CROSSBAR SCHEDULING ALGORITHMS FOR INPUT-QUEUED SWITCHES

A Dissertation
Presented to
The Academic Faculty

By

Long Gong

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

Georgia Institute of Technology

August 2020

CROSSBAR SCHEDULING ALGORITHMS FOR INPUT-QUEUED SWITCHES

Approved by:

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

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

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

Dr. Siva Theja Maguluri
School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Bill Lin
Department of Computer Science and Engineering
University of California, San Diego

Date Approved: April 30, 2020

ACKNOWLEDGEMENTS

I would like to express my thanks to my advisor, Jun (Jim) Xu, for his kind guidance and persistent support during my Ph.D. study. I would also like to give my thanks to other members of my committee for their teaching, and their interests in my research work and their helpful suggestions and valuable comments: Mostafa H. Ammar, Ellen W. Zegura, Siva Theja Maguluri, and Bill Lin.

I would like to thank Yi Xie (Xiamen University), Xinbing Wang (Shanghai Jiao Tong University), and Paul Tune (University of Adelaide) for their guidance and support on our collaborated research works.

I would also like to extend my special gratitude to senior members in our research group, namely Sen Yang and Liang Liu, for their enormous help in both my academic research and daily life. In addition to them, I would also like to express my sincere gratitude to current and former lab members, Lanxi Huang, Tarun Mangla, Yimeng Zhao, Huayi Wang, Jingfan Meng, and many others, and lab visiting scholars, Shenglin Zhang (Tsinghua University), Jianyuan Lu (Tsinghua University), Pin Yin (UCSD), Ziheng Liu (Peking University), Minghua Ma (Tsinghua University), and all others. Thank them for the helpful research conversions and enriching my life at Georgia Tech. My gratitude also goes to my friends, Jingfan Sun, Bichen Zhang, Qiang Hu, Xinyuan Nan, Hongnan Lin, Haolin Zhang, and many others, for making my life more enjoyable.

I give my deepest gratitude to my family, especially my mother, Xiyun Long; and my father, Xinming Gong for their support and great love. Without them, none of this work would be possible.

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Many of today's switches and routers adopt an input-queued crossbar architecture to interconnect the input ports with the output ports. Such a switch needs to compute a crossbar schedule, or a matching, between the input ports and the output ports during each switching cycle, or time slot. A key research challenge in designing large (in number of input/output ports N) input-queued crossbar switches is to develop crossbar scheduling algorithms that can compute high-quality matchings – *i.e.,* those that result in high switch throughput (ideally 100%) and low queueing delays for packets – yet have a very low time complexity to support high link speeds. Indeed, there appears to be a fundamental tradeoff between the time complexity of the crossbar scheduling algorithm and the quality of the computed matchings (crossbar schedules).

This dissertation research consists of two aspects. The first aspect is to investigate crossbar scheduling algorithms that are low in time complexities (preferably $O(1)$ and definitely no more than $O(\log N)$ per port), yet have excellent throughput (ideally equal or close to 100%) and delay performances. The second aspect is to analyze the throughput and the delay performance guarantees of some of the proposed algorithms using Lyapunov stability analysis techniques.

Along the first aspect, we have proposed four algorithms. The first algorithm, called SERENADE (SERENA, the Distributed Edition), is a parallel iterative algorithm that can provably, with a time complexity of only $O(\log N)$ per port, exactly emulate SERENA, a centralized algorithm with $O(N)$ time complexity, which can attain 100% throughput and acceptable delay performance. The second algorithm, called Queue-Proportional Sampling (QPS), is an "add-on" approach that generates superior starter matchings than all other known strategies, yet incurs only $O(1)$ additional time complexity at each input/output port. We use QPS to augment two existing crossbar scheduling algorithms, namely SERENA and iSLIP. The augmented algorithms, which we call QPS-SERENA and QPS-iSLIP,

outperform the original algorithms by a wide margin, under various load conditions and traffic patterns. Building upon QPS, we propose the third algorithm, which we call QPS-r, a parallel iterative crossbar scheduling algorithm with $O(1)$ time complexity per port. We have shown that QPS-3 (r=3 iterations) has comparable empirical throughput and delay performances as maximal matching algorithms that have much higher time complexities. The last algorithm, call Small-Batch QPS (SB-QPS), is a batch (crossbar) scheduling algorithm that builds upon and significantly improves the throughput performance of QPS-r, yet has a time complexity of $O(1)$ per port (per time slot).

Along the second aspect, we have proved, using Lyapunov stability analysis, that QPS-SERENA can achieve 100% throughput and that using matchings generated by QPS-r (even when r=1) as crossbar schedules results in at least 50% switch throughput and order-optimal (*i.e.,* independent of the switch size N) average delay bounds for various traffic arrival processes.

CHAPTER 1
INTRODUCTION

1.1 Motivation

The volume of network traffic across the Internet and in data-centers continues to grow relentlessly, thanks to existing and emerging data-intensive applications, such as big data analytics, cloud computing, and video streaming. At the same time, the number of network-connected devices is exploding, fueled by the wide adoption of smart phones and the emergence of the Internet of things. To transport and "direct" this massive amount of traffic to their respective destinations, switches and routers capable of connecting a large number of ports (called *high-radix* [1, 2, 3]) and operating at very high line rates are badly needed.

Many present day high-performance switching systems in Internet routers and data-center switches employ an input-queued crossbar to interconnect their input ports and output ports. In an input-queued crossbar switch, each input port can be connected to only one output port and vice versa in each switching cycle or time slot. Hence, in every time slot, the switch needs to compute a one-to-one *matching* between input and output ports (*i.e.*, the crossbar schedule). A major research challenge in designing high-link-rate high-radix switches is to develop algorithms that can compute "high quality" matchings – *i.e.,* those that result in high switch throughput and low queueing delays for packets – in a few nanoseconds. Clearly, a suitable crossbar scheduling algorithm has to have very low time complexity, yet output "fairly good" matching decisions most of time.

While considerable research was performed on crossbar scheduling around the turn of the century, the resulting algorithms either have a (relatively) high time complexity that prevents a matching computation from being completed in a short time slot, or cannot deliver excellent throughput and delay performances. For example, SERENA [4] and iSLIP [5] are

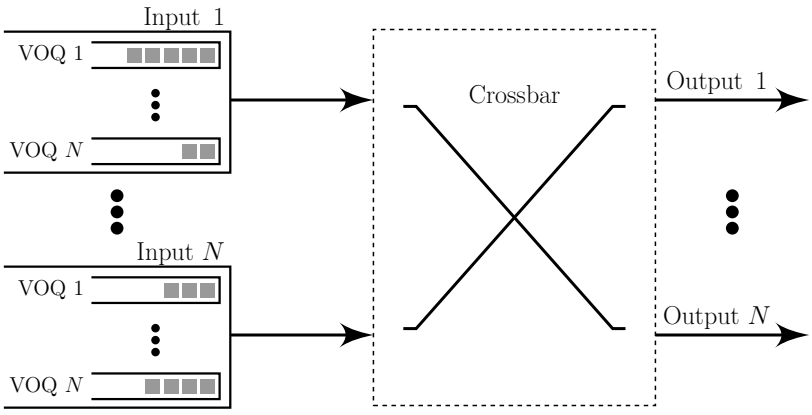


Figure 1.1: Input-queued crossbar switch.

two well-known crossbar scheduling algorithms of that generation. On one hand, although SERENA can attain 100% throughput and has acceptable delay performance, its time complexity is $O(N)$, which is still too high for the matching to be computed in a short time slot when N is large. On the other hand, iSLIP is a parallel iterative algorithm and has a lower per-port time complexity of $O(\log N)$ for each iteration, but iSLIP cannot attain 100% throughput except under the uniform traffic.

1.2 Background

1.2.1 Input-Queued Crossbar Architecture

Figure 1.1 shows a generic input-queued switch employing a crossbar to interconnect N input ports with N output ports. Each input port has N Virtual Output Queues (VOQs) [6]. A VOQ j at input port i serves as a buffer for the packets going into input port i destined for output port j. In such a switch, each input port can be connected to only one output port, and vice versa, in each time slot. Hence, it needs to compute, per time slot, a one-to-one *matching* between input and output ports.

Throughout this thesis, we assume that all incoming variable-length packets are segmented into fixed-length packets (sometimes referred to as cells), which are then reassembled when leaving the switch. Hence we consider the switching of only fixed-length packets in the sequel, and each such fixed-length packet takes exactly one time slot to transmit. We

also assume that both the output ports and the crossbar operate at the same speed as the input ports. Both assumptions above are widely adopted in the literature (*e.g.,* [7, 5, 4]).

1.2.2 System Models

In crossbar scheduling, an $N \times N$ crossbar is generally modeled as a bipartite graph and the crossbar scheduling needs to solve a *bipartite matching problem* [8] in such a bipartite graph during each time slot. Based on whether taking into the weight (*e.g.,* VOQ length) information into consideration, there are two commonly used problem models, namely, the *weight-oblivious model* and the *weight-aware model*.

