RICE UNIVERSITY

Scheduling Optical Circuits in Data Center Networks

by

Xin Huang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Houston, Texas

February, 2016

ABSTRACT

Scheduling Optical Circuits in Data Center Networks

by

Xin Huang

Data center driven by optical circuit switching network, or optical data center, is emerging as an alternative to traditional data center where the electrical packet switching network is already overwhelmed by bulk data transfer. Optical data center promises high bandwidth capability, but it is set against circuit reconfiguration delays, which makes circuit scheduling non-trivial. The optical circuit scheduler must manage traffic over both hybrid and pure optical network architectures, sparse and dense traffic patterns, and scale to large network sizes. In this thesis, we show that the proposed algorithms for circuit scheduling in optical data center fail to meet these goals. To address their deficiencies, we introduce a scheduling algorithm called Decomp. We show that regardless of hybrid or pure architectures, sparse or dense traffic, Decomp simultaneously eliminates the long-tailed flow waiting times that existing algorithms suffer from, achieves high network utilization, and maintains a low computational delay as network size scales up.

Acknowledgments

I am immensely grateful to my advisor, T. S. Eugene Ng, who enlightens me with the appreciation for great work, ample guidance for good research, and various aspects for effective presentation. He gives me freedom to explore the problems, as well as timely feedback on my studies. I am especially fortunate to work with him and I look forward to our collaboration in my PhD studies.

I am also thankful to my thesis committee members, Alan Cox and Chris Jermaine, for their useful feedback.

I am grateful to many colleagues in Rice, especially the members in the BOLD Lab. I have been fortunate to work with Xiaoye Sun, Simbarashe Dzinamarira, Yiting Xia, Zhaolei Fred Liu, and Ruiqi Liu. I learned a lot from each of them.

I would like to express my deepest gratitude to my parents for their love and support. They encourage me to do what I love and to love what I do. My mother instills in me perseverance and my father teaches me the sense of responsibility.

Last but not least, I want to thank my friends for their support throughout my studies.

Contents

Illustrations

Tables

Chapter 1

Introduction

The rising tide of big data driven cluster computation is imposing heavy traffic in data center network. In face of high traffic demand, conventional data centers built with layers of electrical packet switches is fundamentally challenged. Copper cables can no longer support high bit rate at a distance more than a few meters. Besides, large bisection bandwidth requires a highly layered topology with a vast number of switches which leads to more complicated management. Instead, people in both academia [1, 2, 3, 4, 5, 6] and industry [7, 8] are turning to optical fabrics and switches for next generation data center solutions.

However, upgrading to an optical data center is non-trivial. The circuit switching delay of commercially available optical switches (e.g. WSS and 3D-MEMS) are 2 to 5 orders of magnitude slower than what's needed to switch at the packet granularity. Therefore, the switching decisions greatly impact network performance and sophisticated management of optical circuits plays a critical role in optical data centers. As a basic requirement, the circuit scheduler has to achieve low flow waiting times and high network utilization. In addition, the leading proposals for optical data centers [1, 2, 3, 4, 5, 6] and the traffic patterns of data intensive applications [9] point to three requirements that a circuit scheduler has to address:

Handle Hybrid and Pure Architectures Several network architectures have been proposed for optical data centers. One line of work [1, 2, 3, 6] is in hybrid architecture, which combines electrical packet switching with optical circuit switching.

The top-of-rack switches are connected to a conventional electrical packet switched network and an optical circuit switched network. These networks operate side-by-side and heavy traffic flows are off-loaded to the optical circuit switched network. Another line of work [4, 5] is in pure architecture, where the top-of-rack switches are connected solely to an optical circuit switched network. Here, the fastest available optical switches must be employed to minimize the negative performance impact of circuit switching delays. It is also worth noting that the hybrid architecture behaves similarly to the pure architecture if it possesses only a small amount of electrical bandwidth. In data centers where the bisection bandwidth might change vastly due to congestion or link failure, the circuit scheduler must function well regardless of whether the network behaves like a hybrid or a pure optical network.

Robustness under Different Traffic Patterns Data intensive applications generate traffic that has a wide range of traffic density [9]. A data center can carry an arbitrary set of applications that vary, resulting in vastly different traffic density over time. At one time, jobs can trigger multiple one-to-all and all-to-one flow groups, resulting in a uniform dense traffic pattern. At another, there might be only a few large flows among several select machine racks, resulting in a skewed sparse traffic pattern. The circuit scheduler must be robust to different traffic patterns.

Scalability Data intensive applications' storage and computation demands are driving data centers to expand. Particularly in this era of cloud computing, the number of servers in a data center is growing continuously to meet operational requirements. In an optical data center, the circuit scheduler can become a critical bottleneck for network scalability. Therefore, the circuit scheduler must be sufficiently scalable.

It turns out that meeting these requirements cannot be taken for granted. In Figure 1.1, we illustrate a three-dimensional design space of circuit scheduling algo-

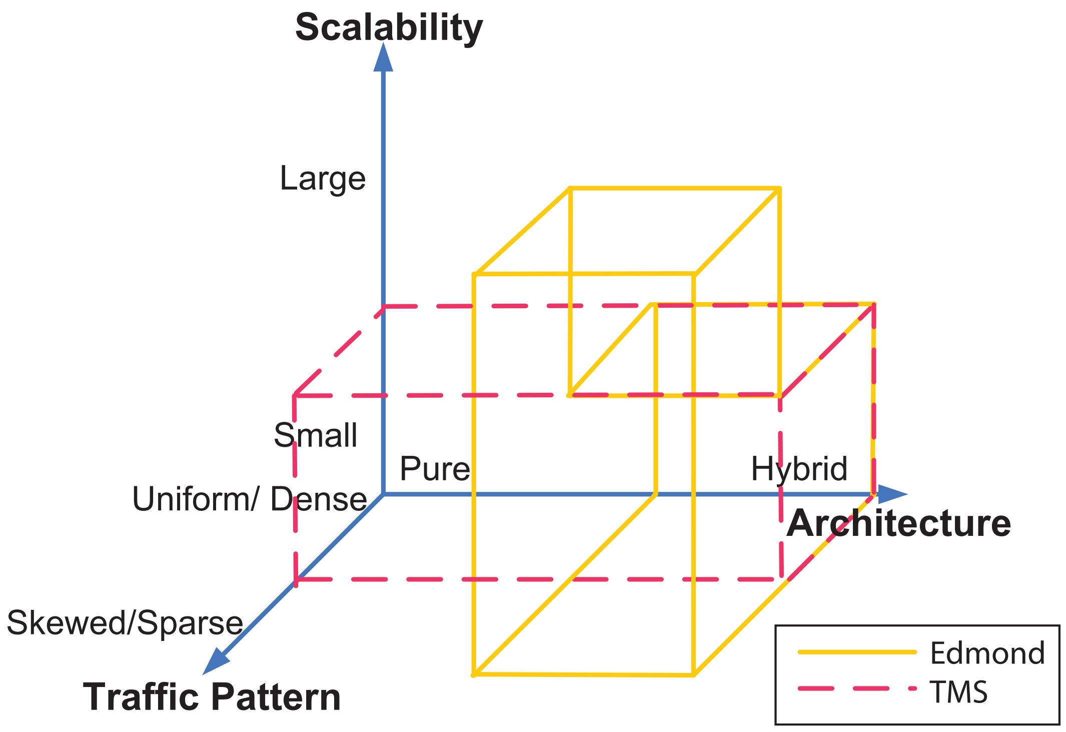

Figure 1.1 : Design space of circuit scheduling algorithms. The feasible zone of Edmond is outlined with the solid line and TMS with the dashed line. These algorithms fail to perform well in all three dimensions.

rithms. Each point in the design space represents a situation that a circuit scheduler might encounter. For stable network performance, a circuit scheduler should function well under various data center sizes, from small to large, different architectures, from pure to hybrid, and unpredictable traffic patterns, from uniform to skewed, dense to sparse.

Unfortunately, in this thesis, we show that the optical data center circuit schedulers proposed in previous works [1, 2, 4, 5] fail to perform well in all three dimensions (scheduling algorithms are shown in Chapter 3). Specifically, the Edmond's algorithm, which we refer to as Edmond hereafter, is used in [1, 2] to maximize the traffic volume over the optical network. In this setting, however, the small flows will starve in the pure optical network or in the hybrid architecture with little electrical bandwidth. Further, Edmond's computation time is very high for a large network with dense traffic. The Traffic Matrix Scheduling (TMS) algorithm proposed in [4, 5] mitigates the starvation problem, but it is not scalable computationally. Its computation time for sparse traffic even under a small network size is prohibitive; moreover, it also schedules wasted circuits due to a distortion of traffic demand in one of its computation steps. Circuits are valid if they have non-zero traffice demand, and therefore *valid circuits* will effectively clear traffic. However, *wasted circuits* are idle circuits that have no demand to serve. The wasted circuits are costly not only because they cause expensive switch reconfigurations, but they may also prohibit valid circuits, further degrading network performance. The feasible zones of Edmond and TMS are outlined by the solid line and the dashed line in Figure 1.1.

To address the multi-dimensional challenges of optical data centers, we propose a circuit scheduling algorithm called Decomp. Decomp can be seen as a framework. Unlike other algorithms, Decomp structures the circuit schedule computation into stages and can be customized with different circuit selection policies. Decomp also employs three key approaches that are not found in previous proposals, namely partitioning, randomization, and parallelization (details are presented in Chapter 5).

We choose simulation as the methodology for evaluating different scheduling algorithms since it allows us to precisely characterize and compare the performance of the scheduling algorithms, and allows us to stress these algorithms at large network sizes.

The results show that regardless of hybrid or pure architectures, sparse or dense traffic, Decomp simultaneously eliminates the long-tailed flow waiting times that existing algorithms suffer from, achieves high network utilization, and maintains a low computational delay as network size scales up. We observe cases where Decomp clears up to 55% more traffic flows than Edmond, or 13% more than TMS within the same length of time; Decomp takes 28000× less time to compute a schedule among 600 racks than TMS, or 170× less than Edmond.

The rest of this thesis is organized as follows. In Chapter 2, we provide the background for the optical circuit scheduling. In Chapter 3, we discuss on a range of related work on circuit scheduling, which should be helpful to readers who are not familiar with this area. We uses examples in Chapter 4 to illustrate the inefficiency behind the algorithms of Edmond and TMS, which are two representative optical circuit scheduling algorithms in the context of data center network. In Chapter 5, we present the design of the Decomp algorithm and further compare the performance of Edmond, TMS, and Decomp in Chapter 6. Finally, we summarize our contributions and conclude in Chapter 7.

Chapter 2

Background

2.1 Network Model

As shown in Figure 2.1, we model the core network of the data center as a hybrid network consisting of an electrical packet switch with bisection bandwidth B^e, and an optical circuit switch with bisection bandwidth B^o ($B^o >> B^e$).

The sources and destinations behind the input and output ports of the switches can vary in practice (e.g. rack-to-rack, pod-to-pod); without loss of generality, we assume a rack-to-rack granularity in the discussions that follow.

In Figure 2.1, each top-of-rack (ToR) switch is both a sender and receiver. A ToR switch will aggregate the traffic from the servers within its domain and route traffic inbound and outbound traffic accordingly. Each ToR is assigned an input port and an output port to each of the electrical and optical switch. However, traffic between two ToR switches can only go through either the electrical switch or the optical switch. By default, traffic will go through the optical switch when a circuit in the corresponding direction is set up, while all other traffic is left to the electrical switch. Note that although traffic from ToR_i to ToR_j will travel through the optical switch if circuit from ToR_i to ToR_j is set up, however, traffic in the reverse direction (from ToR_j to ToR_i) is unaffected and depends on circuit from ToR_j to ToR_i to decide which one of the two switches to travel through.

