, when $\delta = 0.04$ and $s = 8$, LESS+BFF results in approximately 59% shorter overall makespan than Naive+BFF; when $\delta = 0.08$ and $s = 8$, LESS+BFF results in approximately 74% shorter overall makespan than Naive+BFF. Although error bars representing 95% confidence intervals are

used in Figure 7.3, they are barely noticeable since, for every case (point) in the figure, the simulation results are very close to one another.

Figure 7.3 also shows that when δ is mall (say $\delta \leq 0.01$), LESS+BFF results in similar or slightly longer overall makespan than Naive+BFF. Our explanation is as follows. With a LESS split, the sub-workloads D_1, D_2, ..., D_s are line-even but not identical matrices, and although these s matrices have superb total sparsity (i.e., $\sum_{k=1}^{s} \|D_k\|_0$), there can be some variations in their individual sparsities. Due to these variations among the sub-workload matrices and their individual sparsities, the makespans of the s switches can have some variations. In comparison, with a naive split, all s switches are given identical sub-workloads, so the resulting s makespans are identical. Since the performance metric (overall makespan) is the maximum of these s makespans, this identicalness gives the naive solution a performance edge over LESS. As a result, when δ is very small as in this case or when D is extremely sparse (e.g., in the case of $n_L + n_S = 8$ to be presented in subsection 7.2.3), LESS's performance gain from the total sparsity of the split could be dwarfed by naive solution's performance edge from this identicalness.

This said, LESS+BFF may still outperform Naive+BFF in this case (of $\delta = 0.01$) when the performance metric is changed to the average makespan of all s switches. For example, when $\delta = 0.01$ and $s = 4$, although the overall makepsan for LESS+BFF ($= 0.4516$) is longer than that for Naive+BFF ($= 0.4270$), the average makepsn for LESS+BFF ($= 0.4192$) is actually shorter than that of Naive+BFF ($= 0.4270$).

7.2.3 Under Different Traffic Demands

In this section, we compare the overall makespan performances of LESS+BFF and Naive+BFF under a diverse 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 8 to 64, 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 s: (1) $\delta = 0.02, s = 4$; (2) $\delta = 0.04, s = 4$; (3) $\delta = 0.02, s = 8$; and (4) $\delta = 0.04, s = 8$.