Weight-Oblivious Model. In this model, an $N \times N$ input-queued crossbar is modeled as a (unweighted) bipartite graph $G(I \bigcup O)$, of which the two disjoint vertex sets $I = \{I_1, I_2, \cdots, I_N\}$ and $O = \{O_1, O_2, \cdots, O_N\}$ are the N input ports and the N output ports respectively. We note that the edge set in this bipartite graph might change from a time slot to another. In this bipartite graph during a certain time slot t, there is an edge between input port i and output port j, if and only if the VOQ j at input port i, the corresponding VOQ, is nonempty at time slot t. A set of such edges constitutes a *valid crossbar schedule*, or a *matching*, if any two of them do not share a common vertex.

Weight-Aware Model. In this model, an $N \times N$ crossbar is modeled as a weighted complete bipartite graph $G(I \bigcup O)$, with the N input ports and the N output ports represented as the two disjoint vertex sets $I = \{I_1, I_2, \cdots, I_N\}$ and $O = \{O_1, O_2, \cdot, O_N\}$ respectively. We note that the weight of each edge in this bipartite graph might change from a time slot to another. Each edge (I_i, O_j), *i.e.,* the edge between input port i and output port j, corresponds to the VOQ j at the input port i, and its weight is the queue length (*i.e.,* the number of packets buffered) of the VOQ at the corresponding time slot. We denote this edge also as $I_i \rightarrow O_j$ when its direction is emphasized. Same as in the weight-oblivious model, a *valid schedule*, or *matching*, is a set of edges between the N input ports and the N output ports, in which no two distinct edges share a vertex. The weight of a matching is the total

weight of all the edges belonging to it (*i.e.,* the total length of all corresponding VOQs). We say a matching is *full* if all vertices in the bipartite graph $G(I \bigcup O)$ are an endpoint of an edge in the matching, and is *partial* otherwise. Clearly, in an $N \times N$ crossbar, any full matching contains exactly N edges.

In both models, each matching M can be represented as an $N \times N$ *sub-permutation matrix* (a 0-1 matrix that contains at most one entry of "1" in each row and in each column) $S = (s_{ij})$ as follows: $s_{ij} = 1$ if and only if the edge between input port i and output port j is contained in M (*i.e.,* input port i is matched to, or paired with, output port j in M). To avoid any confusion, only S (not M) is used to denote a matching in the sequel, and it can be both a set (of edges) and a matrix.

1.2.3 Three Types of Matchings

Three types of matchings play important roles in crossbar scheduling problems: (I) maximal matchings, (II) maximum matchings, and (III) maximum weighted matchings. A matching S is called a *maximal matching*, if it is no longer a matching, when any edge not in S is added to it. A matching with the largest possible number of edges is called a *maximum matching* or *maximum cardinality matching*. Neither maximal matchings nor maximum matchings take into account the weights of edges, whereas *maximum weighted matchings* do. A maximum weighted matching is one that has the largest total weight among all matchings. By definition, any maximum matching or maximum weighted matching is also a maximal matching, but neither converse is generally true.

1.2.4 Performance Metrics

The research objective of crossbar scheduling is to design scheduling algorithms that compute a good matching, as measured by certain performance metrics, in each time slot, with a reasonable amount of computation. Typically, scheduling algorithms are evaluated on three performance metrics: *throughput*, *delay*, and *complexity*.

Throughput. Normalized throughput is defined as the average number of packets that exit an output port during each time slot. It is a value between 0 and 1 (*i.e.,* 100%). Throughout this thesis, we mean normalized throughput whenever we use the word "throughput".

We say a switch, employing a certain crossbar scheduling algorithm, is stable [9] – under a certain workload – if its total queue (VOQ) length $\|Q(t)\|_1$ satisfies $\sup_{0\leq t<\infty} \mathbb{E}\big[\|Q(t)\|_1\big] < \infty$. A crossbar scheduling algorithm is said to achieve 100% *throughput*, if the switch is stable under any traffic arrival process that is admissible (defined next) and satisfies certain other mild conditions (see §4.3.1). For example, SERENA [4, 10] can achieve 100% throughput under any such admissible arrival process, whereas iSLIP [5] generally cannot.

Delay. We define delay as the number of time slots elapsed since the arrival of a packet to its eventual departure from the switch. An ideal scheduling algorithm has 100% throughput and low delay. Achieving 100% throughput is relatively easier than achieving low delay. For instance, in TASS [11], 100% throughput is achieved, at the cost of high delays, using a simple randomized adaptive algorithm that we will describe in §4.3.2.

Complexity. Another criterion for evaluating a scheduling algorithm is the time complexity of computing a matching. As mentioned earlier, folklore suggests a tradeoff between the quality of matching and the time complexity. A key contribution of this thesis is to strike better performance-complexity tradeoffs than existing crossbar scheduling algorithms such as iSLIP and SERENA.

1.2.5 Admissible Traffic Patterns

Let λ_{ij} be the normalized (to the percentage of the rate of an input/output link) mean arrival rate of packets to the VOQ j (*i.e.,* those destined for output port j) at input port i. Then the traffic pattern, represented by an $N \times N$ traffic matrix $\Lambda = \{\lambda_{ij}\}_{N\times N}$, is called *admissible*

if $\rho_\Lambda < 1$, where ρ_Λ, defined as,

$$\rho_\Lambda \triangleq \max\Big\{ \max_{1\le i\le N}\{\sum_j \lambda_{ij}\}, \max_{1\le j\le N}\{\sum_i \lambda_{ij}\}\Big\} \tag{1.1}$$

is the maximum load factor imposed on any input or output port by Λ. Clearly, $\rho_\Lambda < 1$ is a necessary condition for any crossbar scheduling algorithm to ensure the stability of an input-queued crossbar switch. In the sequel, we drop the subscript term from ρ_Λ and simply denote it as ρ.

Now we state a well-known fact that has been used, usually without a proof, in almost every switch stability proof in the literature.

Fact 1.2.1. For each $N \times N$ admissible traffic matrix Λ, whose maximum per input/output load is ρ (defined in (1.1)), there exist $N \times N$ matching (sub-permutation) matrices M_n, $n = 1, 2, \ldots, K$ such that

$$\Lambda = \sum_{n=1}^{K} \alpha_n M_n \tag{1.2}$$

where $K \le N^2 - 2N + 2$, $\alpha_n > 0$ and $\sum_{n=1}^{K} \alpha_n \le \rho$.

This fact follows from the fact that Λ/ρ is a sub-stochastic matrix, which can be expressed as a linear combination of sub-permutation matrices with positive coefficients summing up to a value no larger than 1, known as the Birkhoff–von Neumann decomposition [12, 13, 14].

1.2.6 The Four Standard Traffic Patterns

Like in many of the literature on crossbar scheduling, the following four standard types of traffic patterns (*i.e.,* load matrices) are used to generate the workloads of the switch in all simulations in this thesis: (I) *Uniform*: packets arriving at any input port go to each output port with probability $\frac{1}{N}$. (II) *Quasi-diagonal*: packets arriving at input port i go to

output port $j = i$ with probability $\frac{1}{2}$ and go to any other output port with probability $\frac{1}{2(N-1)}$. (III) *Log-diagonal*: packets arriving at input port i go to output port $j = i$ with probability $\frac{2^{(N-1)}}{2^N-1}$ and go to any other output port j with probability equal $\frac{1}{2}$ of the probability of output port $j-1$ (note: output port 0 equals output port N). (IV) *Diagonal*: packets arriving at input port i go to output port $j = i$ with probability $\frac{2}{3}$, or go to output port $(i \bmod N) + 1$ with probability $\frac{1}{3}$. The traffic patterns above are listed in order of how skewed the traffic volumes to different output ports are: from uniform being the least skewed, to diagonal being the most skewed.

1.3 Summary of Contributions

In this thesis, we have tackled the crossbar scheduling problem from the following two aspects. The first aspect is to investigate next-generation bipartite matching algorithms for crossbar scheduling, that are low in time complexities (preferably $O(1)$ and definitely no more than $O(\log N)$ per port), yet have excellent throughput and delay performances. The second aspect is to rigorously analyze the throughput and the delay performance guarantees of some of the proposed algorithms using Lyapunov stability analysis techniques [15].