Note that when the electrical bandwidth $B^e \rightarrow 0$, the network becomes a pure

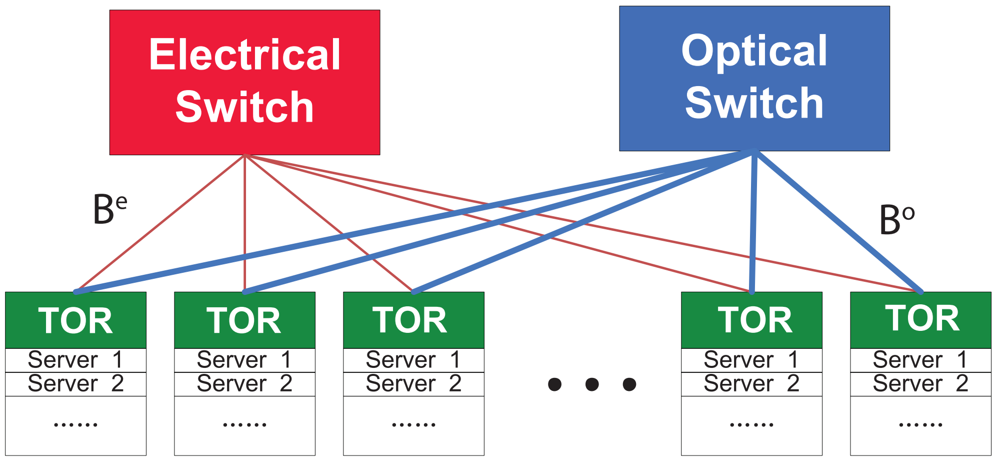

Figure 2.1 : Network model with a electrical packet switch and an optical circuit switch. Each ToR is both a sender and receiver.

optical network, in which all traffic can only travel through the optical switch.

2.2 Optical Circuit Constraints

The optical switch has circuit constraints: only one circuit can be set up between an input port and an output port. Let p denote a circuit assignment of the optical switch at any time. Then p is an $N \times N$ binary matrix, and $p(i, j) = 1$ indicates setting up a circuit from input port i to output port j. Thus p is a permutation matrix corresponding to a one-to-one matching between the input ports and output ports of the optical switch.

The circuit scheduler is to schedule a list of circuit assignments $\{p_1, ..., p_m, ..., p_M\}$ over time, where M is the total number of circuit assignments deployed. Each circuit assignment is coupled with its duration in time $\{t_1, ..., t_m, ..., t_M\}$, so that assignment p_m is active for t_m $(m = \{1, 2, ..., M\})$.

Applying a circuit assignment will incur a reconfiguration delay of δ.

2.3 Traffic Demand Requirement

The scheduling should meet the traffic demand requirement. The traffic demand can be represented as a $N \times N$ matrix D, where N is the number of racks. Each element $D(i, j)$ indicates the byte volume of traffic demand from ToR_i to ToR_j.

2.4 Traffic Scheduling Assumptions

In this study, we focus on the impact of circuit scheduling on network performance. Therefore, we assume the circuit scheduler can only control the circuit assignments $\{p_1, ..., p_M\}$ and the corresponding durations $\{t_1, ..., t_M\}$. Traffic is redirected automatically to the optical switch if the corresponding circuits are set up. The traffic shares the bandwidth with max-min fairness.

Chapter 3

Related Work

3.1 Overview

In the context of data center networks, existing optical circuit schedulers generally apply centralized management and schedule circuits at the beginning of recurring scheduling cycle [1, 2, 4, 5]. A scheduling *cycle* has a length of time T, which is typically fixed and on the order of hundreds of millisecond. Each scheduling cycle can be further divided into *slots*, so that each slot corresponds to a unique circuit assignment.

At the beginning of each scheduling cycle, the scheduler collects the traffic demand, represented as a matrix D. Based on D, the circuit scheduling algorithm computes one or more slot configurations in a list $C = \{c_1, c_2, ..., c_{M_k}\}$, where M_k is the number of slots produced in the k-th scheduling cycle (Figure 3.1). Each slot c_m has a circuit assignment $c_m.p$ and a coefficient $c_m.t$. The coefficient $c_m.t$ indicates the *ratio* of length in time between the slot c_m and the scheduling cycle, i.e. the *time share* of c_m in a cycle. In other words, a slot with time share $c_m.t$ has a length of $c_m.t \times T$ in time ($t_m = c_m.t \times T$ in Figure 3.1), during which circuits in $c_m.p$ are set up for the optical switch. Note that the sum of slot length within a scheduling cycle is equal to T, i.e. $\sum\limits_{m=1}^{M_k} c_m.t = 1$. Each slot corresponds to a circuit assignment with duration of $c_m.t \times T$ in Section 2.2 of Chapter 2.

Figure 3.1 illustrates this cycle based work flow adopted by existing schedulers. In

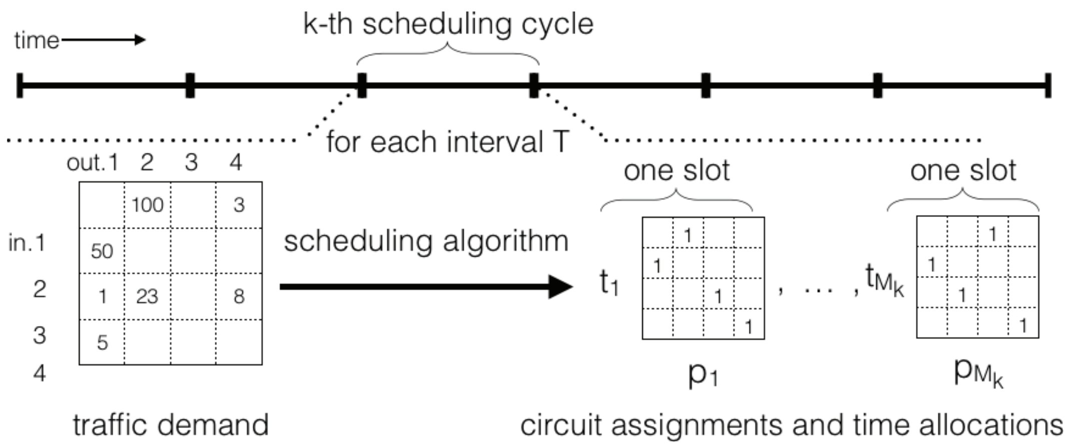

Figure 3.1 : An illustration of existing scheduler's work flow for a 4-port optical switch.

Figure 3.1, for both the traffic demand matrix and the circuit assignment matrices, each row represents an input port and each column represents an output port. The ones in the circuit assignment matrices indicate setting up optical circuits between the corresponding input and output ports.

3.2 Scheduling One Assignment per Cycle

3.2.1 Edmond

In the context of optical circuit scheduling for data center networks, schedulers proposed in [1, 2] use the maximum weighted matching algorithm, or Edmond, as the core scheduling algorithm. Edmond will schedule only *one* slot per scheduling cycle, i.e. $|C| = 1$, and the slot is configured as

$$\begin{aligned} c_1.p &\leftarrow \text{a matching s.t.} \max \langle p, D\rangle \\ c_1.t &\leftarrow 1 \end{aligned} \tag{3.1}$$

where D is the traffic demand matrix. In order words, for each scheduling cycle of length T, Edmond chooses the circuit assignment that maximizes the sum of traffic demand on circuits set up, based on the traffic demand at the beginning of each cycle.

The computation complexity of Edmond is $O(N^3)$ where N is dimension of the demand matrix D, which is the port counts of the optical switch, and thus the rack count inside the data center network.

3.2.2 Other Scheduling Algorithms for Packet Switches

The Edmond algorithm can be traced back to the classic circuit scheduling problem for packet switches, where one circuit assignment is produced for each scheduling cycle to transmit one round of packets. There is a plethora of works that focus on switch scheduling for input queuing packet switches by finding the maximum bipartite matching (MBM) among input and output ports. Edmond[10], PIM [11], SLIP [12], APSARA [13], LAURA [13] and SERENA [13] are some of the matching algorithms that serve this purpose. However, circuit scheduling in optical data centers is different from input queuing switch scheduling. There are two important reasons outlined as follows.

First, in input queuing packet switches, switch scheduling happens at packet granularity, i.e. one MBM is used per scheduling cycle to transmit one round of packets. This is possible because the traffic demand is instantly available to the scheduling algorithm by counting local buffers. However, in optical data centers, the traffic sources are distributed. The core optical switch is bufferless and the traffic is either buffered at packet switches or at hosts. Before scheduling computation, the traffic demand needs to be measured distributedly and communicated to the circuit manager, which incurs a significant and often unpredictable delay. This delay is further compounded by communication overhead, making it impossible to collect traffic demand at packet granularity in optical data centers. Instead, such information is to be collected at coarser granularity with larger update interval on traffic demand.

Second, in input queuing switches, only one MBM is used per scheduling cycle to transmit one round of packets. However, in optical data centers, multiple matchings are generally needed per scheduling cycle.

Consider a slow 3D-MEMS optical switch, which takes tens of milliseconds for circuit reconfiguration. Given such circuit reconfiguration delay and the communication overhead of collecting traffic demand, previous works [1, 2] propose to collect traffic demand and schedule one circuit assignment at O(100 ms) intervals. In a typical data center with 10 Gbps bisection bandwidth, a scheduling interval at O(100 ms) transmits O(80,000) packets, and the communication overhead is an acceptable cost to transmit such amount of data. However, as faster optical switching technologies, such as wavelength-selective switch (WSS), are being proposed, the optical circuit reconfiguration delay drops from tens of milliseconds to tens of microseconds, and can drop even further with silicon photonic switches. On one hand, scheduling at O(100 μs) interval is no longer possible because it causes an unacceptably high communication overhead of collecting distributed traffic demand information. On the other hand, keeping the scheduling interval at the slower O(100 ms) is not desirable because it does not take advantage of the faster switching enabled by these technologies. Such advantages of faster switching include more fine-grained sharing of bandwidth resources among large and small flows, which can significantly increase network throughput and reduce flow starvation problems.

To bridge the gap between coarse update on traffic demand and fast switching of optical circuits, one can schedule *multiple* circuit assignments for one cycle.

3.3 Scheduling Multiple Assignments per Cycle

3.3.1 TMS

TMS is proposed in [4, 5] as the scheduling algorithm for optical circuits in data center networks. It schedules multiple circuit assignments per scheduling cycle. The TMS algorithm relies on the Birkhoff-von Neumann algorithm[14, 15], which we refer to as *BvN* hereafter, to decompose a demand matrix into multiple circuit assignments. BvN assumes its input matrix to have an identical sum for each row and each column. Therefore, the TMS algorithm applies a pre-processing step that uses the Sinkhorn algorithm, referred to as *Sinkhorn* hereafter, and transforms an arbitrary demand matrix to meet the input requirements of BvN. Hence the TMS algorithm consists of two steps, as described in Algorithm 1.

Algorithm 1 TMS scheduling algorithm

```
Input: traffic demand matrix $D$, error tolerance for Sinkhorn $\tau$
Output: a list of slot configurations $C$
1: Doubly Stochastic Matrix $D' \leftarrow$ Sinkhorn($D, \tau$)
2: $C \leftarrow$ BvN($D'$)
```

In the first step, TMS uses the Sinkhorn algorithm, to transform the traffic demand matrix D to a *Doubly Stochastic Matrix* (DSM) D'. A DSM is a matrix whose row sums and column sums are 1, i.e.

$$\begin{aligned} \sum_j D'(i,j) &= 1, \forall i \\ \sum_i D'(i,j) &= 1, \forall j \end{aligned} \tag{3.2}$$

As shown in Algorithm 2, Sinkhorn obtains an DSM by *iteratively* normalizing the rows and columns of a matrix with positive (greater than 0) entries (line 3 - 9).

Algorithm 2 Subroutine: The Sinkhorn Normalization

```
1: procedure SINKHORN(traffic demand $D$, error tolerance $\tau$)
2:     $D' \leftarrow$ Fill zero entries in $D$ with $\sigma > 0$                ▷ Fill zero entries
3:     repeat
4:         $D^{\text{tmp}} \leftarrow 0$
5:         $\forall$ row $i$ in $D'$ : $D^{\text{tmp}}(i,j) \leftarrow \frac{D'(i,j)}{\sum_{j=1}^{N} D'(i,j)}$        ▷ Normalize over rows
6:         $\forall$ column $j$ in $\in D'$ : $D^{\text{tmp}}(i,j) \leftarrow \frac{D'(i,j)}{\sum_{i=1}^{N} D'(i,j)}$    ▷ Normalize over columns
7:         $D' \leftarrow D^{\text{tmp}}$
8:         error $\epsilon \leftarrow \max\left( |\sum_{i=1}^{N} D'(i,j) - 1|, |\sum_{j=1}^{N} D'(i,j) - 1| \right)$
9:     until error $\epsilon <$ tolerance $\tau$
10:    return Doubly Stochastic Matrix (DSM) $D'$
11: end procedure
```

Therefore, the zero entries in D will be replaced by a small value $\sigma > 0$, before the iterative normalization (line 2). The output of Sinkhorn is an approximated DSM, whose row sums and column sums may not be exactly 1, and Sinkhorn repeats the normalizations until the sums are sufficiently close to 1, as specified by the error tolerance (line 9). Sinkhorn's normalization may distort the input demand matrix D, and results in *distorted demand*, which corresponds to entries with zero value in D but non-zero value in the resulting DSM D', i.e. $D(i,j) = 0$, but $D'(i,j) > 0$.