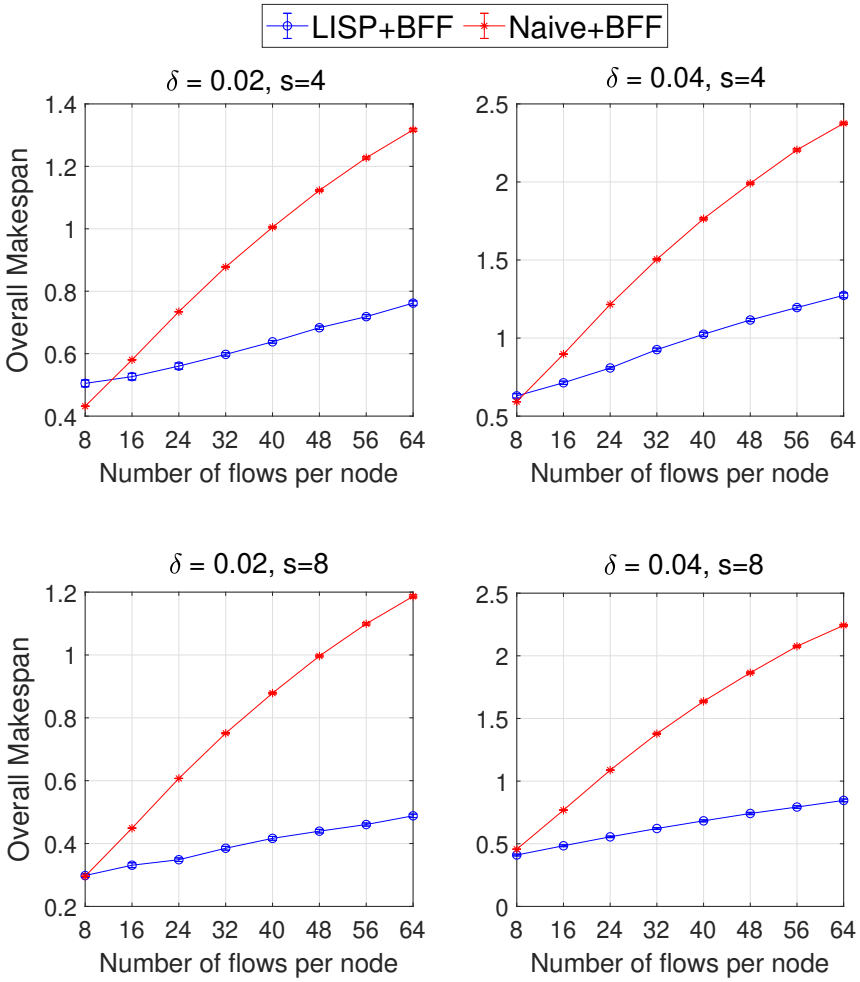

Figure 7.4: LESS+BFF vs. Naive+BFF while varying sparsity of D

Figure 7.4 compares the overall makespan performances of LESS+BFF and Naive+BFF when the sparsity parameter $n_L + n_S$ varies from 8 to 64 and the value of the skewness parameter c_S is fixed at 0.3. Figure 7.5 compares the overall makespan performances of LESS+BFF and Naive+BFF when 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 s above. Both Figure 7.4 and Figure 7.5 show that LESS+BFF invariably results in shorter overall makespans than Naive+BFF, under various traffic demand matrices. In Figure 7.4, LESS+BFF performs consistently better than Naive+BFF, except in some cases where the traffic matrices are extremely sparse (more specifically where $n_L + n_S = 8$).

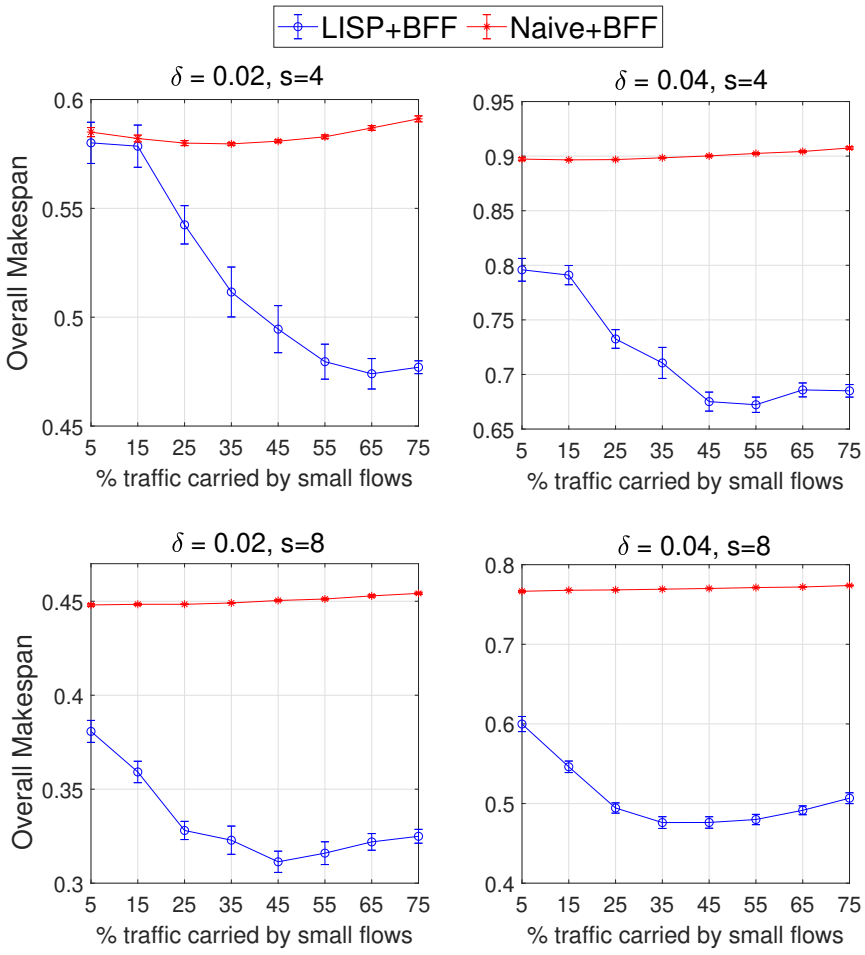

Figure 7.5: LESS+BFF vs. Naive+BFF while varying skewness of D

Although the reason for these outliers has been explained in the previous section, we zoom in on the case of $n_L + n_S = 8$ and $\delta = 0.02$ to emphasize that LESS is "not to blame". In this case, the average number of nonzero entries in a sub-workload matrix resulting from Naive is 731 (or 7.31 per row or column) whereas that from LESS is only 352 (or 3.52 per