Along the first aspect, we have made four breakthroughs. Our first breakthrough is to discover that SERENA can after all be parallelized: In Chapter 3, we have proposed SERENADE (SERENA, the Distributed Edition) that exactly emulates SERENA and has a low per-port time complexity of $O(\log N)$. Our second breakthrough is an add-on algorithm called Queue-Proportional Sampling (QPS). We have shown in Chapter 4 that QPS can be used to augment and significantly boost the throughput and delay performances of other crossbar scheduling algorithms such as SERENA [4] and iSLIP [5], at virtually no additional computation cost due to its low time complexity of $O(1)$ per port. Our third breakthrough is QPS-r, a standalone parallel iterative scheduling algorithm that simply runs a constant r rounds (iterations) of QPS to compute a matching. We have shown in Chapter 5 that in a single iteration (*i.e.*, when $r = 1$), QPS-1 outputs a matching that is in general

not even maximal, yet has exactly the same quality as maximal matchings, in the following sense: Using such matchings as crossbar schedules results in exactly the same provable throughput lower bound of 50% and delay guarantees as using maximal matchings. This discovery is significant because, to the best of our knowledge, all existing parallel iterative crossbar scheduling algorithms (*e.g.,* PIM [16], iSLIP [5], RR/LQF [17]) require up to N iterations to compute a maximal matching. In other words, all existing parallel iterative crossbar scheduling algorithms require up to N iterations to achieve the same provable throughput and delay performance guarantees. Building on QPS, we have made our fourth breakthrough: Small Batch QPS (SB-QPS), a batch scheduling algorithm that has all the desired properties of next-generation matching algorithms (for crossbar scheduling) mentioned above. Along the second aspect, we have investigated the throughput and/or the delay performance guarantees of some of these crossbar scheduling algorithms, such as QPS-SERENA and QPS-r.

We now present an overview of the aforementioned crossbar scheduling algorithms proposed in this thesis, along with the key contributions.

1.3.1 SERENADE (Chapter 3)

In this work, we propose SERENADE (SERENA, the Distributed Edition), a parallel iterative algorithm that emulates each matching computation of SERENA using only $O(\log N)$ iterations between input ports and output ports. Hence, each input or output port needs to do only $O(\log N)$ work to compute a matching, making SERENADE scalable in both the switch size and the line rate per port.

SERENADE consists of two stages: a knowledge-discovery stage and a distributed binary search stage. The knowledge-discovery stage uses a knowledge-discovery procedure, which has at most $1 + \log_2 N$ iterations, to gather information at each input port. After this stage, some input ports might be able to make the same matching decisions as they would under SERENA, whereas other input ports are not able to do so. Then, in the dis-

tributed binary search stage, those input ports will also be able to make the same matching decisions as they would do under SERENA by performing an additional distributed binary search, which has at most $\log_2 N$ iterations, guided by the information gathered during the knowledge-discovery stage. We prove that SERENADE exactly emulates SERENA.

SERENADE overcomes the challenge of parallelizing SERENA, namely the monolithic nature of the MERGE procedure, by making do with less. More specifically, we will show toward the end of §3.3.1, in SERENADE, after its $O(\log N)$ iterations, each input port has much less information to work with than the (central) switch controller in SERENA. In other words, SERENADE does not precisely parallelize SERENA, in that it does not duplicate the full information gathering capability of SERENA; rather, it gathers just enough information needed to make a matching decision that is exactly as wise. This making do with less is a major innovation and contribution of this work.

To reduce the complexities, we propose an early-stop version of SERENADE, called O-SERENADE, to approximately emulate SERENA. O-SERENADE gets rid of the distributed binary search and only approximately emulates SERENA by making opportunistic matching decisions after the knowledge-discovery stage. Despite this approximation, the delay performance of O-SERENADE is similar to or slightly better than that of SERENA, under various load conditions and traffic patterns.

1.3.2 QPS (Chapter 4)

In this work, we propose a general approach that can significantly boost the performance of both SERENA and iSLIP, yet incurs only $O(1)$ additional time complexity at each input/output port. Our approach is a novel proposing strategy, called *Queue-Proportional Sampling (QPS)*, that generates an excellent *starter matching*, better than the arrival graph used by SERENA. Scheduling algorithms that start from "scratch" (*i.e.,* an empty matching), such as iSLIP, may also benefit significantly from QPS, by instead starting from a QPS-generated starter matching.

Our proposing strategy, QPS, at any input port, is extremely simple to state: the input port proposes to an output port with a probability proportional to the length of the corresponding VOQ. QPS's name comes from the fact that the output port proposed to by any input port is sampled, out of all N output ports, using the queue-proportional distribution at the input port. We note that, although this general approach – of serving queues at rates/probabilities proportional to their lengths – to resource allocation is classical [18], QPS is a novel application of this approach to crossbar scheduling.

We will show in §4.2 that QPS is also extremely cheap to execute: we developed an $O(1)$ data structure and algorithm for generating such a sample at each input port. This may be surprising to readers, since even to "read" the lengths of all N VOQs at an input port takes $O(N)$ time. Due to its $O(1)$ (per port) time complexity, any QPS-augmented algorithm has the same asymptotic complexity as the original algorithm.

In this work, we consider two QPS-augmented algorithms: QPS-iSLIP and QPS-SERENA, which combine QPS with iSLIP [5] and SERENA [4, 10] respectively. Both QPS-augmented algorithms are shown to outperform the original algorithms, in both throughput and delay, under various load conditions and traffic patterns, by a wide margin in §4.4. As the QPS approach is very general, it can be used to augment other low-complexity crossbar scheduling algorithms in the future.

We make the following three major contributions in this work. First, we propose QPS, a simple yet effective approach to crossbar scheduling, and use it to augment both iSLIP and SERENA. Second, we propose a data structure that carries out each QPS operation with only $O(1)$ computation per port. Third, for proving the stability of QPS-SERENA, we derive a new and stronger theorem for proving the stability of a large family of crossbar scheduling algorithms.

1.3.3 QPS-r (Chapter 5)

In this work, we propose QPS-r, a parallel iterative algorithm that has the lowest possible time complexity: $O(1)$ per port. More specifically, QPS-r requires only r (a small constant independent of N) iterations to compute a matching, and the time complexity of each iteration is only $O(1)$; here QPS stands for Queue-Proportional Sampling, an add-on technique, as mentioned earlier, we will describe it in detail in Chapter 4. Yet, even the matchings that QPS-1 (running only a single iteration) computes have the same quality as maximal matchings in the following sense: Using such matchings as crossbar schedules results in exactly the same aforementioned provable throughput and delay guarantees as using maximal matchings, as we will show using Lyapunov stability analysis. Note that QPS-r performs as well as maximal matching algorithms not just in theory: We will show that QPS-3 (running 3 iterations) has comparable empirical throughput and delay performances as iSLIP (running $\log_2 N$ iterations), a refined and optimized representative maximal matching algorithm adapted for switching, under various load conditions and traffic patterns.

QPS-r has another advantage over parallel iterative maximal matching algorithms such as iSLIP and PIM: Its per-port communication complexity is also $O(1)$, much smaller than that of maximal matching algorithms such as iSLIP. In each iteration of QPS-r, each input port sends a request to only a single output port. In comparison, in each iteration of PIM or iSLIP, each input port has to send requests to all output ports to which the corresponding VOQs are nonempty, which incurs $O(N)$ communication complexity per port.

Although QPS-r builds on the QPS data structure and algorithm proposed in Chapter 4, our work on QPS-r is very different in three important aspects. First, in Chapter 4, QPS was used only as an add-on to other crossbar scheduling algorithms such as SERENA [4, 10] and iSLIP [5] by generating a starter matching for other crossbar scheduling algorithms to further refine, whereas in this work, QPS-r is used only as a stand-alone algorithm. Second, in this work, we discover and prove that (QPS-r)-generated matchings and maximal matchings provide exactly the same aforementioned throughput and delay guarantees, whereas

in Chapter 4, no such mathematical similarity or connection was mentioned. Third, the establishment of this mathematical similarity is an important theoretical contribution in itself, because maximal matchings have long been established as a cost-effective family both in crossbar scheduling [16, 5] and in wireless networking [19, 20], and with this connection we have considerably enlarged this family.

Although we show that QPS-r has exactly the same throughput and delay bounds as that of maximal matchings established in [21, 19, 20], our proofs are different for the following reason. A *departure inequality* (see Property 5.2.1), satisfied by all maximal matching algorithms was used in the stability analysis of [21] and the delay analysis of [19, 20]. This inequality, however, is not satisfied by QPS-r in general. However, QPS-r satisfies this departure inequality in expectation, which is a weaker guarantee. A methodological contribution of this work is to prove two theorems stating that this much weaker guarantee is sufficient for obtaining the same throughput and delay bounds respectively.

1.3.4 SB-QPS (Chapter 6)

This work is motivated by the following two observations: (I) The throughput performance of QPS-r, even when r=3, is only around 80% under the aforementioned four standard traffic patterns and grows very slows when r increases beyond 3; and (II) it is possible to improve the quality of the matching without increasing the time complexity of the crossbar scheduling algorithm using a strategy called batching [22, 23, 24]. Unlike in a regular crossbar scheduling algorithm, where a matching decision is computed for every time slot, in a batch scheduling algorithm, multiple (say T) consecutive time slots are grouped as a batch and these T matching decisions are batch-computed. Hence, in a batch scheduling algorithm, each of the T matchings-under-computation in a batch has a period of T time slots to find opportunities to have the quality of the matching (in terms of cardinality and/or weight) improved by the underlying bipartite matching algorithm, whereas in a regular crossbar scheduling algorithm, each matching has only a single time slot to find such

opportunities. As a result, a batch scheduling algorithm can usually produce matchings of higher qualities than a regular crossbar scheduling algorithm using the same underlying bipartite matching algorithm, because such opportunities for improving the quality of a certain matching usually do not all present themselves in a single designated time slot (for a regular crossbar scheduling algorithm to compute this matching). Clearly, the larger the batch size T is, the better the quality of a resulting matching is, since a larger T provides a wider "window of opportunities" for improving the quality of the resulting matching as just explained.