In the second step, TMS generates multiple circuit assignments by decomposing the DSM D' with BvN. As shown in Algorithm 3, BvN is an iterative algorithm for matrix decomposition. In each iteration, BvN produces an assignment and the corresponding coefficient, i.e. time share in the circuit schedule (line 4 - 5), and updates the matrix for the next iteration (line 8). BvN will schedule the circuits

Algorithm 3 Subroutine: The BvN Decomposition

```
1: procedure BvN(DSM $D'$)
2:     a list of slot configurations $C \leftarrow \emptyset$
3:     while $D' \neq \emptyset$ do
4:         $p \leftarrow$ perfect matching over $D'$
5:         $t \leftarrow \min\{D'_{i,j} : (i,j) \in p\}$
6:         slot configuration $c.p \leftarrow p$, $c.t \leftarrow t$
7:         append $c$ to $C$
8:         $D' \leftarrow D' - tp$
9:     end while
10:    return $C$
11: end procedure
```

to serve the *exact* demand specified with the input DSM D'. In other words, the slot configurations produced by BvN satisfy $\sum_{m=1}^{M_k} c_m.p \times c_m.t = D'$ and $\sum_{m=1}^{M_k} c_m.t = 1$. Since the BvN input D' may be distorted by the previous Sinkhorn step, BvN may be misguided, so that the scheduled circuits may poorly serve the original demand matrix D. Besides, BvN comes with an expensive cost of circuit configurations. It may produce up to $O(N^2)$ circuit assignments per scheduling cycle, where N is the port count of the optical switch.

Both Sinkhorn and BvN are iterative algorithms, and thus are hard to be parallelized. The run time of Sinkhorn depends on both the input demand matrix D and the error tolerance τ. The computation complexity of BvN is $O(N^{4.5})$ for an optical switch with N ports. Therefore, the computation complexity of TMS is at least $O(N^{4.5})$.

3.3.2 Other Scheduling Algorithms for Crossbar Switches

A number of algorithms can generate multiple matchings from one demand matrix (e.g. [16, 17, 18]), and the TMS algorithm that we already mentioned [4, 5] is a representative example. These algorithms can be traced back to the classic crossbar switch scheduling problem, or the so-called Time Slot Assignment (TSA) problem[16]. The solution algorithms usually rely on the techniques of matrix decomposition, which is similar to BvN in the sense that they would need to iteratively update and decompose a matrix into multiple assignments, with a weighted coefficient for each assignment. However, these algorithms all suffer from high computational complexity. Furthermore, they are hard to be parallelized because they rely on iteratively updating the matrix to produce one assignment after another. In contrast, we propose a circuit scheduling algorithm that can run in parallel to handle large scale circuit scheduling problem with reasonable computation time.

Chapter 4

Motivating Example

In this chapter, we will illustrate the inefficiency behind the two existing circuit scheduling algorithms with an example. Particularly, we apply the algorithms over the same traffic demand and the scheduling results are shown in Figure 4.1 and Figure 4.2.

4.1 Inefficiency behind Edmond

For each scheduling cycle, Edmond only schedules one set of circuits with one assignment. Traffic on unscheduled circuits can not take advantage of the optical bandwidth and thus are starved in a pure optical network, as shown with the shaded demand in Figure 4.1. Particularly, Edmond schedules circuits based on the maximum weighted matching of the traffic demand, because Edmond considers the corresponding links to be hot spots and thus Edmond caters the circuit schedule for the heaviest demand. As heavy demand also takes longer time to serve, consequently, traffic on circuits with smaller demand may be delayed for long time before they are allocated with optical circuits and benefit from the optical bandwidth. For example in Figure 4.1, $D(1,4)$ and $D(3,2)$ are competing with $D(1,2)$ for the same input port or output port. Edmond would prioritize the circuit for $D(1,2)$, and the circuits for $D(3,2)$ and $D(1,4)$ would need to wait until $D(1,2)$ is served with considerable amount of time to allow circuits for $D(3,2)$ and $D(1,4)$, based on the maximum weighted matching

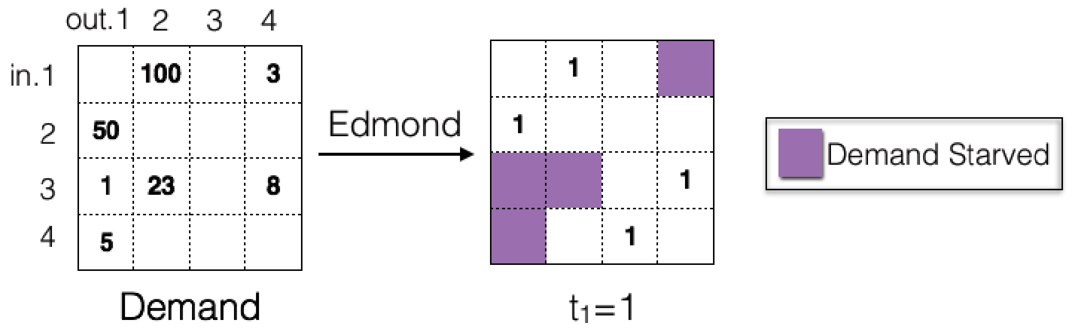

Figure 4.1 : Scheduling result of Edmond. Grape shaded cells indicate demanding circuits NOT covered in the scheduling.

of the traffic demand. In a pure optical network, or a hybrid network with small electrical bandwidth, traffic on circuits with small demand (e.g. $D(3,2)$ and $D(1,4)$) ends up with long delay because they can hardly receive any optical bandwidth when some large demand (e.g. $D(1,2)$) dominates the optical circuits scheduling.

The problem of delaying small traffic is mitigated in the studies on the Edmond scheduling [1, 2], which are based on a hybrid network with sufficiently large electrical bandwidth, so that the traffic on circuits with small demand may take advantage of the abundant electrical bandwidth and avoid starvation, even if they are not allocated with optical circuits.

4.2 Inefficiency behind TMS

TMS schedules multiple circuit assignments to cover more traffic demand in each scheduling cycle. However, TMS's both steps, i.e. Sinkhorn and BvN, would result in significant inefficiency due to wasted circuit resource and heavy switching overhead.

For one thing, Sinkhorn's normalization may heavily distort the original demand and produce a misleading DSM for BvN. Particularly for demand matrix with zero entries, Sinkhorn can bring in distorted demand, by modifying the demand of a circuit

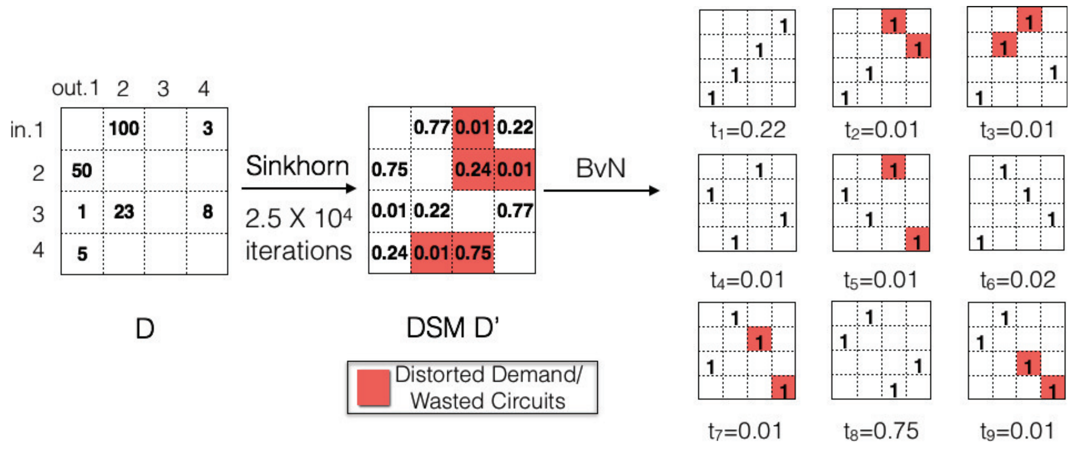

Figure 4.2 : Scheduling result of TMS. All numbers are rounded up to the second decimal. Red shaded cells indicate distorted demand and wasted circuits.

in D' to arbitrary value even even if the circuit has *no* traffic demand in the original demand matrix D, i.e. $D(i,j) = 0$, but $D'(i,j) > 0$. For example in Figure 4.2, the shaded non-zero demand in the DSM D' corresponds to zero demand in the original demand matrix. As the scheduling produced by BvN exactly reflects the input DSM, BvN would schedule *wasted* circuits based on the misleading DSM. As shown in Figure 4.2, multiple wasted circuits are shaded in the assignments produced by BvN. These wasted circuits do not effectively serve any traffic demand. Besides, setting up these wasted circuits largely harms performance because they not only cause switching overhead, but they may also take up the ports required by other circuits with non-zero demand in the original demand matrix. For example in p_3 of Figure 4.2, rather than setting up the two wasted circuits, circuit from input port 1 to output port 2 could be set up to serve traffic demand. In summary, Sinkhorn may distort the original demand matrix heavily so that the resulting circuits scheduled would poorly serve the original demand.

Besides, BvN schedules circuit switching *aggressively*, producing up to $O(N^2)$

assignments per cycle, which may incur excessive circuit reconfiguration overhead as the optical switch port count N grows. The computation overhead of TMS ($O(N^{4.5})$) also grows rapidly with N, which makes TMS less capable to schedule circuits for an optical switch with large port count.

Unfortunately, the Sinkhorn distortion problem and the limited scalability are ignored in the studies of the TMS algorithm [4, 5], which are based on uniform dense traffic demand on a small scale optical switch with 6 ports. For one thing, compared with a skewed and sparse matrix, Sinkhorn's iterative normalization would apply less distortion on a matrix full of uniform non-zero entries. For another, a 6-port switch in the studies is too small to unveil the inefficiency of the overhead due to switching and computation, both of which increasingly grow heavier with larger switch port count.

Chapter 5

Efficient Optical Circuit Scheduling with Decomp

5.1 Algorithm Overview

Decomp can be seen as a framework. Unlike other algorithms, Decomp structures the circuit schedule computation into three stages, namely **partition**, **schedule**, and **merge**, so that 1) the traffic demand matrix is first partitioned into multiple regions, 2) the schedule stage makes circuit selection decisions on different regions, and 3) the merge stage coordinates selected circuits on different regions. Particularly, Decomp employs three key techniques that are not found in previous proposals, i.e. 1) partitioning the demand matrix, 2) randomization before partitioning, and 3) parallelization for scalability, discussed as follows.

5.1.1 Key Techniques

Partitioning the Demand Matrix Existing algorithms such as Edmond or TMS that schedule all circuits over the entire traffic demand matrix allow large traffic demand to easily dominate the use of the optical circuits, which starves small demand in the optical network. To solve this problem, Decomp first partitions the traffic demand matrix into multiple regions and schedules circuits for different regions in isolation (the randomization technique described next further improves the algorithm's robustness against small demand starvation). In this way, Decomp distributes the optical bandwidth among small and large demand because the large demand can now only dominate a single region rather than the entire network. Then, Decomp merges partial scheduling results into a global scheduling decision for all circuits, and enforces high utilization of the optical network in merging.

Randomization before Partitioning A partition of traffic demand for parallel computation might still result in scheduling decisions biasing some circuits over the other. For example, circuits in sparse regions are more likely to be set up than those in dense regions with intensive competition for optical bandwidth. To maintain robust and stable performance over various traffic patterns, Decomp randomizes the labeling of racks before partitioning demand regions, so that one region may represent demand for different subset of optical circuits in each computation iteration. In this way, Decomp removes the chance of persistent bias for certain circuits.