row or column). Hence on average, an input port pays 7.31δ reconfiguration cost in the case of Naive and 3.52δ reconfiguration cost in the case of LESS. However, when $\delta = 0.02$, this advantage of LESS in reconfiguration cost is dwarfed by the identicalness advantage enjoyed by Naive.

7.2.4 Sparsity Evenness of LESS

In this section, we show that, the s sub-workload matrices resulting from LESS generally have similar sparsities (numbers of nonzero entries) empirically as measured by their normalized mean absolute deviation (NMAD), although as explained earlier this property is not theoretically guaranteed. The mean absolute deviation (MAD, or average absolute deviation) of a data set $\{x_1, x_2, ..., x_n\}$ is defined as the average distance between x_i and its mean $\bar{x}$: $\frac{1}{n}\sum_{i=1}^{n} \mid x_i - \bar{x} \mid$. The normalized mean absolute deviation (NMAD) is defined as MAD divided by $\bar{x}$. Smaller NMAD means better evenness. Table 7.1 shows the mean and the 95% percentile of NMAD of $\{\|D_1\|_0, \|D_2\|_0, ..., \|D_s\|_0\}$ (the number of nonzero entries in the s sub-workload matrices), for $s = 4$ and $s = 8$, under D with the default parameter settings ($n_L = 4$, $n_S = 12$, $c_L = 0.7$, $c_S = 0.3$). Table 7.1 shows that the average NMAD is only 5% (i.e., deviates 5% from the mean on average) when $s = 4$ and only 4% when $s = 8$.

	$s = 4$	$s = 8$
Mean NMAD	5.00%	4.01%
95%-percentile NMAD	7.36%	5.61%

Table 7.1: Variations among $\{\|D_1\|_0, \|D_2\|_0, ..., \|D_s\|_0\}$

7.2.5 Execution Times of LESS

In this section, we compare the (single-processor) execution times of LP-based LESS and combinatorial LESS, both implemented in C++, under D with the default parameter settings ($n_L = 4$, $n_S = 12$, $c_L = 0.7$, $c_S = 0.3$), on an Apple MacBook Air laptop equipped

with an 1.6 GHz Intel Core i5 processor and 8 GB 2133 MHz LPDDR3. We select Gurobi [76] as the LP solver in the former algorithm due to its superior computational efficiency. As shown in Table 7.2, the execution times of the combinatorial LESS are roughly an order of magnitude shorter than those of LP-based LESS.

	$s = 2$	$s = 4$	$s = 8$
Combinatorial	11.51ms	23.35ms	35.26ms
Gurobi [76]	85.72ms	216.55ms	431.02ms

Table 7.2: Execution Time Comparison

The former are already generally lower than the execution times of BFF (the underlying optical/hybrid switching algorithm), which as reported in [16] is much more computationally efficient than any other hybrid switching algorithm. With parallel processing (described in subsubsection 7.1.2), the former can be further improved by 20% to 40%, as we have estimated through experiments.

This said, as mentioned earlier, the epoch duration is typically a few milliseconds long (e.g., $3ms$), so ideally the execution time of LESS should be no more than that. Currently, with software implementation, our combinatorial algorithm takes roughly an order of magnitude longer, when $n = 100$ (i.e., 100×100 switch) and $k = 8$ (parallel switches). However, we believe this execution time gap can be closed with ASIC implementation, because our combinatorial algorithm is heavy on memory I/O (mostly linked list traversals), which can be done much faster if all data reside in on-chip SRAM. The SRAM cost of ASIC implementation is quite low: Only tens of KBs of SRAM is needed when $n = 100$ and $k = 8$.