However, existing batch scheduling algorithms are not without shortcomings. They all suffer from at least one of the following two problems. First, all existing batch scheduling algorithms except [24] are sequential algorithms and it is not known whether any of them can be parallelized. As a result, they all have a time complexity of at least $O(N)$ per matching computation, since it takes $O(N)$ time just to "print out" the computed result. This $O(N)$ time complexity is clearly too high for high-radix high-line-rate switches as just explained. Second, most existing batch scheduling algorithms require a large batch size T to produce high-quality matchings that lead to high throughputs, as will be elaborated in Chapter 2. A large batch size T is certain to lead to poor delay performance: Regardless of the offered load condition, the average packet delay for any batch scheduling algorithm due to batching is at least $T/2$, since any packet belonging to the current batch has to wait till at least the beginning of the next batch to be switched.

In this work, building upon QPS, we propose a novel batch scheduling algorithm, called SB-QPS (Small-Batch QPS), that addresses both weaknesses of existing batch scheduling algorithms. First, it can attain a high throughout of over 0.87, under various traffic patterns, using only a small batch size T being 32 time slots. This much smaller batch size translates into much better delay performances than those of existing batch scheduling algorithms, as will be shown in §6.3. Second, SB-QPS is a fully distributed algorithm so that the matching computation load can be efficiently divided evenly across the $2N$ input and output ports.

As a result, its time complexity is the lowest possible: $O(1)$ per matching computation per port.

1.4 Bibliographic Note

The contents presented in Chapter 3, Chapter 4 and Chapter 5 appear as three conference papers in [25], [26] and [27], respectively.

CHAPTER 2

LITERATURE REVIEW

2.1 Crossbar Scheduling Algorithms

2.1.1 Parallel/Distributed MWM Algorithms

Using Maximum Weighted Matchings (MWM) as crossbar schedules results in 100% throughput and near-optimal delay performance, but its state-of-the-art implementation [28] has a prohibitively high time complexity of $O(N^{2.5} \log W)$, where W is the maximum possible length of a VOQ. Note that MWM-α [29] and MWM-0^+ [30] are variants that only explore the MWM policy space by adopting different edge weight functions; they contain no algorithmic innovations that would reduce the complexity of MWM. This dilemma has motivated the development of a few parallel or distributed algorithms that, by distributing this computational cost across multiple processors (nodes), bring down the per-node time complexity.

The most representative among them are [31, 32, 33]. A parallel algorithm with a sub-linear per-node time complexity of $O(\sqrt{N} \log^2 N)$ was proposed in [31] for computing MWM exactly in a bipartite graph. However, this algorithm requires the use of $O(N^3)$ processors. Another two [32, 33] belong to the family of parallel/distributed iterative algorithms based on belief-propagation (BP). In this family, the input ports engage in multiple iterations of message exchanges with the output ports to learn enough information about the lengths of all N^2 VOQs so that each input port can decide on a distinct output port to match with. The resulting matching either is, or is close to, the MWM. Note that the BP-based algorithms are simply parallel/distributed algorithms to compute the MWM: the total amount of computation, or the total number of messages needed to be exchanged, is $O(N^3)$, but is distributed evenly across the input and the output ports (*i.e.,* $O(N^2)$ work for

each input/output port).

2.1.2 MVM and LHPF

Another approach to reducing the complexity to $O(N^{2.5})$ while achieving performance similar to MWM is the family of Maximum Vertex-weighted Matching (MVM) policies [7]. The MVM family was later extended to a larger family called Lazy Heaviest Port First (LHPF) [34] that also has $O(N^{2.5})$ complexity. In a standard MVM policy, each input or output port, denoted as a vertex, is assigned a weight that is equal to the total number of packets (across all N VOQs) queued at the vertex. The weight of an edge (i, j) is the sum of the weights of its two vertices i and j, if there is at least one packet in the corresponding VOQ (*i.e.,* the VOQ j at input port i), and is 0 otherwise. An MVM policy dictates that the heaviest (vertex-weighted) matching be used for crossbar scheduling. MVM can achieve 100% throughput, and has a delay performance quite close to that of MWM.

2.1.3 BP-Assisted Algorithms

A technique called BP-assisted scheduling (here BP stands for Belief Propagation) was proposed in a recent work [35], in which BP is used to boost the performance of certain parallel/distributed iterative algorithms (called "carrier" algorithms) that are not BP-based such as iLQF [36]. Its idea is to replace the contents of the messages exchanged between input and output ports by those that would be exchanged in a BP-based algorithm. The "BP assistance" part alone has a total time complexity of $O(N^2)$, so it is best suited for a carrier algorithm that has the same asymptotic complexity, such as iLQF.

2.1.4 Lower-Complexity Randomized Algorithms

Several randomized algorithms, starting with TASS [11] and culminating in SERENA [4, 10] were proposed to push the total complexity further down to $O(N)$ (*i.e.,* linear complexity). We will describe in §4.3.2 both TASS and SERENA in detail.

A randomized scheduling algorithm specialized for switching variable-length packets was proposed in [37] that has $O(1)$ total time complexity (per switch). It belongs to a family of randomized algorithms (*e.g.,* [38, 39, 40, 41]) primarily designed for computing a collision-free transmission schedule, which corresponds to an independent set in the interference graph, in a wireless network. These algorithms all build upon a Markov Chain Monte-Carlo (MCMC) technique called Glauber dynamics [42] for computing independent sets (convertible to bipartite matchings in the switching context).

The algorithm in [37] for computing, at each time slot t, the matching for the next time slot $S(t+1)$, works follows. It samples one of the N^2 VOQs (edges) uniformly at random. Suppose the sampled VOQ (edge) is the VOQ j at input port i (*i.e.,* edge (i, j)). Then, with probability $e^w/(e^w + 1)$, it adds the edge (i, j) (*i.e.,* pairing input port i with output port j) to or keeps the edge in $S(t + 1)$, if neither i nor j is currently matched (in $S(t)$) or (i, j) already belongs to the $S(t)$. Here the weight w is set to the celebrated slowly varying weight function $\ln(\ln(e+x))$ proposed in [38], where x is the weight of the edge (i, j) (*i.e.,* the length of the corresponding VOQ). Clearly, the algorithm makes at most one change (hence $O(1)$ total complexity), from any time slot t to the next, to the configuration of the crossbar (*i.e.,* the matching).

It was proven in [38] that all such algorithms that use this weight function, including the algorithm in [37], can achieve 100% throughput. However, our simulation results (presented in §B.5.4) show that, when used for switching fixed-length packets, the algorithm in [37] has very poor delay performance and the total queue length does not stabilize (*i.e.,* keeps increasing) until after a very large number of time slots. These simulation results are not surprising: all algorithms that adopt this $\ln \ln(e + \cdot)$ weight function have similar poor delay performance, because as explained in [39], the $\ln \ln(e + \cdot)$ weight function, aimed at achieving 100% throughput [38], reacts very slowly to changes in queue lengths and hence allows long queues to build up.

2.1.5 Parallel Iterative Algorithms

As mentioned earlier, maximal matchings have long been recognized as a cost-effective family in switching. Among various types of algorithms that compute maximal matchings, the family of parallel iterative algorithms [43, 17, 44, 36, 45, 46] is widely adopted. Parallel iterative algorithms compute a maximal matching via multiple iterations of message exchanges between the input and output ports. Generally, each iteration contains three stages: request, grant, and accept. In the request stage, each input port sends requests to output ports. In the grant stage, each output port, upon receiving requests from multiple input ports, grants to one. Finally, in the accept stage, each input port, upon receiving grants from multiple output ports, accepts one. Unfortunately, all these parallel iterative algorithms in switching require up to N iterations to guarantee that the resulting matching is a maximal matching. In other words, they need up to N iterations to achieve the same provable throughput and delay performance guarantees as QPS-1 (running 1 iteration).

2.1.6 Batch Scheduling Algorithms

In all algorithms above, a matching decision is made every time slot. An alternative type of algorithms is batch scheduling [22, 23, 47, 24, 48] in which multiple (say K) consecutive time slots are grouped a batch and these K matching decisions are batch-computed, which usually has lower time complexity than K independent matching computations. However, since K is usually quite large (*e.g.,* $O(N^2)$ [22]), and a packet arriving at the beginning of a batch has to wait till at least the beginning of the next frame to be switched, batch scheduling generally results in higher queueing delays.

Although batch scheduling can reduce the average time complexity of a matching computation via amortization over the batch, most of existing batch scheduling algorithms are centralized and have a relatively high time complexity even after the amortization. For example, the Fair-Frame algorithm [23] based on the Birkhoff–von Neumann Decomposition (BvND) has a time complexity of $O(N^{1.5} \log N)$ per matching computation.