Parallelization for Scalability Decomp's computations can be parallelized in K ways where K is a power of 4 (to maintain the square shape of each sub-matrix). To do so, inter-rack traffic demand matrix is first partitioned into regions, each representing traffic demand for a portion of links between source and destination racks. Next, each parallel process takes in one region and computes partial circuit assignments for links included in the region. In other words, each process controls a subset of all optical circuits and schedules them in the scope of the partitioned region. Finally the partial circuit assignments are merged to obtain the schedule for all circuits.

5.1.2 Decomp Techniques and the Design Space

These key techniques help Decomp to cover the design space of the circuit scheduling algorithm (Chapter 1).

Scalability Decomp can run in parallel to handle scheduling computation for an optical switch with large port count. To avoid high computational overhead, a lightweight heuristic can be applied in the schedule stage. Moreover, the schedule stage can be customized with different circuit selection policies that aim for a variety of performance objectives. Particularly, it can also be explicitly designed to mitigate excessive switching overhead as the switch port count grows.

Handling Hybrid and Pure Architectures As we discussed in Section 5.1.1, compared with the existing algorithms such as Edmond or TMS that schedule all circuits over the entire traffic demand matrix, Decomp schedules circuits based on each partitioned demand regions and distributes the optical bandwidth among small and large traffic demand. The randomization technique also helps to mitigate the large demand dominance problem.

Robustness under Different Traffic Patterns. In contrast with TMS that may apply arbitrary distortion to the traffic demand and misguide the resulting circuit schedule, Decomp may use a heuristic in the schedule stage so that circuits are scheduled based on the *original* traffic demand. Moreover, the merge stage of Decomp is also designed to respect the scheduling results on different regions, so that the ultimate circuit scheduling may better matches the demand originally requested.

5.2 The Decomp Algorithm

As described in Algorithm 4, Decomp has three major stages, i.e. partition, schedule and merge. As a pre-processing step, the order of racks is randomized with a random permutation function f^r (line 1). The randomization is repaired by mapping the sources and destinations of circuits in the output configurations back to the corresponding racks using the reverse function of f^r (line 17).

Algorithm 4 Decomp scheduling algorithm

Input: traffic demand matrix D, parallel in 4^L-way

Output: a list of slot configurations C

```
 1: $f^r \leftarrow$ random permutation of $\{1, ..., N\}$                          ▷ Randomization
 2: construct randomized demand $D^r$ s.t. $D^r(i,j) = D(f^r(i), f^r(j))$
 3: divide $D^r$ into $4^L$ regional SDMs $\{D^s_1, ..., D^s_{4^L}\}$                 ▷ Partition
 4: for all SDM $D^s \in \{D^s_1, ..., D^s_{4^L}\}$ do                        ▷ Parallel Schedule
 5:     regional configurations $C^s \leftarrow$ Schedule($D^s$)
 6: end for
 7: for merge level $l = 1$ to $L$ do
 8:     divide $D^r$ into $4^{(L-l)}$ SDMs
 9:     for all major-region covered by $D^s \in \{D^s_1, ..., D^s_{4^{(L-l)}}\}$ do      ▷ Parallel merge
10:         $(C_0, C_1, C_2, C_3) \leftarrow$ configurations of 4 minor-regions within major-region
11:         major-region configurations $C^s \leftarrow$ Merge($C_0, C_1, C_2, C_3, D^s$)
12:     end for
13: end for
14: $C \leftarrow$ configurations of the one major-region on the L-th merge level
15: $f^{-r} \leftarrow$ repairing permutation s.t. $f^{-r}(f^r(i)) = i$
16: for all $c \in C$ do
17:     construct $p^{-r}$ s.t. $p^{-r}(i,j) = c.p(f^{-r}(i), f^{-r}(j))$      ▷ Repair from randomization
18:     $c.p \leftarrow p^{-r}$
19: end for
```

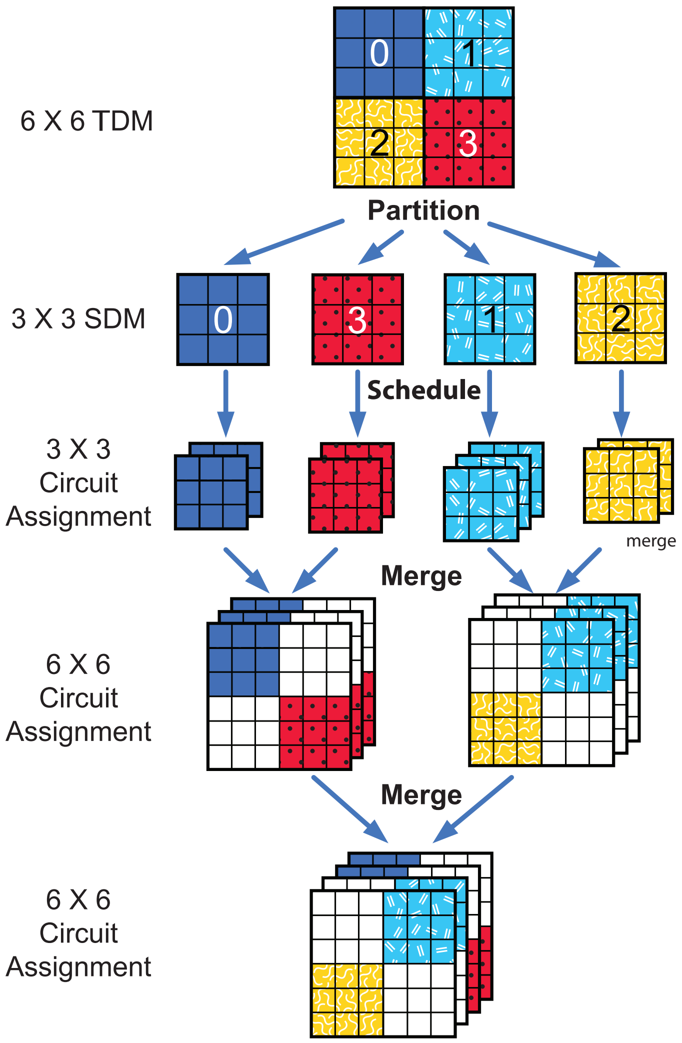

Figure 5.1 : An example illustrating 4-way Decomp for 6 racks.

5.2.1 Stage I: Partition

The schedule stage of Decomp can be parallelized in 4^L ways ($L \in \{1, 2, ...\}$). In 4^L-way Decomp over N racks, the inter-rack traffic demand is a $N \times N$ traffic demand matrix (TDM) D and then divided into 4^L regions, each in the size of $\frac{N}{2^L} \times \frac{N}{2^L}$(line 3). A regional demand matrix is called sub-demand matrix (SDM). For example, in 4-way Decomp over 6 racks, the 6×6 TDM for racks in random order is partitioned into four 3×3 SDM, as shown in the *Partition* step in Figure 5.1.

5.2.2 Stage II: Schedule

During the schedule stage, a list of slot configurations are computed for each region based on the regional demand SDM. Different from Edmond and TMS, each slot configuration in Decomp has an extra parameter, *weight*, which indicates the demand covered by the slot. Time duration for each slot is assigned proportionally to weight and the slot weights are further used in the merge stage of Decomp.

Parallel Schedule. Each SDM is fed into a subroutine to compute a list of slot configurations for the circuits covered by the SDM. Computation on each SDM can be run in parallel in different processes since there is no dependency among SDMs (line 4 in Algorithm 4). For example, in the *Schedule* stage of Figure 5.1, the SDMs are fed into four parallel processes.

Greedy Matrix Decomposition. The Schedule subroutine is described in Algorithm 5. We choose a light-weighted greedy algorithm as the schedule function, which iteratively decomposes SDM into multiple circuit assignments. In each iteration, we scan through the descending list of sorted entries in SDM (line 3), and add circuits for demand whenever the new circuit does not conflict with circuits already assigned in the same iteration (line 7 to line 12). The weight for each slot is set to the maximum demand covered by a slot's circuit assignment during the iterations of the greedy decomposition (line 6). We remove the demand entries with circuits assigned (line 10) and keep producing new circuit assignments until demand entries are drained (line 4).

We choose this greedy decomposition algorithm for several reasons. Firstly, unlike Edmond which uses the maximum weighted matching algorithm and generates

Algorithm 5 Subroutine: Schedule (Greedy Matrix Decomposition)

```
1: procedure SCHEDULE(demand matrix $D$)
2:     a list of slot configurations $C \leftarrow \emptyset$
3:     $D^L \leftarrow$ list of descending-sorted non-zero entries in $D$
4:     while $D^L$ not empty do                                  ▷ Greedy matrix decomposition
5:         circuit assignment $p \leftarrow \emptyset$
6:         weight $w \leftarrow$ largest entry in $D^L$
7:         for demand entry $d \in D^L$ do
8:             if the circuit from $d.src$ to $d.dst$ does not conflict with $p$ then
9:                 $p \leftarrow p+$ the circuit from $d.src$ to $d.dst$
10:                $D^L \leftarrow D^L - d$
11:            end if
12:        end for
13:        add non-conflicting circuits with non-zero demand in $D$ to $p$     ▷ Back Filling
14:        slot assigned circuits $c.p \leftarrow p$, weight $c.w \leftarrow w$
15:        append $c$ to $C$
16:    end while
17:    cutoff $\leftarrow \max_{p \in C.p} |p|$
18:    remove $c \in C$ if $|c.p| <$ cutoff                         ▷ Dynamic Pruning
19:    time share for $c \in C$ : $c.t \leftarrow c.w / \sum_{c \in C} c.w$
20:    return $C$
21: end procedure
```

only one circuit assignment per cycle, the greedy decomposition algorithm produces *multiple* circuit assignments based on demand, so that more valid circuits may be covered in each cycle. Secondly, the greedy decomposition algorithm has much lower computation overhead. For $\frac{N}{2^L} \times \frac{N}{2^L}$ SDM, the running time of the greedy algorithm is $O(n^3)$ with $n = \frac{N}{2^L}$ [19]. To produce multiple circuit assignments per cycle, one can also replace the greedy scanning (line 7 to line 12) with the maximum weighted matching algorithm, so that in each iteration, the maximum weighted matching algorithm produces a set of circuits based on the SDM. Then the SDM is updated by removing the the demand entries covered by the circuits in the maximum weighted matching, before the updated SDM is fed into the next iteration. However, using the maximum weighted matching algorithm instead of greedy scanning would incurs computation in $O(n^4)$. TMS also produces multiple circuit assignments per cycle, however, its computation complexity is even higher in $O(n^{4.5})$. The Schedule subroutine is a heavily used function so we pick a light-weighted greedy algorithm to avoid excessive delay in computing control logics. Thirdly, the greedy decomposition algorithm incurs less circuit reconfiguration overhead compared with TMS. For $\frac{N}{2^L} \times \frac{N}{2^L}$ SDM, the number of assignments generated per cycle by the greedy algorithm is in $O(n)$[19], compared with $O(n^2)$ under TMS. It is also worth mentioning that replacing the greedy scanning (line 7 to line 12) with the maximum weighted matching algorithm will not help to reduce the number of assignments generated per cycle from $O(n)$[19]. Fourthly, this greedy algorithm is efficient by grouping circuits with demand close to one another into the same assignment, which effectively reduce circuit idleness incurred when relatively small demand is drained in one assignment. The upper half of Figure 5.2 shows the results of the Greedy matrix decomposition algorithm on the example demand not partitioned.