Although the focus of this work is on the MSB problem and the LESS solution, we understand that for LESS to be practically useful, its "companion" scheduler, which is chosen to be BFF in this work, also should have an execution time not exceeding the epoch duration. While with software implementation BFF [16] takes around $20 - 30ms$ to compute a schedule when $n = 100$, we believe it too can be sped up by an order of magnitude, by

replacing the expensive maximum weighted matching (MWM) computation (at the beginning) with a much less expensive but slightly lower-quality matching computation (e.g., using iSLIP [79]) and by using the ASIC implementation.

CHAPTER 8
CONCLUSION

This thesis proposed three algorithms that exploit different methodologies for the hybrid and optical switching scheduling problem. We evaluate the performance of three algorithms and compare it with that of the state-of-the-art algorithm. For our simulation, we use both the constructed traffic demand matrices used in the previous works and the generated traffic matrices from real-world traces. The results show that all of the three algorithms outperform the state-of-the-art algorithm in different metrics (i.e., transmission time, execution time, etc). These three algorithms have different appliable conditions and advantages, and we comprehensively compare and summarize them in chapter 6.

Another contribution of this thesis is, we investigated a closely related research problem about the optical and hybrid switching scheduling, that is, when the racks of servers are connected by multiple independent(i.e., parallel) optical switches, how to split the traffic demand matrix into sub-workload matrices and give them to the parallel optical switches as their respective workloads. We formulate it as a matrix split and balance problem and develop a general algorithm to split a matrix into balanced and sparse matrices. Our evaluation results show that, using this matrix split algorithm, parallel optical switches deliver balanced and ideal throughput performance under various system parameter settings and various traffic demands.

Appendices

APPENDIX A

PROOF OF NP-COMPLETENESS

In this section, we prove the Open Shop Scheduling (OSS) problem when allowing preemption and having uniform interprocessor time delay is NP-complete. This proof uses the same reduction in [70], which is used for proving a "weaker" statement, that is, the Open Shop scheduling problem when *NOT* allowing preemption and having uniform interprocessor time delay is NP-complete. The proof in [70] can also be applied to prove the "stronger" (i.e., the former) statement under slightly modifications. As in [70], we choose PARTITION, which is proved to be NP-complete in [80], to do reduction.

Preemptive Open Shop with uniform delays

Instance: A set of n jobs, each job, j, having an associated vector of nonnegative integers, $(l_{1j}, l_{2j}, \cdots, l_{nj})$, where l_{ij} is the time to process the job on machine i ($i = 1, 2, \cdots, n$); a uniform interprocessor time delay, $\delta \in \mathbb{N}^+$ and a bound $L \in \mathbb{N}^+$.

Question: Is there a valid open shop schedule with completion time $\leq L$?

Partition

Instance: Finite set A with cardinality of n (i.e., $|A| = n$), and a size function $s(a) \in \mathbb{N}^+$ for each $a \in A$.

Question: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a) \tag{A.1}$$

Without loss of generality, we can assume all instances of PARTITION have $\sum\{s(a) \mid a \in A\} = 2M$ for some integer M; otherwise the instance is trivially a *NO* instance. Moreover, by introducing a scaling factor, we can assume each $s(a) \geq 2$. We then define the reduction function f by mapping each $a \in A$ into job j_a with the associated process-

ing time vector of $(s(a), s(a), 0, \cdots, 0)$. We also introduce four *capital* jobs, denoted by V, W, X, Y. The associated vectors are $(1, M, 0, \cdots, 0)$, $(1, M, 0, \cdots, 0)$, $(M, 1, 0, \cdots, 0)$ and $(M, 1, 0, \cdots, 0)$ respectively. Thus, given an instance I of PARTITION, the constructed instance $f(I)$ of OSS comprises these $n + 4$ jobs, a uniform delay $\delta = 2M$, and a bound $L = 4M + 2$. Note that only the first two machines have tasks to process, the rest of the machines do not. Clearly, f is a polynomial transformation. We only need to show that $I \in Y_{\text{PARTITION}}$ iff $f(I) \in Y_{\text{OSS}}$. Here $Y_{\text{PARTITION}}$ denotes the set of PARTITION instances that satisfies Equation A.1, and Y_{OSS} denote the set of OSS instances that have a completion time $\leq L$.

(1) $I \in Y_{\text{PARTITION}} \Rightarrow f(I) \in Y_{\text{OSS}}$: Assume A' is a subset of A that satisfies

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a) = M$$

Then a valid schedule $S(t)$ for $f(I)$ with delay $2M$ and of length $4M + 2$ is given in Figure A.1. It induces $f(I) \in Y_{\text{OSS}}$.