Recently, a parallel batch scheduling algorithm based on complex coloring [49] was proposed in [24]. Although this algorithm requires a batch size of only $O(\log N)$ and has a relatively low time complexity of only $O(\log N)$ per time slot per port, it does not provide any provable throughput guarantees. In addition, the constant factor in the first $O(\log N)$ is large: A batch size of $3{,}096$ was needed for the algorithm to attain around 96% throughputs under some traffic patterns when $N = 300$ as reported in [24].

2.2 Queue-Proportional Resource Allocation

Serving queues at rates or probabilities proportional to their (queue) lengths is an intuitively appealing resource allocation approach that has been used in various computer and communications systems for many years. For example, in [18], a simple queue-proportional scheduler was proposed for scheduling transmissions in wireless broadcast channels, and a geometric programming based formulation of this problem specialized to the Gaussian broadcast channel was later established in [50, 51]. However, unlike our QPS strategy, in which an input port proposes to an output port with a *probability* proportional to the length of the corresponding VOQ, the scheduler in [18, 50, 51] dictates that each link receives an service *rate* proportional to its current queue length during each time slot. As a result, it has to solve a convex optimization problem that has a much higher time complexity.

In [52], B. Li *et al.* proposed a generalized version of the above queue-proportional scheduler called Queue-Proportional Rate Allocation (QPRA), with the objective of achieving maximum throughput in a multi-hop wireless network. As the QPRA algorithm is generally hard to implement in practice, they further proposed a low-complexity version called LC-QPRA to make their scheme more practical. The LC-QPRA algorithm resembles the proposing step in our QPS scheme in that, during each time slot, a sender proposes (attempts to transmit) to each receiver with a probability proportional to the length of the corresponding "VOQ".

There are three key differences between QPRA and QPS however. First, in QPRA,

during any time slot, the probability with which each sender proposes (to any receiver) is also proportional to its total queue length, whereas in QPS, this probability is 1 for any sender unless its total queue length is 0. Second, in QPRA, if two senders propose (transmit) to the same receiver during a time slot, both transmissions are corrupted, whereas in QPS, only one is allowed to eventually transmit a packet to the receiver. Third, in QPRA, the outcomes (successful or corrupted) of these proposals (attempted transmissions) define the final matching, whereas QPS only generates a starter matching that will be further refined into a full or more complete matching.

Finally, another policy was proposed in [53] for scheduling packets in a single-hop network, where crossbar scheduling is a special case. However, this policy is closely related to MWM-0^+ [30], and is unrelated to QPRA or QPS.

2.3 Wireless Transmission Scheduling

Transmission scheduling in wireless networks with primary interference constraints [54] shares a common algorithmic problem with crossbar scheduling: to compute a good matching for each "time slot". The matching computation in the former case is however more challenging, since it needs to be performed over a general graph that is not necessarily bipartite. Several wireless transmission scheduling solutions were proposed in the literature [55, 54, 56, 57, 58, 59] that are based on distributed computation of matchings in a general graph.

Most of these solutions tackle the underlying distributed matching computation problem using an adaptation/extension of either [60] (used in [58, 57, 59]), or [61] (used in [56, 55]). In [60], a parallel randomized algorithm was proposed that outputs a maximal matching with expected runtime $O(\log |E|)$, where $|E|$ is the number of edges in the graph. This time complexity, translated into our crossbar scheduling context, is $O(\log N)$. However, maximal matching algorithms are known to only guarantee at least 50% throughput [7]. The work of Hoepman [61] converts an earlier sequential algorithm for computing approx-

imate MWM [62] to a distributed one. However, the distributed algorithm in [61], like its sequential version [62], can only guarantee to find a matching whose weight is at least half of that of the MWM, and hence can only guarantee at least 50% throughput also.

The only exception, to distributed matching algorithms being based on either [60] or [61], is [54], in which the scheduling algorithm, called MIX, is a distributed version of the MERGE procedure in SERENA, albeit in the wireless networking context. The objective of MIX is to compute an approximate MWM for simultaneous non-interfering wireless transmissions of packets, where the weight of a directed edge (say a wireless link from a node X to a node Y) is the length of the VOQ at X for packets destined for Y, in the SERENA manner: MERGE the matching used in the previous time slot with a new random matching. Unlike in SERENA, however, neither matching has to be full and the connectivity topology is generally not bipartite in a wireless network, and hence the graph resulting from the union of the two matchings can contain both cycles and paths.

MIX has three variants. As we will explain in §A.6 in details, all three variants compute the total – or equivalently the average – green and red weights of each cycle or path either by linearly traversing the cycle or path, or via a gossip algorithm [63]; they all try to mimic SERENA in a wireless network and have a time complexity at least $O(N)$, as compared to $O(\log N)$ for SERENADE. To summarize, they are clearly all "wireless SERENA", not "wireless SERENADE".

CHAPTER 3

SERENADE

3.1 SERENA

To explain SERENADE, we first explain SERENA [4, 10], the algorithm it parallelizes. SERENA consists of two steps. The first step is to derive a full matching $R(t)$ from the set of packet arrivals $A(t)$. The second step is to *merge* $R(t)$ with the full matching $S(t–1)$ used in the previous time slot, to arrive at the full matching $S(t)$ to be used for the current time slot t. After we briefly describe the first step, we will focus on the second step, MERGE, in the rest of this section. In [4, 10], the set of packet arrivals $A(t)$ is modeled as an *arrival graph*, which we denote also as $A(t)$, as follows: an edge (I_i, O_j) belongs to $A(t)$ if and only if there is a packet arrival[1] to the corresponding VOQ at time slot t. Note that $A(t)$ is not necessarily a matching, because more than one input ports could have a packet arrival (*i.e.,* edge) destined for the same output port at time slot t. Hence in this case, each output port prunes all such edges incident upon it except the one with the heaviest weight (with ties broken randomly). The pruned graph, denoted as $A'(t)$, is now a matching.

This matching $A'(t)$, which is typically partial, is then populated into a full matching $R(t)$ by pairing the yet unmatched input ports with the yet unmatched output ports in a round-robin manner. Although this POPULATE procedure alone, with the round-robin pairing, has $O(N)$ time complexity, we will show in §A.1 that it can be reduced to the computation of prefix sums and solved using the classical parallel algorithm [64] whose time complexity in this context is $O(\log N)$ per port.

[1]Like in [4, 10], we assume the arrival processes are *i.i.d.* Bernoulli. Therefore, there is at most one packet arrival to any input port during each time slot.

3.1.1 Overview of The MERGE Procedure

In this section, we explain the MERGE procedure through which SERENA selects heavier edges for $S(t)$ from both $R(t)$ with $S(t–1)$, so that the weight of $S(t)$ is larger than or equal to those of both $R(t)$ and $S(t-1)$. We color-code and orient edges of $R(t)$ and $S(t-1)$, like in [4, 10], as follows. We color all edges in $R(t)$ red and all edges in $S(t-1)$ green, and hence in the sequel, rename $R(t)$ to S_r ("r" for red) and $S(t–1)$ to S_g ("g" for green) to emphasize the coloring. *We drop the henceforth unnecessary term t here with the implicit understanding that the focus is on the MERGE procedure at time slot t.* We also orient all edges in S_r as pointing from input ports (*i.e.,* I) to output ports (*i.e.,* O) and all edges in S_g as pointing from output ports to input ports. We use notations $S_r(I \rightarrow O)$ and $S_g(O \rightarrow I)$ to emphasize this orientation when necessary in the sequel. Finally, we drop the term t from $S(t)$ and denote the final outcome of the MERGE procedure as S.

Now we describe how the two color-coded oriented full matchings $S_r(I \rightarrow O)$ and $S_g(O \rightarrow I)$ are merged to produce the final full matching S. The MERGE procedure consists of two steps. The first step is to simply union the two full matchings, viewed as two subgraphs of the complete bipartite graph $G(I \bigcup O)$ (see the **Weight-Aware Model** described in §1.2.2), into one that we call the *union graph* and denote as $S_r(I \rightarrow O) \bigcup S_g(O \rightarrow I)$ (or $S_r \bigcup S_g$ in short). In other words, the union graph $S_r(I \rightarrow O) \bigcup S_g(O \rightarrow I)$ contains the directed edges in both $S_r(I \rightarrow O)$ and $S_g(O \rightarrow I)$.

It is a mathematical fact that any such union graph can be decomposed into disjoint directed cycles [4]. Furthermore, each directed cycle, starting from an input port I_i and going back to itself, is an alternating path between a red edge in S_r and a green edge in S_g, and hence contains equal numbers of red edges and green edges. In other words, this cycle consists of a red sub-matching of S_r and a green sub-matching of S_g. Then in the second step, for each directed cycle, the MERGE procedure compares the weight of the red sub-matching (*i.e.,* the total weight of the red edges in the cycle), with that of the green sub-matching, and includes the heavier sub-matching in the final merged matching S.