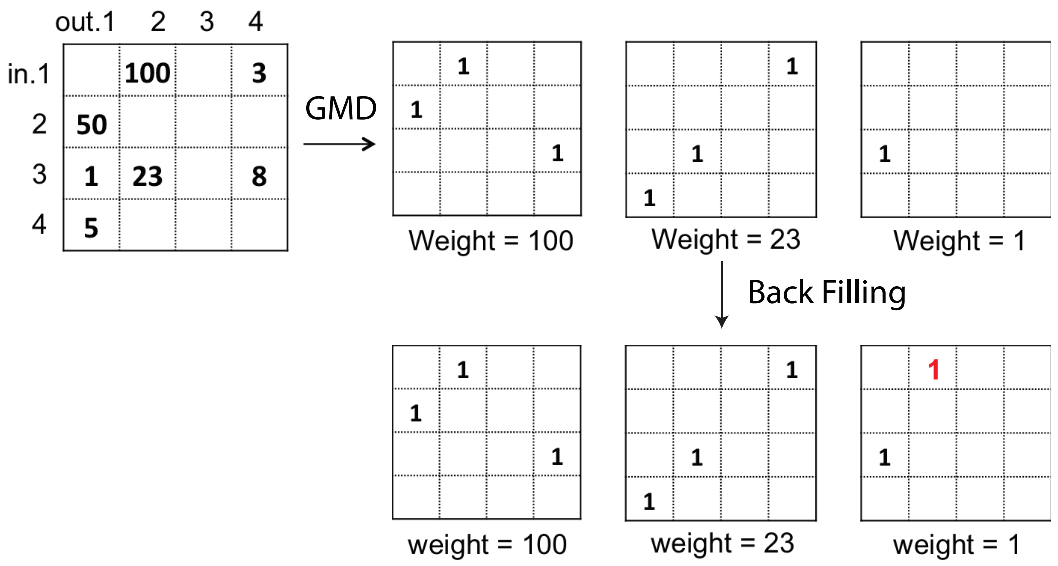

Figure 5.2 : Greedy matrix decomposition (GMD) algorithm on the example demand. Demand is not partitioned. Back filling circuits are marked red.

Back Filling Circuits. In the greedy matrix decomposition algorithm, demand entries are removed iteratively (line 10). Hence assignments produced in those later iterations might fail to cover some non-conflicting circuits that have non-zero demand. For example, in Figure 5.2, the circuit from input port 1 to output port 2 is not covered in the last circuit assignment produced. In an online system, letting resource idle may hurt performance. As a result, we propose to perform circuit back filling on all circuit assignments generated, by adding circuits that have non-zero demand in SDM as long as the circuits are not conflicting with circuits already included in the assignment (line 13). For example in Figure 5.2, we add an extra circuit from input port 1 to output port 2 to the last circuit assignment.

Dynamic Slot Pruning. We can dynamically prune out undesired slots to maintain high circuit utilization. Particularly, we choose to keep slots with maximum number of concurrent circuits and prune out the rest in order to with maximize circuit utilization(line 18). For example in Figure 5.2, the last slot is pruned because it has only 2 concurrent circuits, which is less than other slots with 3 circuits. Note that other pruning policy may also apply to satisfy different scheduling purposes. Finally, depending on slot weights, we scale the time share for the remaining slots to fill the whole scheduling cycle, so that slots covering larger demand are allocated more time share (line 19). For example in Figure 5.2, after pruning out the last slot, we may assign the first two slots with duration of 100/123 and 23/123 respectively.

5.2.3 Stage III: Merge

Taking Advantage of Non-conflicting Regions. After the schedule stage, we obtain regional circuit scheduling, a list of slot configurations for each region. However, we cannot apply these slots directly because the scheduled circuits in different slots might be conflicting with each other. Particularly, the ports covered by two *conflicting regions* are overlapped, and thus the circuits to be scheduled in these two regions might be conflicting if they required the same port. For example, in Figure 5.1, region 0 and region 1 are conflicting regions because they cover the same set of input ports. Nevertheless, *non-conflicting regions* (e.g. region 0 and region 3 in Figure 5.1) cover different subset of ports and thus slots in two non-conflicting regions can be scheduled at the same time.

The main task of merge is to compute a globally feasible circuit scheduling from the regional scheduling, while respecting the scheduling results on different regions. Instead of serializing all regional scheduling, we propose an efficient merge algorithm which takes advantage of the diagonal non-conflicting regions by first 1) merging scheduling on non-conflicting regions and then 2) serializing merged scheduling on conflicting regions. Particularly, we make use of *major regions* and *minor regions* so that a major region contains 4 minor regions in 2-by-2 layout. We obtain circuit scheduling for the major regions by merging slots from the diagonal non-conflicting minor regions. We merge recursively until the circuit scheduling that covers all circuit is obtained.

A 4^L-way Decomp with 4^L regions in the schedule stage has L merge levels (line 7 in Algorithm 4). The major regions and minor regions are recursively defined in different merge level. A *minor* region on one level is defined as the major region on the last lower level, with each of the 4^L regions in the schedule stage defined as the minor regions on the 1st merge level. A *major* region on level l is defined as the links covered by one of the $4^{(L-l)}$ SDMs, each in size of $\frac{N}{2^{L-l}} \times \frac{N}{2^{L-l}}$, divided from the randomized demand matrix D^r (line 9 in Algorithm 4). For example, in the *Merge* step of Figure 5.1, each of the 3×3 blocks is a minor region on the 1st merge level and the 6×6 block is a major region on the 1st merge level and a minor region on the 2nd merge level.

Computation of merging on different major regions can run in parallel (line 9 in Algorithm 4). Further more, merging configurations from minor regions on different diagonal can also run in parallel (details to be discussed in Section5.3.1). Note that on the L-th merge level, there is only one major region, and the configurations for this one major region cover all circuits (line 14 in Algorithm 4).

Merging Slots from Non-conflicting Regions. Merging slots from two non-conflicting regions is non-trivial: we must avoid excessive switching overhead but also need to make the most of regional scheduling results. On one hand, a cross product of the slots from both regions may result in excessive circuit switching overhead because the number of the resulting assignments would be a multiplication of the number of slots in both regions. On the other, merging slots from each region one by one would constraint the number of resulting circuit assignments. However, it is less desirable because the number of slots and their weights may differ vastly on different regions, and a slot from the region with more slots may fail to find a matched slot in another region. Besides, it is more favorable to merge two slots with similar weights so as to group circuits with similar demand into one assignment.

We propose an efficient way to merge slots from the non-conflicting regions, so that the merge stage may respect and make the most of regional scheduling decisions, as well as avoiding excessive scheduling overhead. Particularly, we allocate virtual time for each slots based on slot weights and combine two slots employed at the same virtual time on the non-conflicting minor regions into one slot on the major region (line 7 to line 15), with the weight for the combined slot set to the maximum weight among the two original slots (line 14). For example in Figure 5.3, the schedule stage generates two slots $C_0(0)$ and $C_0(1)$ for region 0 (blue) with virtual time share 0.8 and 0.2 respectively, and another two assignments $C_3(0)$ and $C_3(1)$ for region 3 (red) with virtual time shares of 0.5 and 0.5, then merging region 0 and 3 yields three configurations. The first one has a circuit assignment of $C(0).p = C_0(0).p \cup C_3(0).p$, the second $C(1).p = C_0(1).p \cup C_3(0).p$, and the third $C(2).p = C_0(1).p \cup C_3(1).p$.

The slots should be sorted descending on weight before merge (line 4). By sorting, we try to group circuits with similar weight into one slot and assign to the new slot a new weight that is close to the original ones, such that the new weight may better represent the demand covered by the new assignment.

Note that each combined slot may utilize one slot from both regions and only $O(n)$ new slots is produced on the resulting $2n$-by-$2n$ major region given $O(n)$ slots from the original n-by-n minor regions.

Serializing Slots from Conflicting Regions. The merged slots from conflict-

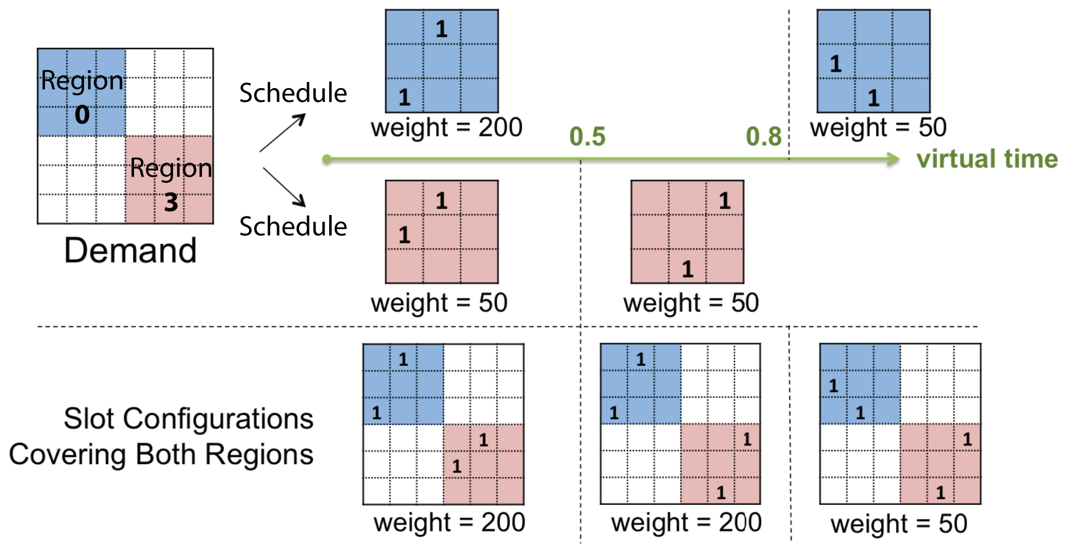

Figure 5.3 : Decomp merges slots from non-conflicting regions. Generate a new slot by combining two slots employed at the same virtual time in different regions.

ing regions, e.g. the merged slots from region 0 and 3, and the merged slots from region 1 and 2, might contain conflicting circuits, and therefore these slots are serialized to be the slots for the major region (line 23).

Optimizing with Back Filling and Dynamic Slot Pruning. Similar to schedule stage, the circuit assignments in the merged slots should be filled with non-conflicting valid circuits (line 16). Since the two circuit assignments to be merged has been filled in the schedule stage (or previous merges), extra circuits are less likely in the two merging regions. However, if the two merging assignments does not take up all input and output ports, more valid circuits might be added in the rest two regions not involved in the merge, by using the idle ports left after the two merging assignments. For example in Figure 5.4, in the first 3 merged circuit assignments from region 0 (blue) and 3 (red), extra circuits might exist in region 1 (cyan) and 2 (yellow). Besides, we can also further optimize the scheduling decisions by pruning out undesired slots to enforce high circuit utilization(line 22). For example in Figure 5.4,

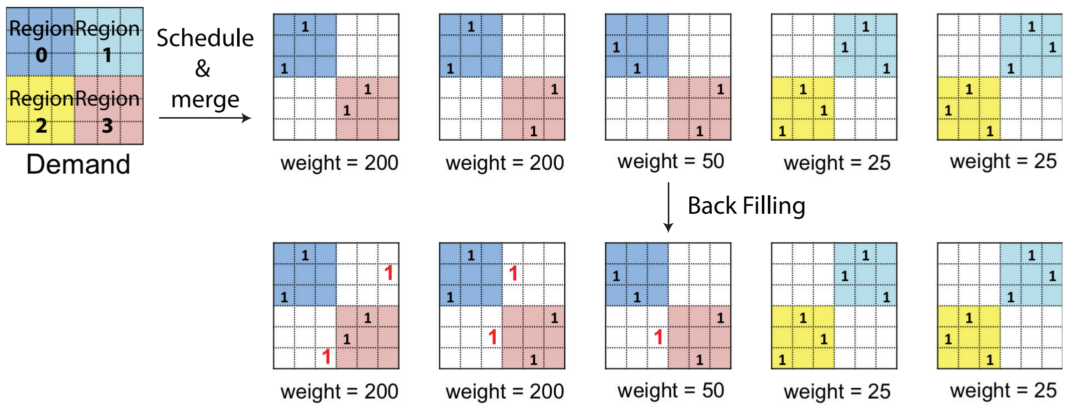

Figure 5.4 : Back filling extra non-conflicting valid circuits in the merge stage. Back filling circuits are marked red.

we can prune out the third slots since it only has 5 concurrent circuits, which is less than the rest slots which all have 6 circuits.

5.3 Decomp Algorithm Analysis

5.3.1 Process Usage

The schedule stage of 4^L-way Decomp takes 4^L processes for each minor-region in the first level. The merge stage in the l-th level uses $2 \times 4^{(L-l)}$ processes for merging minor-regions on two diagonals in each of the $4^{(L-l)}$ major regions. For example, in Figure 5.1, one process is used to merge region 0 and 3 and another one is used to merge region 1 and 2 in parallel.