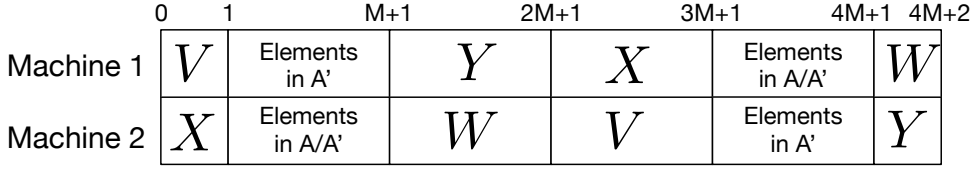

Figure A.1: A valid constructed schedule of length $4M + 2$

(2) $f(I) \in Y_{\text{OSS}} \Rightarrow I \in Y_{\text{PARTITION}}$: Consider the valid schedule $S(t)$ of OSS with a completion time $\leq 4M + 2$. Since machine 1 and machine 2 each has tasks of length $4M + 2$ to process, the schedule $S(t)$ must be an *optimal schedule* (i.e., with the shortest completion time) with duration of exactly $L = 4M + 2$ and have no idling.

As in [70], we call a schedule a *staged schedule* if and only if the schedule has the following two properties: (1) The schedule is non-preemptive; and (2) There exists time T_1 and T_2 such that for all $t \leq T_i$, machine i processes first tasks and for all $t > T_i$, machine

i processes second tasks. Here the *first task* of a job is the task that is processed firstly (i.e., earlier than the other task), and the *second task* of a job is the task that is processed secondly (i.e., later than the other task). In this case, we have the following lemma, which will be proved later in section A.1.

Lemma 2. *$S(t)$ can be converted into a staged optimal schedule in polynomial time.*

Based on Lemma 2, we can simply assume $S(t)$ is a staged schedule (or we consider the staged optimal schedule that is converted from $S(t)$). Note that at machine 1, the last processed task before time T_1 corresponds to a job that also has a second task to be processed by machine 2. The completion time of the job is at least $T_1+\delta+1$ (the completion time of the first task plus a delay of δ and the lower bound of the completion time of the second task, which is 1). This completion time should be less than or equal to L, which induces that $T_1 \leq 2M+1$. Similarly, we have $T_2 \leq 2M+1$. Also, no second task can possibly be process before time $1+\delta$ (the lower bound of the completion time of the corresponding first task plus a delay). Hence we have both T_1 and T_2 are larger than or equal to $2M+1$, which induces $T_1 = T_2 = 2M+1$.

To reach the equality in the above analysis, both the first processed task and the last processed task at machine 1 and machine 2 each must have size 1. Note that only the four capital jobs have a task of size 1 in each (since $s(a) \geq 1$ for $\forall a \in A$). So these four size-1 tasks must be scheduled within the interval of $[0,1]$ or $[4M+1, 4M+2]$ at machine 1 or machine 2. Furthermore, the corresponding four size-M tasks (of the capital jobs) must be scheduled within the interval of $[M+1, 2M+1]$ or $[2M+1, 3M+1]$ at machine 1 or machine 2. In other words, the only valid schedule must have the capital jobs filling the shaded area of Figure A.2.

The remaining jobs must be processed at the unshaded area. Let $A' \triangleq \{a \mid j_a$ is a first job on machine $1\}$. Then we have $\sum_{a \in A'} s(a) = M$, so $I \in Y_{\text{PARTITION}}$.

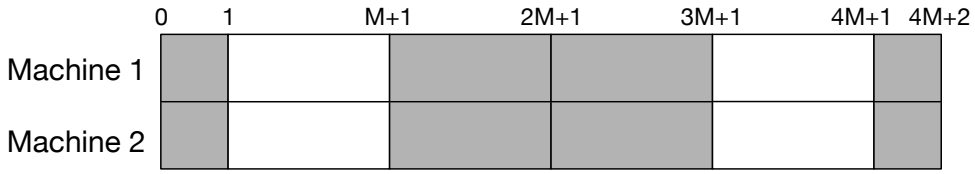

Figure A.2: The only valid schedule structure of length $4M + 2$

A.1 Proof of Lemma 2

In this section, we show how to convert $S(t)$ to a staged optimal schedule in three steps.