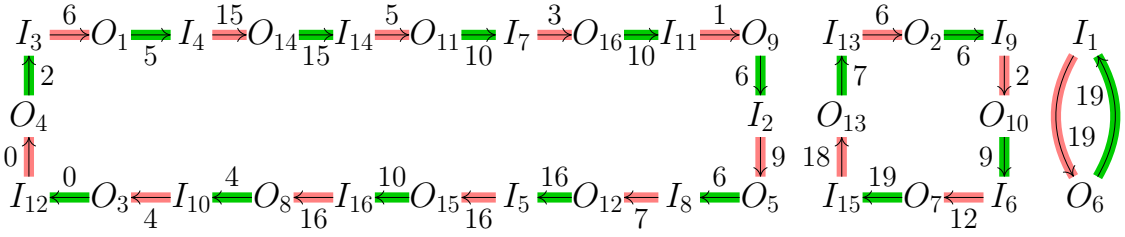


Figure 3.1: Cycles in $S_r(I \to O) \bigcup S_g(O \to I)$: Edges with red and green shadows are from $S_r(I \to O)$ and $S_g(O \to I)$ respectively.

An Illustrative Example. To illustrate the MERGE procedure by an example, Figure 3.1 shows the union graph of the following two full matchings over a 16×16 bipartite graph (crossbar). The number around each edge is its weight. The first full matching $S_r(I \to O)$, written as a permutation with input port numbers (1 as I_1, 2 as I_2, and so on) at the top and output port numbers at the bottom (1 as O_1, 2 as O_2, and so on), is $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 6 & 5 & 1 & 14 & 15 & 7 & 16 & 12 & 10 & 3 & 9 & 4 & 2 & 11 & 13 & 8 \end{pmatrix}$. We denote this permutation as σ_r. For example, σ_r mapping (input port) 3 to (output port) 1 corresponds to the red edge $I_3 \to O_1$ in Figure 3.1 (*i.e.*, I_3 pairing with O_1 in S_r, the "red" matching). The second full matching $S_g(O \to I)$, written as a permutation with output port numbers at the top and input port numbers at the bottom, is $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 4 & 9 & 12 & 3 & 8 & 1 & 15 & 10 & 2 & 6 & 7 & 5 & 13 & 14 & 16 & 11 \end{pmatrix}$. We denote this permutation as σ_g^{-1}. The union graph contains three disjoint directed cycles that are of lengths $22, 8, 2$ respectively. Now, we illustrate the MERGE procedure on the leftmost cycle in Figure 3.1. It is not hard to check that the total weight of the red sub-matching in this cycle is 82 and that of the green sub-matching is 84. Then, the heavier sub-matching, *i.e.,* the green one, is included into the final merged matching S. The standard centralized algorithm for implementing the MERGE procedure is to linearly traverse every cycle once, by following the directed edges in the cycle, to obtain the weights of the green and the red sub-matchings that comprise the cycle [4, 10]. Clearly, this algorithm has a time complexity of $O(N)$. The primary contribution of SERENADE is to reduce this complexity to $O(\log N)$ per input/output port through parallelization.

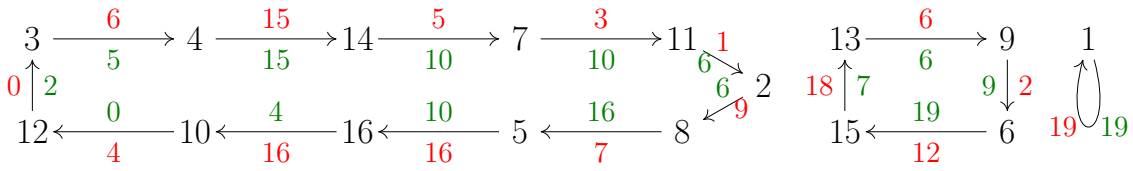


Figure 3.2: Combinatorial cycles correspond to the cycles in Figure 3.1.

3.1.2 A Combinatorial View of MERGE

In this section, to better describe MERGE under SERENADE however, we introduce a combinatorial view of MERGE, through which the MERGE procedure can be very succinctly characterized by a single permutation $\sigma \triangleq \sigma_g^{-1} \circ \sigma_r$, the composition of the two aforementioned full matchings $S_r(I \rightarrow O)$ and $S_g(O \rightarrow I)$ written as permutations. We do so using the example shown in Figure 3.1. It is not hard to verify that, in this example, $\sigma \triangleq \sigma_g^{-1} \circ \sigma_r = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 8 & 4 & 14 & 16 & 15 & 11 & 5 & 6 & 12 & 2 & 3 & 9 & 7 & 13 & 10 \end{pmatrix}$. We then decompose this permutation σ into disjoint combinatorial cycles. In this example, $\sigma = (3\,4\,14\,7\,11\,2\,8\,5\,16\,10\,12)(13\,9\,6\,15)(1)$, and its *cycle decomposition graph*, which contains precisely these three combinatorial cycles, is shown in Figure 3.2.

Note there is a one-to-one correspondence between the graph cycles (of the union graph $S_r \bigcup S_g$) shown in Figure 3.1 and the combinatorial cycles of σ shown in Figure 3.2. For example, cycle $I_3 \rightarrow O_1 \rightarrow I_4 \rightarrow O_{14} \rightarrow \cdots \rightarrow O_3 \rightarrow I_{12} \rightarrow O_4 \rightarrow I_3$, the leftmost cycle in Figure 3.1, corresponds to the leftmost combinatorial cycle $(3\,4\,14\,7\,11\,2\,8\,5\,16\,10\,12)$ in Figure 3.2. Note that two consecutive edges – one belonging to the red matching S_r and the other to the green matching S_g – on the graph cycle "collapse" into an edge on the corresponding combinatorial cycle. For example, two directed edges (I_3, O_1) $(\in S_r)$ and (O_1, I_4) $(\in S_g)$ in Figure 3.1 collapse into the directed edge from (input port) 3 to (input port) 4 in the combinatorial cycle $(3\,4\,14\,7\,11\,2\,8\,5\,16\,10\,12)$ in Figure 3.2. Hence each combinatorial cycle subsumes a red sub-matching and a green sub-matching that collapse into it. Note also that each vertex on the cycle decomposition graph corresponds to an input port. For example, vertex "3" in $(3\,4\,14\,7\,11\,2\,8\,5\,16\,10\,12)$ in Figure 3.2 corresponds to input port 3

(*i.e.,* I_3) in Figure 3.1. Hence, we use the terms "vertex" and "input port" interchangeably in the sequel of this chapter.

We assign a green weight $w_g(\cdot)$ and a red weight $w_r(\cdot)$ – to each combinatorial edge e in the cycle decomposition graph – that are equal to the respective weights of the green and the red edges that collapse into e. For example, the green and red numbers around each edge shown in Figure 3.2 represent its green and red weights respectively. We also define the green (or red) weight of a combinatorial cycle as the total green (or red) weight of all combinatorial edges on the cycle. Clearly, this green (or red) weight is equal to the weight of the green (or red) sub-matching this cycle subsumes. Under this combinatorial view, the MERGE procedure of SERENA can be stated as follows: *For each combinatorial cycle of* σ*, we compare its red weight with its green weight, and include in* S *the corresponding heavier sub-matching.*

3.1.3 Walks on Cycles

Finally, we introduce the concept of walk on a cycle decomposition graph. It greatly simplifies the descriptions of SERENADE, as it will become clear later that SERENADE is all about how to emulate SERENA using information, each input port obtains, regrading a few walks with lengths of power of 2. Recall that a *walk* in a general graph $G(V, E)$ is an ordered sequence of vertices, $v_1 \to v_2 \to \cdots \to v_k$ such that $(v_j, v_{j+1}) \in E$ for any $j \in \{1, 2, \ldots, k-1\}$; note that a *walk*, unlike a *path*, can traverse a vertex or edge more than once. Clearly, in the cycle decomposition graph of σ, every walk (say starting from a vertex i) circles around a combinatorial cycle (the one that i lies on), and hence necessarily takes the following form: $i \to \sigma(i) \to \sigma^2(i) \to \cdots \to \sigma^m(i)$. For notational convenience, we denote this walk as $i \rightsquigarrow \sigma^m(i)$. For example, with respect to the combinatorial cycle $(3\,4\,14\,7\,11\,2\,8\,5\,16\,10\,12)$ in Figure 3.2, the walk $3 \rightsquigarrow \sigma^8(3)$ represents 3→4→14→7→11→2→8→5→16, which consists of 8 directed edges on the cycle.

Generalizing this notation (of a walk), we define $\sigma^{m_1}(i) \rightsquigarrow \sigma^{m_2}(i)$ as the $(m_2 - m_1)$-

edge-long walk $\sigma^{m_1}(i) \to \sigma^{(m_1+1)}(i) \to \cdots \to \sigma^{m_2}(i)$, where $m_1 < m_2$ are integers, and both m_1 and m_2 could be negative. We define its red weight, denoted as $w_r(\sigma^{m_1}(i) \rightsquigarrow \sigma^{m_2}(i))$, as the sum of the red weights of all edges in $\sigma^{m_1}(i) \rightsquigarrow \sigma^{m_2}(i)$. Note that if an edge is traversed multiple times in a walk, the red weight of the edge is accounted for multiple times. The green weight of the walk, denoted as $w_g(\sigma^{m_1}(i) \rightsquigarrow \sigma^{m_2}(i))$, is similarly defined.