5.3.2 Computation Complexity

In 4^L-way Decomp for N racks, each schedule process takes $O((\frac{N}{2^L})^2 \log \frac{N}{2^L})$ to sort the $(\frac{N}{2^L})^2$ entries in each SDM. The number of loops for picking up non-conflicting entries from the sorted list (line 4 of Algorithm 5) is $O(\frac{N}{2^L})$. In each iteration, the

Algorithm 6 Subroutine: Merge

```
1: procedure MERGE(four minor-regions configurations $C_0, C_1, C_2, C_3$, major-region demand $D$)
2:     major region configurations $C \leftarrow \emptyset$
3:     for all minor-regions $(i_1, i_2)$ on one diagonal $((0,3)$ or $(1,2))$ do
4:         Apply descending sort on slots from both regions according to weight
5:         index of configurations in $C_{i_1}, C_{i_2} : j_1, j_2 \leftarrow 0$
6:         while $j_1 < \left|P^k_{i_1}\right|$ or $j_2 < \left|P^k_{i_2}\right|$ do
7:             if $C_{i_1}(j_1)$ ends before $C_{i_2}(j_2)$ then
8:                 $j_1 \leftarrow j_1 + 1$
9:             else if $C_{i_1}(j_1)$ ends after $C_{i_2}(j_2)$ then
10:                $j_2 \leftarrow j_2 + 1$
11:            else if $C_{i_1}(j_1)$, $C_{i_2}(j_2)$ end simultaneously then
12:                $j_1 \leftarrow j_1 + 1$, $j_2 \leftarrow j_2 + 1$
13:            end if
14:            weight $w = \max(C_{i_1}(j_1).w, C_{i_2}(j_2).w)$
15:            assignment $p \leftarrow C_{i_1}(j_1).p \cup C_{i_2}(j_2).p$
16:            add non-conflicting circuits with non-zero demand in $D$ to $p$ ▷ Back Filling
17:            slot assigned circuit $c.p \leftarrow p$, weight $c.w \leftarrow w$
18:            append $c$ to $C$
19:        end while
20:    end for
21:    cutoff $\leftarrow \max_{p \in C.p} |p|$
22:    remove $c \in C$ if $|c.p| <$ cutoff                ▷ Dynamic Pruning
23:    time share for $c \in C : c.t \leftarrow c.w / \sum_{c \in C} c.w$
24:    return $C$
25: end procedure
```

sorted list is scanned and its size decreases. Thus, the loop takes $O((\frac{N}{2^L})^3)$, resulting $O((\frac{N}{2^L})^3)$ for the schedule stage.

In the l-th level of merge, the maximum number of loops (line 6 in Algorithm 6) is the total number of assignments in two diagonal minor-regions, or $O(\frac{N}{2^{L-l+1}})$. Thus merges take $O((\frac{N}{2}) - (\frac{N}{2^{(L+1)}}))$ in total. This is a loose estimation and merge has a much lower complexity in practice. We observe that the the computation time on the merge stage is a very small compared with that on the schedule stage. The bottle neck of Decomp is thus its schedule stage in $O((\frac{N}{2^L})^3)$.

5.3.3 Circuit Configurations

The number of circuit configurations for each scheduling cycle is subject to the algorithm used in the schedule stage to decompose each SDM. The greedy matrix decomposition algorithm used by Decomp is shown [20] to produce at most $2\frac{N}{2^L} - 1$ circuit assignments for $\frac{N}{2^L} \times \frac{N}{2^L}$ SDM. In the 1st merge level, each minor region has no more than $2\frac{N}{2^L} - 1$ circuit assignments. Thus merging four minor regions produces no more than $4 \times (2\frac{N}{2^L} - 1)$ circuit assignments, with $2 \times (2\frac{N}{2^L} - 1)$ from merging minor regions on each of the two diagonals. Thus each minor region on 2nd merge level has no more than $2\frac{N}{2^{(L-2)}}$ circuit assignments. In the l-th level of merge, assume each minor region has no more than $2\frac{N}{2^{(L-2l+2)}}$ circuit assignments. Thus on the $(l+1)$-th merge level, each minor has no more than $2\frac{N}{2^{(L-2(l+1)+2)}}$ circuit assignments. By induction, we know that on the final merge level, the total number of circuit assignments is less than $2^{(L+1)}N$ with L as a constant. Thus the number of circuit configurations for Decomp is in $O(N)$ for N racks.

	Computation Complexity	**Slots/Cycle**
4^L**-way Decomp**	$O((\frac{N}{2^L})^3) + O((\frac{N}{2}) - (\frac{N}{2^{(L+1)}}))$	$O(N)$
Edmond	$O(N^3)$	1
TMS	$O(N^{4.5})$	$O(N^2)$

Table 5.1 : Comparison of scheduling algorithms on N racks. (Decomp is the only algorithm that is parallelizable)

5.3.4 Comparison with Existing Algorithms

Table 5.3.4 shows a comparison of 4^L-way Decomp, Edmond and TMS on N racks. Although Decomp is a polynomial-time algorithm like Edmond and TMS, Decomp is the only algorithm among the three that is parallelizable. As we will show in Section 6.2.2, Decomp is much faster than Edmond and TMS.

Besides, Decomp applies a modest amount of circuit switchings per scheduling cycle. On one hand, unlike Edmond which rejects all circuits outside the maximum weighted matching set, Decomp schedules multiple circuit assignments per cycle, so as to cover traffic demand in each cycle and mitigate starvation problem as in Edmond. On the other hand, Decomp maintains network utilization by bounding the number of circuit configurations per cycle to O(N), instead of O(N^2) in TMS, to allow for better scalability.

Chapter 6

Evaluation

6.1 Methodology

We study the performance of different scheduling algorithms under the same settings, i.e. topology, bandwidth allocation, circuit switching delay, etc.

6.1.1 Network Topology and Link Bandwidth

We simulate a network consisting of 16 racks, with 20 servers per rack. In the hybrid architecture, each top of rack (TOR) switch both connects to a 16×16 optical switch and an electrical Ethernet packet switch. In the pure architecture, each TOR switch *only* connects to the optical switch. In the hybrid architecture, the electrical bandwidth B_e varies from 10 Mbps to 1 Gbps in the experiments. The bandwidth of optical links in both architectures is 10 Gbps. The links between TOR and server are 10 Gbps. Thus each server can saturate either the optical or electrical links in the core network with its own traffic.

6.1.2 Metrics

We use three metrics, i.e. traffic finish time, flow waiting time and algorithm computation time to evaluate the scheduling algorithms. These metrics describes the network resource utilization and traffic delay under a scheduling algorithm, as well as how responsive a scheduling algorithm can be in case of traffic change.

- Traffic Finish Time T_F

  Consider a set of flows F in a certain traffic pattern. A flow $f \in F$ arrives at $s(f)$ and finishes at $e(f)$. The traffic finish time is the time when all flows finish, i.e.

  $$T_F = \max_{f \in F} e(f). \tag{6.1}$$

  The traffic finish time reveals the network utilization when all flows arrive at time 0 and wait to be serviced. To clear the same set of flows, the shorter the traffic finish time is, the higher the utilization.

- Flow Waiting Time $t_W(f)$ and T_W

  The waiting time of a flow $f \in F$ is defined as the length of time from its arriving time to its finishing time, i.e.

  $$t_W(f) = e(f) - s(f). \tag{6.2}$$

  Summing the waiting time of all flows gives the total waiting time T_W for flow set F.

  $$T_W = \sum_{f \in F} t_W(f) \tag{6.3}$$

  $t_W(f)$ measures the responsiveness experienced by individual flow, exposing the impact on short and long flows, while T_W measures overall responsiveness. The waiting time reflects how long a flow has to wait before it goes through the network. For applications blocked to wait for data from another end host, the waiting time determines how fast they are able to respond. Long total waiting time implies overall slow response of applications running on the data center.

- Computation Time

The computation time affects how a scheduling algorithm can be adaptive to network changes, and therefore it should be computational efficient. The computation time of the scheduling algorithm is a function of network size and traffic pattern.

6.1.3 Simulation Settings

Algorithms are called at the beginning of a scheduling cycle. Edmond generates only one circuit assignment for each cycle consisting of one slot. Decomp and TMS can generate multiple circuit assignments corresponding to multiple slots within one cycle. In our experiment, the circuit switching delay is 10 μs, which is typical for optical wavelength-selective switches (WSS). The scheduling cycle is 0.1 second to allow for a fair trade-off between optical circuit utilization and fast adaptivity to traffic change.

6.1.4 Traffic Workloads

In data centers, traffic demand fluctuates over time, resulting in various traffic patterns, uniform or skewed, dense or sparse. At one time, traffic might be distributed uniformly across racks but at another, hotspots show up if a few racks have significantly larger demand. Besides, the traffic is said to be dense when most racks have traffic demands for each other, while the traffic is sparse when only some racks have traffic demands for each other. We study the performance of scheduling algorithms under these four combinations of patterns.

- *Urand (uniform-dense):* Each server sends a flow of 125 MB to another N randomly chosen servers, where N is the number of racks. In a data center with N racks and M servers per rack, this implies that each rack has NM inward and

NM outward flows on average. Urand is a standard pattern used in existing work [3, 4, 5] to simulate intensive communications in data centers.

- *Stride (uniform-sparse):* The servers are indexed from 0 to 320. Server i sends a flow of 125 MB to each of 8 other servers in 8 different neighbor racks, which are server $(i + 20 * j + 1) \bmod 20$, with $j \in \{1, ..., 8\}$. In Stride, each rack has traffic to half of all TORs. Stride is another standard pattern used in existing work [3].

- *Hotspots (skewed-dense):* Hotspot is a mixture of heavy flows, each in size of 10 GB between 10 pairs of randomly chosen servers, and small flows of uniform random dense traffic of 100 KB each. We include it because an extensive measurement study of data center traffic [21] reports that hotspots are common.

- *Isolated (skewed-sparse):* Three non-conflicting isolated flows, each in size of 100 MB, coexist in a 16×16 traffic matrix. In this traffic pattern, all flows can be routed by the optical switch at the same time. The reason we include it is to demonstrate certain deficiency of the TMS algorithm on sparse and skewed traffic in Section 6.2.4.

6.2 Numerical Results

6.2.1 Decomp achieves low flow waiting time while preserving network utilization

In this experiment, we zero out the algorithm computation time and circuit switching time in order to remove the influence from these two factors on performance. In other words, we want to show how each algorithm assigns circuits for the same set of flows.

In practice, the performance would also be impacted by algorithm computation time and circuit reconfiguration delay.

Figure 6.1 shows the distribution of Hotspot's flow waiting time under a 10 Gbps pure optical architecture. We make the following observations. Firstly, flows tend to experience long waiting time under Edmond in the pure optical architecture. More than 50% of the flows are still alive after 5 seconds under Edmond, while with the same time, both TMS and Decomp have already cleared more than 89% of the flows. With an objective of maximizing the total traffic served by each circuit assignment, Edmond is more likely to prioritize large flows into optical paths, leaving small ones waiting until their sizes are comparable with what is remained of the large flows. Without assistance of electrical network which allows small flows to pass quickly, the small flows suffer under Edmond.

Secondly, flow waiting time is significantly improved with Decomp. Both Decomp and TMS allow bandwidth sharing among flows with several different circuit assignments in each scheduling cycle. Thus even small flows can take advantage of optical circuits assigned in short slots. Furthermore, Decomp has even better flow waiting time than TMS. Decomp partitions traffic demand and schedules each partition separately, so that the large flows can only dominate a single partition rather than the entire network. In TMS, heavy demand can dominate the use of optical network more easily given that TMS schedules all circuits over the entire traffic demand matrix. Experiments show that by 0.13 second, Decomp has cleared more than 97% of the flows, while the number for TMS and Edmond is merely 84% and 42% respectively. We also observe that Decomp reduces the average per flow waiting time by more than 0.14 second for Stride pattern and 0.78 second for Urand pattern compared with TMS and Edmond under the same network settings.