First, we consider an arbitrary job that is preemptively processed in $S(t)$. Note that each job can only be transported once between the two machines, otherwise the completion time of the job is at least $2\delta + 4 = 4M + 4$ (two times of interprocessor delay plus the lower bound of the processing time of the job), which is larger than L. Therefore in $S(t)$, one of the task of the job must be processed completely (at one machine) before being transported to the other machine to process the other task. In other words, the processing time intervals of the job must be like the shaded areas of one of the diagrams in Figure A.3. Due to this property, we can also classify the two task into a first task and a second task for a preemptively processed job, as we did for a nonpreempively processed job.

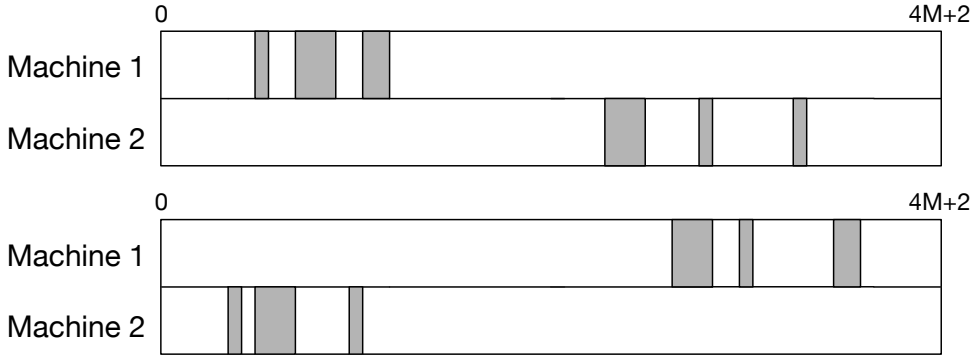

Figure A.3: Possible processing time intervals of the job

Second, we "sort" the order of tasks and sub-tasks (of the preemptively processed tasks) to achieve the following property: For machine $i = 1, 2$, there exists a time T_i such that for all $t \leq T_i$, machine i processes first tasks (or the sub-tasks of first tasks) and for all $t > T_i$,

machine i processes second tasks (or the sub-tasks of the second tasks).

Without loss of generality, assume machine 1 processes the second task (or a sub-task of it) of job j immediately before the first task (or a sub-task of it) of job k. We can interchange these two tasks and still obtaining a valid optimal schedule. Repeating this swap steps and we will get a valid optimal schedule has the above property.

Third, we "merge" the sub-tasks of the same task into an entirety. For the sub-tasks of a first task, we slide all the sub-tasks towards the last sub-task and concatenate them. The order of the rest task remains the same. Then the resulting schedule is also valid. Similarly, for the sub-tasks of a second task, we slide all the sub-tasks towards the first sub-task and concatenate them. The sliding and concatenation process of a job is shown in Figure A.4.

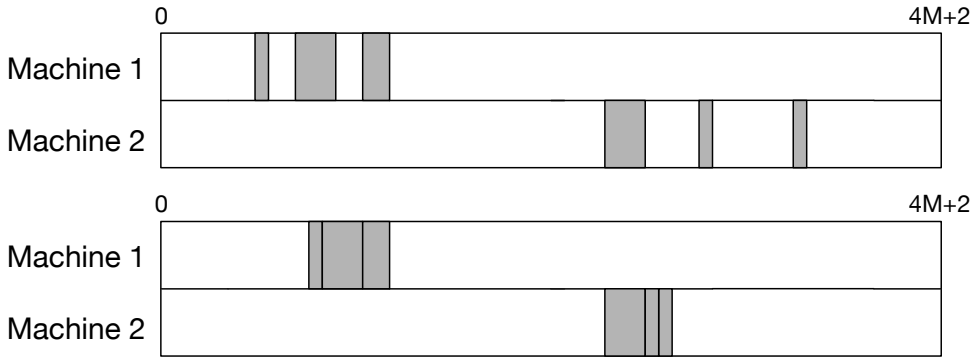

Figure A.4: The sliding and concatenation process of a job (Top: before sliding, Bottom: after sliding)

After the above three processes, the resulting schedule (from $S(t)$) is an optimal and staged schedule. It is obvious the conversion can be done in polynomial time.