3.2 Overview of SERENADE

In this section, we provide a high-level overview of SERENADE. We first introduce the core idea of SERENADE that is based on an important concept called "discover". Then, we give a high-level description of two algorithmic stages of SERENADE: a knowledge-discovery stage and a distributed binary search stage. For ease of presentation (*e.g.,* no need to put floors or ceilings around each occurrence of $\log_2 N$), we have assumed that N is a power of 2 throughout this chapter; SERENADE works just as well when N is not.

3.2.1 Core Idea of SERENADE

The core idea of SERENADE is for all vertices on a combinatorial cycle, or a designated vertex among them, to *discover* (defined next) itself or another vertex on the same cycle twice. As will be shown next in Lemma 3.2.1, when this happens, these vertices will know precisely whether the green weight or the red weight of the cycle is larger, and hence will select the same heavier sub-matching as they would under SERENA. If this happens on every combinatorial cycle, then SERENADE exactly emulates SERENA.

Definition 3.2.1. Given two vertices i, j in any combinatorial cycle of σ, we say that vertex i *discovers* vertex j if i learns the identity of j and the (green and red) weights of a walk from i to j or from j to i.

By this definition, every vertex i discovers itself, once at the beginning (*i.e.*, before any algorithmic steps), via the empty (0-edge-long) walk from i to i.

Lemma 3.2.1 (Property of "Discover")**.** Let i and j be two vertices, which may or may not be the same vertex, on a combinatorial cycle of σ. If i discovers j twice via two different walks, then vertex i knows precisely whether the green weight or the red weight of the cycle is larger.

Proof: See §A.2. ■

3.2.2 High-Level Description of SERENADE

As mentioned earlier, SERENADE consists of two algorithmic stages: a knowledge-discovery stage and a distributed binary search stage. In this section, we give the high-level descriptions of these two stages, deferring their details to §3.3 and §3.5 respectively.

Knowledge-Discovery Stage. The knowledge-discovery stage uses the standard technique of two-directional exploration with successively doubled distance in distributed computing [65]. The basic idea of the algorithm is for each vertex i to exchange information, during the k^{th} ($1 \leq k \leq \log_2 N$) iteration[2], with vertices "($\pm 2^{k-1}$) σ-hops" away (*i.e.,* $\sigma^{2^{k-1}}(i)$ and $\sigma^{-(2^{k-1})}(i)$) to discover two vertices "($\pm 2^k$) σ-hops" away (*i.e.,* $\sigma^{2^k}(i)$ and $\sigma^{-(2^k)}(i)$). If either of the two vertices has been discovered twice by i, then, by Lemma 3.2.1, we know that vertex i can make the same matching decision as it would under SERENA. In the example shown in Figure 3.2, vertex 3 communicates, during the 1^{st} iteration, with vertices $4 = \sigma(3)$ and $12 = \sigma^{-1}(3)$ to discover vertices $14 = \sigma^2(3)$ and $10 = \sigma^{-2}(3)$, and communicates, during the 2^{nd} iteration, with the newly-discovered vertices 14 and 10 to discover vertices $11 = \sigma^4(3)$ and $5 = \sigma^{-4}(3)$, and so on in the next $(\log_2 N) - 2$ iterations.

Distributed Binary Search Stage. After the $1 + \log_2 N$ iterations of the knowledge discovery, a vertex i, residing on a cycle, will discover a vertex on the same cycle twice and hence make the same matching decision as it would under SERENA, if the cycle is ouroboros (to be defined in §3.3.4). However, not all cycles are ouroboros, as will be shown in §3.3.4.

[2] As will be shown in §3.3.2, there is a 0^{th} iteration at the beginning, with which each vertex i discovers $\sigma(i)$ and $\sigma^{-1}(i)$.

Those and only those vertices, residing on non-ouroboros cycles, then perform an additional distributed binary search, the purpose of which is to let a designated vertex in each non-ouroboros cycle discover itself for a second time. We will show in §3.4 that the elections of those designated vertices (*i.e.,* leader election) can be seamlessly embedded into the $1+\log_2 N$ iterations of the knowledge-discovery procedure. We will show in §3.5 that the distributed binary search finishes in at most $\log_2 N$ iterations. After the distributed binary search, each designated vertex informs the switch controller whether the green or the red sub-matching should be selected on the non-ouroboros cycle it resides on. The switch controller then broadcasts these decisions to all N vertices, and every vertex on a non-ouroboros cycle will carry out the corresponding matching decision.

Theorem 3.2.1 is a main result of this work. Its proof is straightforward after we have proved the correctness of the knowledge-discovery procedure (§3.3.2) and the distributed binary search (§3.5).

Theorem 3.2.1. SERENADE exactly emulates SERENA [4, 10] within $O(\log N)$ iterations. More precisely, at most $1+\log_2 N$ iterations are needed for the knowledge discovery procedure and at most $\log_2 N$ iterations for the distributed binary search.

3.3 Knowledge-Discovery Stage

In this section, we describe the details of the knowledge-discovery stage. We start with describing the information obtained by the knowledge-discovery procedure in §3.3.1; the detailed algorithmic steps in each iteration will be described later in §3.3.2. In §3.3.3, we analyze the time and message complexities of the knowledge-discovery procedure. Finally, we explain in §3.3.4 which vertices can discover some vertex twice during the knowledge-discovery procedure by introducing the concept of "ouroboros cycle".

3.3.1 Knowledge Sets

We will show next that, for $0 \leq k \leq \log_2 N$ (there is a 0^{th} iteration at the beginning), after the k^{th} iteration, each vertex i learns the following two knowledge sets: $\phi_{k+}^{(i)}$ and $\phi_{k-}^{(i)}$. Knowledge set $\phi_{k+}^{(i)}$ contains three quantities concerning the vertex (input port) that is 2^k σ-hops "downstream" (*w.r.t.* the "direction" of σ), relative to vertex i, in the cycle decomposition graph of σ:

(1) $\sigma^{2^k}(i)$, the identity of that vertex,

(2) $w_r(i \rightsquigarrow \sigma^{2^k}(i))$, the red weight of the 2^k-edge-long walk from i to that vertex, and

(3) $w_g(i \rightsquigarrow \sigma^{2^k}(i))$, the green weight of the walk.

Similarly, knowledge set $\phi_{k-}^{(i)}$ contains the three quantities concerning the vertex that is 2^k σ-hops "upstream" relative to vertex i, namely $\sigma^{-2^k}(i)$, $w_r(\sigma^{-2^k}(i) \rightsquigarrow i)$, and $w_g(\sigma^{-2^k}(i) \rightsquigarrow i)$. For example, after the 3^{rd} iteration, vertex 3 learns the identities of $16 = \sigma^8(3)$ and $7 = \sigma^{-8}(3)$ (vertices "(± 8) σ-hops" away) and the green and the red weights of the walks $3 \rightsquigarrow \sigma^8(3)$ and $\sigma^{-8}(3) \rightsquigarrow 3$.

As we will show in §3.3.2, the knowledge-discovery procedure might halt before finishing the $1+\log_2 N$ iterations, so each vertex learns at most $2+2\log_2 N$ knowledge sets during the knowledge-discovery procedure. Note the $2+2\log_2 N$ knowledge sets are a tiny percentage of information the switch controller has under SERENA: the former scales as $O(\log N)$ whereas the latter scales as $O(N)$. For example, in Figure 3.2, vertex 3 knows only the values the permutation function $\sigma(\bullet)$ takes on argument values 3, 4, and 12, whereas under SERENA the (central) switch controller would know that on all $N = 16$ argument values. In general, different vertices have very different sets of such partial knowledge under SERENADE. For example, vertex 2 knows only the values the permutation function $\sigma(\bullet)$ takes on argument values 2, 8, and 11. However, despite this "blind men (different vertices) and an elephant (σ and the green and the red weights of all walks on the combinatorial cycles of σ)" situation, these vertices manage to collaboratively perform the approximate or the

for $k \leftarrow 0$ **to** $\log_2 N$ **do**
  if $k = 0$ **then**
    `// The` 0^{th} `iteration`
    Receive from output port o_r pairing with i in S_r: identity $\sigma(i)$ and weight $w_g(i \rightarrow \sigma(i))$;
    Receive from output port o_g pairing with i in S_g: identity $\sigma^{-1}(i)$ and weight $w_r(\sigma^{-1}(i) \rightarrow i)$;
    Compute knowledge set $\phi_{k+}^{(i)}$: $\phi_{k+}^{(i)} \leftarrow \{\sigma(i), w_r(i \rightarrow \sigma(i)), w_g(i \rightarrow \sigma(i))\}$;
    Compute knowledge set $\phi_{k-}^{(i)}$ similarly;
  else
    `// The subsequent iterations`
    $i_D \leftarrow \sigma^{2^{k-1}}(i)$;
    $i_U \leftarrow \sigma^{-2^{k-1}}(i)$;
    Send to i_U the knowledge set $\phi_{(k-1)+}^{(i)}$;
    Receive from i_D the knowledge set $\phi_{(k-1)+}^{(i_D)}$;
    Send to i_D the knowledge set $\phi_{(k-1)-}^{(i)}$;
    Receive from i_U the knowledge set $\phi_{(k-1)-}^{(i_U)}$;
    Compute knowledge set $\phi_{k}^{(i)}$ with $\phi_{(k-1)+}^{(i)}$ and $\phi_{(k-1)+}^{(i_D)}$ by (3.1);
    Compute knowledge set $\phi_{k-}^{(i)}$ similarly with $\phi_{(k-1)-}^{(i)}$ and $\phi_{(k-1)-}^{(i_U)}$;
  `// Halt checking`
  Halt if vertex i discovers any vertex twice (in light of newly discovered vertices $\sigma^{2^k}(i)$ and $\sigma^{-2^k}(i)$);

Algorithm 1: Knowledge-discovery procedure at vertex i.

exact MERGE operation (*i.e.*, making do with less).