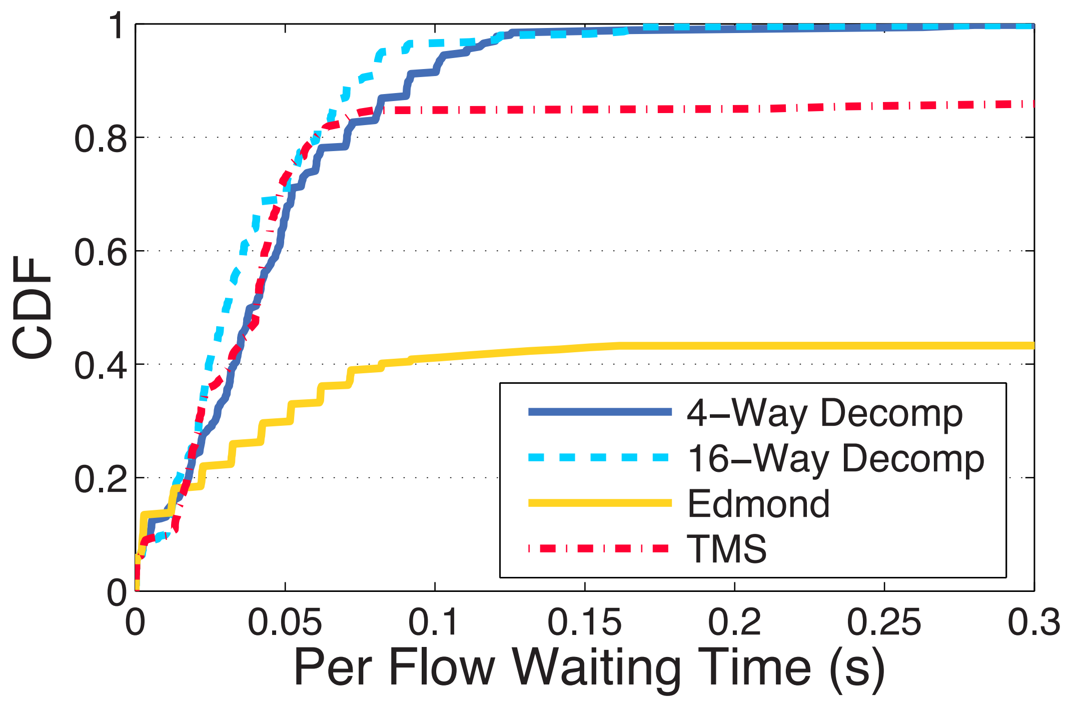

Figure 6.1 : CDF of flow waiting time for the Hotspot pattern under the 10 Gbps pure optical network architecture. Decomp is suitable for the pure optical network architecture and eliminates the long-tailed flow waiting times that Edmond and TMS suffer from.

Furthermore, we present the traffic finish time for Hotspot, Urand and Stride under the pure architecture in Figure 6.2. Decomp achieves similar network utilization as Edmond and TMS in that Decomp clears traffic with roughly the same traffic finish time as the other two algorithms. But with comparable network utilization, Decomp is much better at allocating bandwidth among flows to reduce flow waiting time.

Next, we extend our study to the hybrid network. Figure 6.3 shows the total flow waiting time for the Hotspot pattern under the hybrid architecture with different ratios of electrical and optical bandwidth. Decomp has short waiting time over all hybrid network settings. In contrast, flows still suffer from long waiting time in Edmond when there is insufficient electrical bandwidth. Like Edmond, TMS also

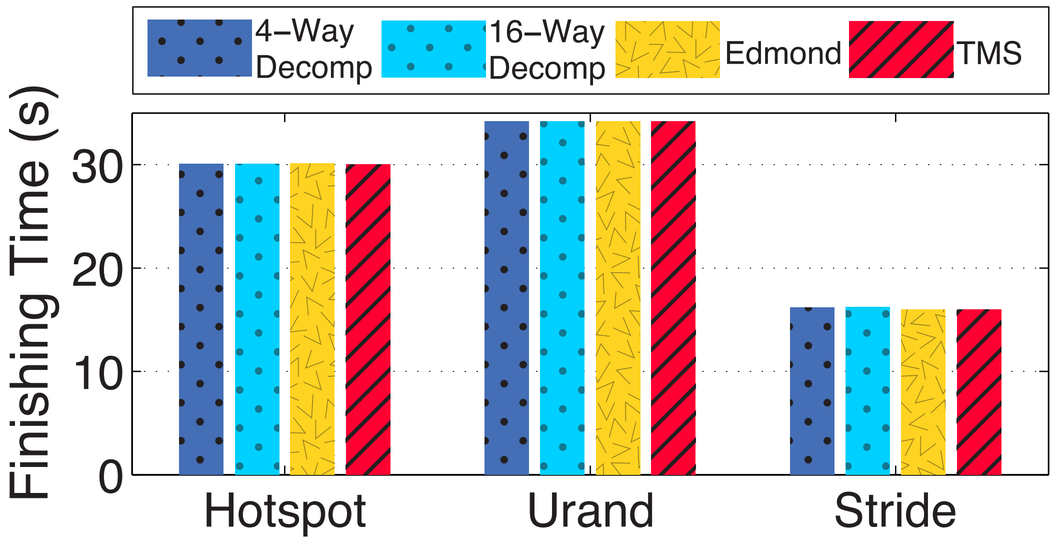

Figure 6.2 : Traffic finish time under the 10 Gbps pure optical architecture. Decomp is as good as other algorithms in terms of network utilization regardless of traffic density, while simultaneously eliminating long-tailed flow waiting times.

suffers from not having enough electrical bandwidth, though it is more tolerant. In the hybrid network with as much as 100 Mbps electrical bandwidth, the total flow waiting time for Decomp is less than 44% of TMS and only 4.6% of Edmond. When the electrical bandwidth drops further down to 10 Mbps, the total flow waiting time for Decomp is only 9.3% of TMS and 1.2% of Edmond. The starvation problem of small flows under TMS and Edmond is mitigated only after we add a considerable amount of electrical bandwidth for the small flows to go through easily without significant delay.

6.2.2 Decomp requires far less computation time than Edmond and TMS

We measure the execution time of the scheduling algorithms on a 3.10 GHz Intel dual Core i3 processor with 3.7 GB memory. The computation time is not memory bounded in that the maximum memory consumption is only 1521 MB for 16-way

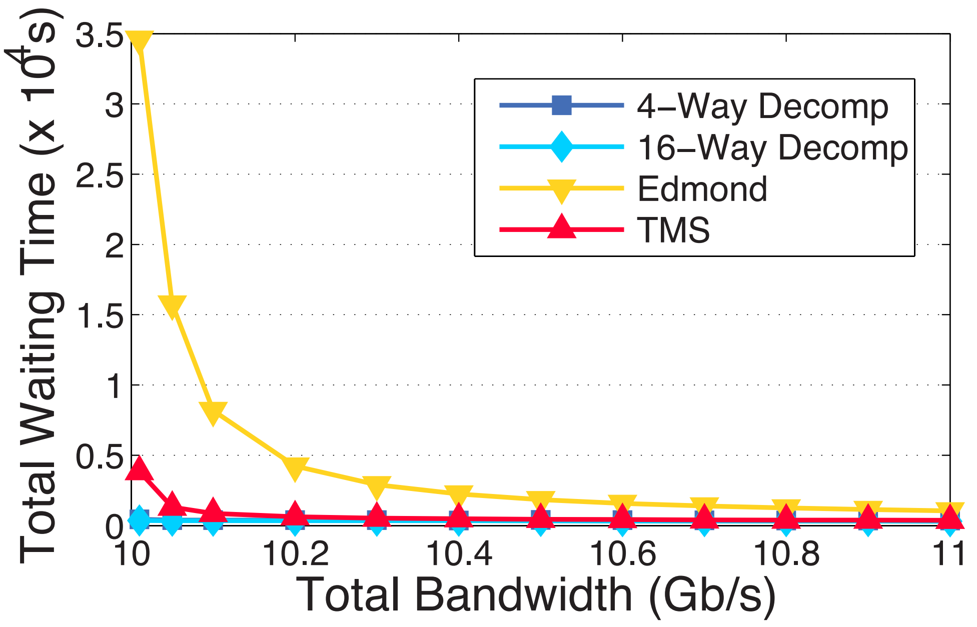

Figure 6.3 : Total flow waiting time for Hotspot under the hybrid network architecture (Total bandwidth is the sum of 10 Gbps constant optical bandwidth and variable electrical bandwidth). Decomp is suitable for the hybrid network architecture and achieves the lowest flow waiting times regardless of the amount of electrical network bandwidth available.

Decomp over 600 racks. All algorithms are implemented in the C language for best performance. We have ensured that Edmond and TMS are well optimized. The computation time of Decomp is measured as the total time spent on schedule and merge. The time spent on the schedule phase is the maximum time spent on calculating each SDM. The time spent on merges in different levels is the maximum time spent on each major-region, and the time for the whole merge phase is the total time spent on merges in each level.

Figure 6.4 shows the computation time for Urand traffic matrices in size of up to 600×600. We observe that Decomp takes much less computation time than Edmond and TMS. 16-way Decomp is $28000\times$ faster than TMS and $170\times$ faster than Edmond

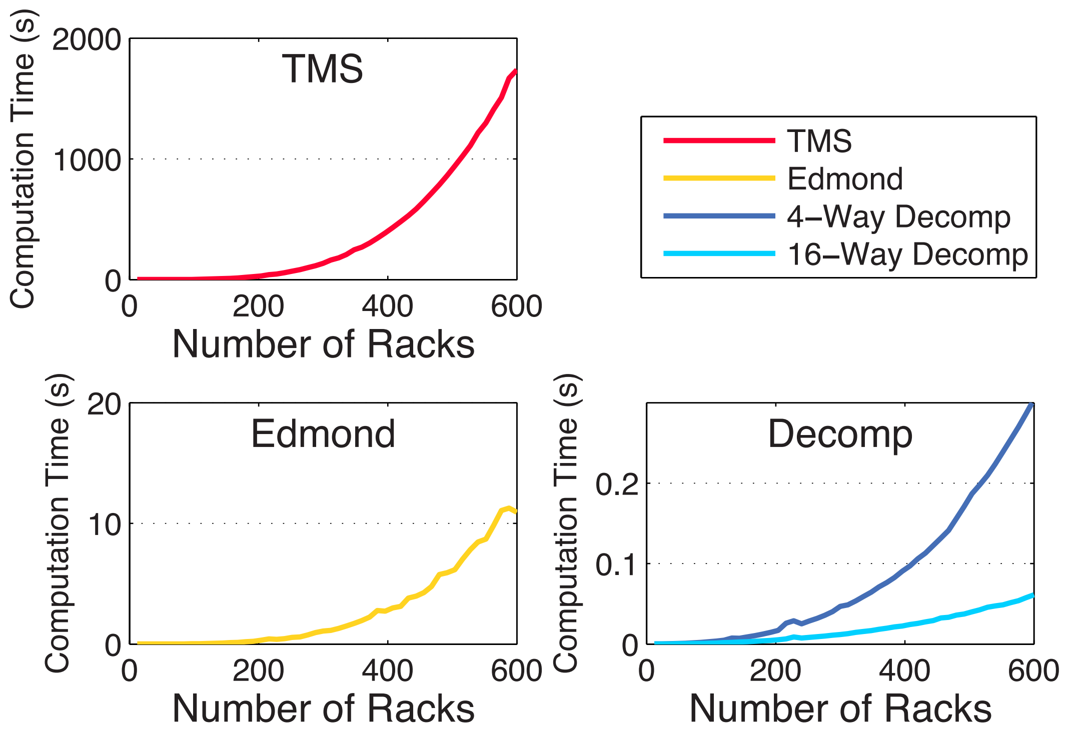

Figure 6.4 : Computation time for the dense Urand pattern under different algorithms. Note that the y-axes have very different scales, and Decomp is orders of magnitude more computationally efficient. Parallelization allows Decomp to scale well to large network sizes. While not shown in this figure, Decomp is also much more efficient for sparse traffic (see Section 6.2.2).

when the topology scales to 600 racks. Under the sparse Stride pattern (graphs omitted due to space constraint), the computation time of Decomp and TMS drops significantly because a sparse matrix can be easily decomposed into a small number of assignments, reducing the computation complexity of the Birkhoff decomposition [15] step in TMS and the schedule/merge in Decomp. However, Decomp is still much faster. Under Stride on average across the different network sizes, 4-way and 16-way Decomp speeds up computation by 9× and 42× respectively compared with TMS, while 4-way and 16-way Decomp are 92× and 426× faster than Edmond respectively.