REFERENCES

[1] L. Liu, L. Gong, S. Yang, J. J. Xu, and L. Fortnow, "2-hop eclipse: A fast algorithm for bandwidth-efficient data center switching," in *International Conference on Cloud Computing*, Springer, 2018, pp. 69–83.

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

[3] L. Liu, J. Xu, and L. Fortnow, "Quantized bvnd: A better solution for optical and hybrid switching in data center networks," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, IEEE, 2018, pp. 237–246.

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

[5] 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.*, ACM, vol. 43, 2013, pp. 207–218.

[6] 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.

[7] 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, OTuP3.

[8] 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, OW3H–3.

[9] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky, G. Porter, and A. C. Snoeren, "Scheduling techniques for hybrid circuit/packet networks," in *ACM CoNEXT*, ser. CoNEXT '15, Heidelberg, Germany: ACM, 2015, 41:1–41:13, ISBN: 978-1-4503-3412-9.

[10] 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.

[11] 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.*, ACM, vol. 40, 2010, pp. 327–338.

[12] 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.

[13] 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, 2013.

[14] X. Li and M. Hamdi, "On scheduling optical packet switches with reconfiguration delay," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 7, pp. 1156–1164, 2003.

[15] 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.

[16] L. Liu, L. Gong, S. Yang, J. Xu, and L. Fortnow, "Best first fit (bff): An approach to partially reconfigurable hybrid circuit and packet switching," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 426–433.

[17] C.-S. Chang, W.-J. Chen, and H.-Y. Huang, "On service guarantees for input-buffered crossbar switches: A capacity decomposition approach by birkhoff and von neumann," in *Quality of Service, 1999. IWQoS'99. 1999 Seventh International Workshop on*, IEEE, 1999, pp. 79–86.

[18] D. Birkhoff, "Tres observaciones sobre el algebra lineal," *Universidad Nacional de Tucuman Revista , Serie A*, vol. 5, pp. 147–151, 1946.

[19] F. Dufossé and B. Uçar, "Notes on birkhoff–von neumann decomposition of doubly stochastic matrices," *Linear Algebra and its Applications*, vol. 497, pp. 108–115, 2016.

[20] 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*, Florianopolis, Brazil, 2016, pp. 216–229, ISBN: 978-1-4503-4193-6.

[21] 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*, Chicago, Illinois, USA, 2014, pp. 319–330, ISBN: 978-1-4503-2836-4.

[22] E. Amaldi and V. Kann, "On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems," *Theoretical Computer Science*, vol. 209, no. 1-2, pp. 237–260, 1998.

[23] L. Liu, J. Xu, and M. Singh, "Less: A matrix split and balance algorithm for parallel circuit (optical) or hybrid data center switching and more," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 187–197.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

[33] 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.

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

[35] 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.

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

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

[38] 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.

[39] A. Bianco, P. Giaccone, and M. Ricca, "Scheduling traffic for maximum switch lifetime in optical data center fabrics," *Comput. Netw.*, vol. 105, no. C, pp. 75–88, 2016.

[40] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter, "Rotornet: A scalable, low-complexity, optical datacenter network," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ACM, 2017, pp. 267–280.

[41] K. Xi, Y.-H. Kao, and H. J. Chao, "A petabit bufferless optical switch for data center networks," in *Optical interconnects for future data center networks*, Springer, 2013, pp. 135–154.

[42] S. Vargaftik, K. Barabash, Y. Ben-Itzhak, O. Biran, I. Keslassy, D. Lorenz, and A. Orda, "Composite-path switching," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, ACM, 2016, pp. 329–343.

[43] W. M. Mellette, G. M. Schuster, G. Porter, G. Papen, and J. E. Ford, "A scalable, partially configurable optical switch for data center networks," *Journal of Lightwave Technology*, vol. 35, no. 2, pp. 136–144, 2017.

[44] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, ACM, vol. 39, 2009, pp. 51–62.

[45] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 39, 2009, pp. 39–50.

[46] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers, randomly.," in *NSDI*, vol. 12, 2012, pp. 17–17.

[47] M. Shafiee and J. Ghaderi, "A simple congestion-aware algorithm for load balancing in datacenter networks," *IEEE/ACM Transactions on Networking*, 2017.

[48] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 38, 2008, pp. 63–74.

[49] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, ACM, 2011, p. 8.

[50] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *NSDI*, vol. 10, 2010, pp. 19–19.

[51] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 41, 2011, pp. 266–277.

[52] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, pp. 51–62, 2007.

[53] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 465–478, 2015.

[54] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 44, 2014, pp. 503–514.

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

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

[57] S. Even, A. Itai, and A. Shamir, "On the complexity of time table and multi-commodity flow problems," in *FOCS*, IEEE, 1975, pp. 184–193.

[58] L. R. Ford Jr and D. R. Fulkerson, "A suggested computation for maximal multi-commodity network flows," *Management Science*, vol. 5, no. 1, pp. 97–101, 1958.

[59] T. C. Hu, "Multi-commodity network flows," *Operations research*, vol. 11, no. 3, pp. 344–360, 1963.

[60] N. Garg and J. Koenemann, "Faster and simpler algorithms for multicommodity flow and other fractional packing problems," *SIAM J. Comput*, vol. 37, no. 2, pp. 630–652, 2007.

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

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

[63] J. E. Hopcroft and R. M. Karp, "An $n\hat{5}/2$ algorithm for maximum matchings in bipartite graphs," *SIAM Journal on computing*, vol. 2, no. 4, pp. 225–231, 1973.

[64] J. Von Neumann, "A certain zero-sum two-person game equivalent to the optimal assignment problem," *Contributions to the Theory of Games*, vol. 2, pp. 5–12, 1953.

[65] J. Chou and B. Lin, "Birkhoff-von neumann switching with statistical traffic profiles," *Computer Communications*, vol. 33, no. 7, pp. 848–851, 2010.

[66] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 4, pp. 889–901, 1999.

[67] C. E. Koksal, R. G. Gallager, and C. E. Rohrs, "Rate quantization and service quality over single crossbar switches," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, IEEE, vol. 3, 2004, pp. 1962–1973.

[68] B. Lin and I. Keslassy, "A scalable switch for service guarantees," in *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, IEEE, 2005, pp. 93–99.

[69] R. Cole, K. Ost, and S. Schirra, "Edge-coloring bipartite multigraphs in o (e log d) time," *Combinatorica*, vol. 21, no. 1, pp. 5–12, 2001.

[70] 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.

[71] Y. Azar and I. Gamzu, "Efficient submodular function maximization under linear packing constraints," in *International Colloquium on Automata, Languages, and Programming*, Springer, 2012, pp. 38–50.

[72] I Bárány and T Fiala, "Nearly optimum solution of multimachine scheduling problems," *Szigma*, vol. 15, no. 3, pp. 177–191, 1982.

[73] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 123–137.

[74] C. Avin, M. Ghobadi, C. Griner, and S. Schmid, "On the complexity of traffic traces and implications," 2020.

[75] N. Farrington, A. Forencich, G. Porter, P.-C. Sun, J. E. Ford, Y. Fainman, G. C. Papen, and A. Vahdat, "A multiport microsecond optical circuit switch for data center networking," *IEEE Photonics Technology Letters*, vol. 25, no. 16, pp. 1589–1592, 2013.

[76] I. Gurobi Optimization, "Gurobi optimizer 8.1.1," *URL http://www.gurobi.com*, 2019.

[77] L. C. Lau, R. Ravi, and M. Singh, *Iterative methods in combinatorial optimization*. Cambridge University Press, 2011, vol. 46.

[78] W. M. Mellette, A. C. Snoeren, and G. Porter, "Toward optical switching in the data center," in *High Performance Switching and Routing*, IEEE, 2018.

[79] N. McKeown, "The islip scheduling algorithm for input-queued switches," *IEEE/ACM transactions on networking*, vol. 7, no. 2, pp. 188–201, 1999.

[80] R Karp, "Reducibilities among combinational problems," in *Complexity of Computer Computations*, Plenum Press New York, 1972.

VITA

Liang Liu received his B.S. degree in Electronic Engineering from Shanghai Jiao Tong University, China in 2014. Liang joined School of Computer Science, Georgia Institute of Technology, as a Ph.D. student in August 2014. His thesis work was conducted under the able guidance of Dr. Jun (Jim) Xu.