A valid algorithm should adapt quickly to different traffic. For dense traffic, it will take thousands of seconds for TMS to respond in a data center with 600 racks, which is unacceptable. In other words, if the traffic in data centers experiences dramatic change in a short time and the scheduling algorithm is required to respond and adapt within, say 0.05 second, then the topological size could not go beyond 48 racks with TMS, or 132 racks with Edmond. In contrast, Decomp can leverage modern multi-core processors. Under the same responsiveness requirement, 4-way Decomp can support up to 312 racks and 16-way Decomp can support up to 552 racks.

6.2.3 Decomp generates far fewer slots than TMS

Unlike Edmond which generates only one circuit assignment at a time, Decomp and TMS schedule multiple circuit assignments or slots after each run. Having a larger number of slots per scheduling cycle leads to lower efficiency because more time is wasted on reconfiguration circuits. Figure 6.5 shows the number of slots, or circuit assignments, generated by Decomp and TMS for Urand traffic matrices in size of up to 600×600. We find that the number of slots generated by Decomp is $O(N)$ for N racks, much less than $O(N^2)$ of TMS. Under Stride, the number of slot drops compared with Urand both for Decomp and TMS because sparse matrix can be easily decomposed into less assignments, but still Decomp generates at least 35% fewer slots than TMS on average across the different network sizes. While pruning out small slots might seem a plausible approach to improve utilization for TMS, however, doing so many inadvertently harm small flows.

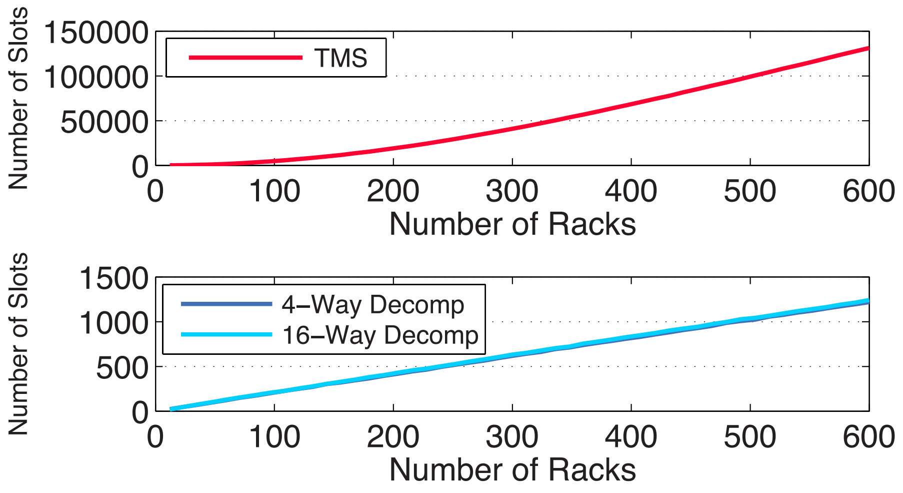

Figure 6.5 : Number of slots for the Urand pattern under Decomp and TMS. Decomp generates two orders of magnitude fewer scheduling slots, leading to much lower circuit switching overhead. Recall that switching an optical circuit takes up to tens of milliseconds.

6.2.4 TMS makes inefficient scheduling decisions for Isolated under combined effect of computation and reconfiguration overhead

TMS is studied only with dense traffic in previous works [6, 4, 5]. Instead we subject TMS to sparse and skewed matrices in the Isolated pattern. The Sinkhorn algorithm [22], a building block of TMS, takes in matrices with *strictly positive* elements. TMS does not work when it is given a matrix with zero entries. A work-around solution is to substitute all zeros with a small quantity σ before feeding the matrix to Sinkhorn. In this way, the σ entries can be scaled to fit into a doubly stochastic matrix. Otherwise, the zeros persist all the way through the iterative scaling of columns and rows in Sinkhorn, preventing the algorithm from converging.

However, the substitutions will also bring in distortion of the traffic demand and long computation time for TMS. Let's consider a simple example such as the Isolated pattern. Ideally Sinkhorn will converge into a permutation matrix for Isolated, with entries of one covering the original traffic demand. But convergence to a strictly doubly stochastic matrix is time consuming and a more common practice is to terminate Sinkhorn when the error is considered *tolerable.* Nevertheless, it still takes a long time to scale rows and columns of substituted σ to below the error tolerance. Besides, by allowing error tolerance, Sinkorn leaves a lot of small entries in the resulting doubly stochastic matrix, but most of the links corresponding to the small entries have no traffic demand at all. When given this doubly stochastic matrix with error, the Birkhoff decomposition step in TMS generates a lot of circuit assignments corresponding to the non-existing demand. Although the distorted assignments tend to have small time slot durations, setting up these unnecessary circuits is a large waste given that unnecessary optical circuit reconfigurations waste network resources. Besides, Birkhoff takes long time to generate these assignments.

The authors in [4] suggest pruning out circuit assignments with small time share after Birkhoff by scheduling only a few longest slots. This approach will introduce a predefined threshold to decide which slots are to be taken away. But short slots are not necessarily a waste, but they might just represent traffic with small demand. A predefined threshold does not distinguish slots generated due to distortion or small demand.

The long execution time under Isolated implies that TMS cannot respond to traffic change quickly. One can try to enforce a short scheduling cycle with the risk that the last slot in current cycle ends before computation for next cycle finishes. To accommodate long computation in short cycle, one can (fix 1) extend the last slot until computation finishes. But the last slot might be a wasteful one generated due to distortion. Instead, one can (fix 2) always schedule the last slot as the one with

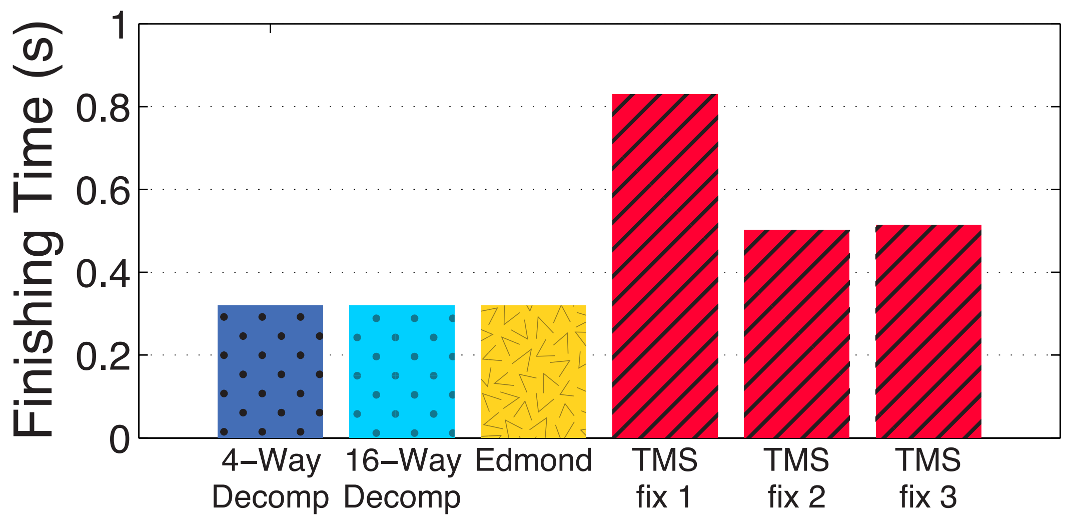

Figure 6.6 : Finish time of the Isolated traffic pattern under the 10 Gbps pure optical architecture. TMS's inefficiencies are rooted in its high computation and circuit reconfiguration overhead.

the longest time share because it is more likely to represent valid demand. In this case, the longest slot is extended disproportionally. There could be a lot of bandwidth wasted if the traffic is cleared before computation ends, while traffic represented in other slots is left unserved. To avoid this problem, one can also try to (fix 3) repeat the assignments in the current cycle during computation.

In Figure 6.6, we show the traffic finishing time of different algorithms under the Isolated pattern. In this experiment, two Isolated traffic patterns (I_1 and I_2) are used. Every 8 ms, a new set of traffic that alternates between I_1 and I_2 is injected. This aims to test how the algorithms adapt to traffic changes. Besides, computation time and circuit switching time are accounted for in this experiment. As shown in Figure 6.6, the traffic finishing time of TMS with fix 2 and 3 is 1.5× longer than Decomp and Edmond. This is because whenever a new set of traffic arrives at time t, TMS takes tens of milliseconds to compute a new set of circuit assignments and during this time, traffic is being served by inefficient assignments computed based on the traffic volumes prior to time t. In addition to this slow adaptation problem, TMS with fix 1 is even worse because it extends wasteful slots during computation, resulting in 2.5× longer traffic finishing time than Decomp and Edmond.

Chapter 7

Conclusion

We make two contributions in this thesis. First, we explore the challenges of circuit scheduling for next-generation optical data centers in three dimensions, i.e. handle hybrid and pure architectures, robustness under sparse and dense traffic patterns, and scalability. As a result, the weaknesses of the existing optical data center circuit scheduling algorithms are exposed. Secondly, we propose Decomp, which provides a framework that can be customized with different circuit selection policies and incorporates partitioning, randomization, and parallelization approaches that are not found in existing algorithms, and we show that it significantly outperforms the existing algorithms along all three dimensions.

Bibliography

[1] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan, “c-Through: Part-time Optics in Data Centers,” in *SIGCOMM ’10*, (New Delhi, India), p. 327, Aug. 2010.

[2] 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,” in *SIGCOMM ’10*, (New Delhi, India), p. 339, Aug. 2010.

[3] 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,” 2012.

[4] N. Farrington, G. Porter, Y. Fainman, G. Papen, and A. Vahdat, “Hunting Mice with Microsecond Circuit Switches,” in *ACM HotNets*, (Redmond, WA, USA), oct 2012.

[5] 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,” in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pp. 447–458, ACM, 2013.

[6] 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 *Proceedings of the 11th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA*, 2014.

[7] Calient, “Software defined packet-optical datacenter networks,” accessed July, 2014.

[8] Corning, “Fiber optic solutions for data centers and sans,” accessed July, 2014.

[9] M. Chowdhury and I. Stoica, “Coflow: A Networking Abstraction for Cluster Applications,” in *Hotnets 12*, (Seattle, WA, USA), pp. 31–36, Oct. 2012.

[10] J. Edmonds, “Paths, trees, and flowers,” *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, Jan. 1965.

[11] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, “High-speed switch scheduling for local-area networks,” *ACM Transactions on Computer Systems (TOCS)*, vol. 11, no. 4, pp. 319–352, 1993.

[12] N. McKeown, P. Varaiya, and J. Walrand, “Scheduling cells in an input-queued switch,” *Electronics Letters*, vol. 29, no. 25, pp. 2174–2175, 1993.

[13] P. Giaccone, B. Prabhakar, and D. Shah, “Towards simple, high-performance schedulers for high-aggregate bandwidth switches,” in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, pp. 1160–1169, IEEE, 2002.

[14] G. Birkhoff, “Tres observaciones sobre el algebra lineal,” *Univ. Nac. Tucumán Rev. Ser. A*, vol. 5, pp. 147–151, 1946.

[15] 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*, pp. 79–86, IEEE, 1999.

[16] T. Inukai, "An efficient ss/tdma time slot assignment algorithm," *Communications, IEEE Transactions on*, vol. 27, no. 10, pp. 1449–1455, 1979.

[17] E. Balas and P. R. Landweer, "Traffic assignment in communication satellites," *Operations Research Letters*, vol. 2, no. 4, pp. 141–147, 1983.

[18] J. L. Lewandowski, J. W. Liu, and C. Liu, "Ss/tdma time slot assignment with restricted switching modes," in *NTC'81; National Telecommunications Conference, Volume 3*, vol. 3, p. 7, 1981.

[19] A. Kesselman and K. Kogan, "Nonpreemptive scheduling of optical switches," *Communications, IEEE Transactions on*, vol. 55, no. 6, pp. 1212–1219, 2007.

[20] I. Keslassy, M. Kodialam, T. Lakshman, and D. Stiliadis, "On guaranteed smooth scheduling for input-queued switches," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 2, pp. 1384–1394, IEEE, 2003.

[21] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The Nature of Data Center Traffic: Measurements and Analysis," in *IMC '09*, (Chicago, Illinois, USA), pp. 202–208, Nov. 2009.

[22] R. Sinkhorn *et al.*, "A relationship between arbitrary positive matrices and doubly stochastic matrices," *The annals of mathematical statistics*, vol. 35, no. 2, pp. 876–879, 1964.