3.3.2 Knowledge-Discovery Procedure

We now describe the $1+\log_2 N$ iterations of the knowledge-discovery procedure in detail and explain how these iterations allow every vertex i to concurrently obtain its knowledge sets $\phi_{k+}^{(i)}$ and $\phi_{k-}^{(i)}$ for $0 \leq k \leq \log_2 N$. The pseudocode of the knowledge-discovery procedure at vertex i is presented in Algorithm 1, that executed at any other vertex is identical.

3.3.2.1 *The 0^{th} Iteration*

We start with describing the 0^{th} iteration, the operation of which is slightly different than that of subsequent iterations in that whereas messages are exchanged only between input ports in all subsequent iterations, messages are also exchanged between input ports and output ports in the 0^{th} iteration. Suppose input port i is paired with output port o_r in the (red) full matching S_r and with output port o_g in the (green) full matching S_g. The 0^{th} iteration contains two rounds of message exchanges. In the first round, input port i receives from output port o_r a message which includes the identity $\sigma(i)$ and the weight of edge $o_r \rightarrow \sigma(i)$, *i.e.,* the green weight of edge $i \rightarrow \sigma(i)$ (Line 3). Note that input port $\sigma(i)$ is paired with o_r in S_g, so o_r knows the identity of $\sigma(i)$ and the weight of edge $o_r \rightarrow \sigma(i)$. With the newly received information, input port i can calculate knowledge set $\phi_{0+}^{(i)}$ (Line 5). For example, in this round, input port 3 in Figure 3.1 receives from output port 1 the identity of input port 4 and the weight, which is 5, of edge $O_1 \rightarrow I_4$. Similarly, in the second round, input port i receives from output port o_g a message which includes the identity $\sigma^{-1}(i)$ and the weight of edge $\sigma^{-1}(i) \rightarrow o_g$, *i.e.,* the red weight of edge $\sigma^{-1}(i) \rightarrow i$ (Line 4). Note that input port $\sigma^{-1}(i)$ is paired with o_g in S_r. With the newly received information, input port i can calculate knowledge set $\phi_{0-}^{(i)}$ (Line 6). For example, in this round, input port 3 in Figure 3.1 receives from output port O_4 the identity of input port 12 and the weight, which is 0, of edge $I_{12} \rightarrow O_4$. Therefore, after this 0^{th} iteration, each input port i obtains the knowledge sets $\phi_{0+}^{(i)}$ and $\phi_{0-}^{(i)}$, or in other words, discovers $\sigma(i)$ and $\sigma^{-1}(i)$.

3.3.2.2 *Subsequent Iterations*

The subsequent iterations can be described inductively as follows. Suppose after iteration $k-1$ (for any $1 \leq k \leq \log_2 N$), every vertex i discovers its upstream vertex $i_U = \sigma^{-2^{k-1}}(i)$ and its downstream vertex $i_D = \sigma^{2^{k-1}}(i)$. Lines 10-15 in Algorithm 1 show how vertex i discovers $\sigma^{2^k}(i)$ and $\sigma^{-2^k}(i)$ via two rounds of message exchanges of the iteration k. In the first round, vertex i sends the knowledge set $\phi_{(k-1)+}^{(i)}$, obtained during iteration $k-1$, to the

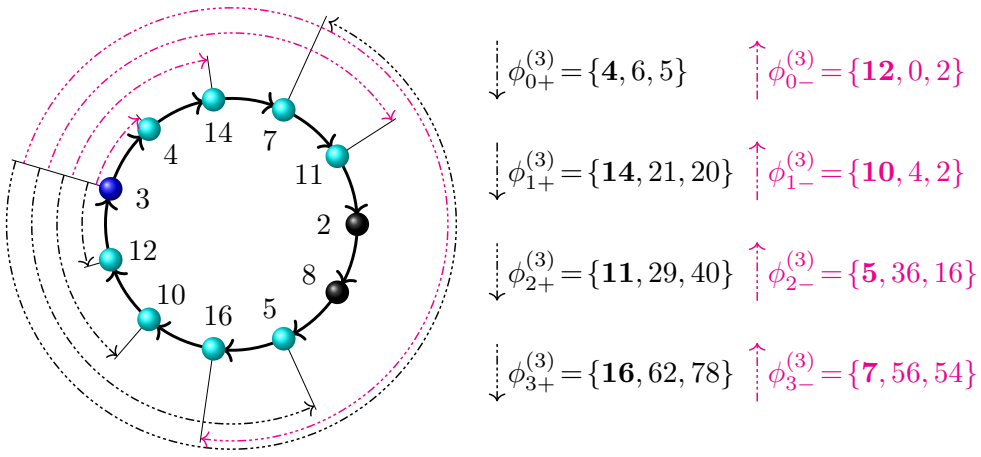


Figure 3.3: Illustration of the knowledge-discovery procedure: messages sent by vertex 3 in Figure 3.2.

upstream vertex i_U (Line 10). Meanwhile, vertex i receives from the downstream vertex i_D its knowledge set $\phi^{(i_D)}_{(k-1)+}$ (Line 11), which, as explained earlier, contains the values of $\sigma^{2^{k-1}}(i_D)$, $w_r(i_D \rightsquigarrow \sigma^{2^{k-1}}(i_D))$, and $w_g(i_D \rightsquigarrow \sigma^{2^{k-1}}(i_D))$. Having obtained these three values, vertex i pieces together its knowledge set $\phi^{(i)}_{k+}$ (Line 14) as follows.

$$\begin{cases} \sigma^{2^k}(i) & \leftarrow \sigma^{2^{k-1}}\big(i_D\big) \\ w_r\big(i \rightsquigarrow \sigma^{2^k}(i)\big) & \leftarrow w_r\big(i \rightsquigarrow \sigma^{2^{k-1}}(i)\big) + w_r\big(i_D \rightsquigarrow \sigma^{2^{k-1}}(i_D)\big) \\ w_g\big(i \rightsquigarrow \sigma^{2^k}(i)\big) & \leftarrow w_g\big(i \rightsquigarrow \sigma^{2^{k-1}}(i)\big) + w_g\big(i_D \rightsquigarrow \sigma^{2^{k-1}}(i_D)\big) \end{cases} \tag{3.1}$$

Note that vertex i already knows $\phi^{(i)}_{(k-1)+}$, which includes $w_r(i \rightsquigarrow \sigma^{2^{k-1}}(i))$ and $w_g(i \rightsquigarrow \sigma^{2^{k-1}}(i))$.

Similarly, in the second round of message exchanges, vertex i sends $\phi^{(i)}_{(k-1)-}$ to the downstream vertex i_D (Line 12), and meanwhile receives $\phi^{(i_U)}_{(k-1)-}$ from the upstream vertex i_U (Line 13). The latter knowledge set (*i.e.*, $\phi^{(i_U)}_{(k-1)-}$), combined with the knowledge set $\phi^{(i)}_{(k-1)-}$ that vertex i already knows, allows i to piece together the knowledge set $\phi^{(i)}_{k-}$. Therefore, vertex i obtains $\phi^{(i)}_{k+}$ and $\phi^{(i)}_{k-}$, or in other words discovers $\sigma^{2^k}(i)$ and $\sigma^{-2^k}(i)$, after the k^{th} iteration.

An Illustrative Example. Figure 3.3 shows the messages sent by vertex 3 in Figure 3.2 during the k^{th} $(1 \leq k \leq 4)$ iteration of the knowledge-discovery procedure. For

example, in the 3^{rd} iteration, vertex 3 sends to vertex $11 = \sigma^4(3)$ the knowledge set $\phi_{2-}^{(3)} = \{\sigma^{-4}(3), w_r(\sigma^{-4}(3) \rightsquigarrow 3), w_g(\sigma^{-4}(3) \rightsquigarrow 3)\} = \{\mathbf{5}, 36, 16\}$ (the 3^{rd} up arrow from top to bottom in right half of Figure 3.3) it learns during the 2^{nd} iteration. It also sends to vertex $5 = \sigma^{-4}(3)$ the knowledge set $\phi_{2+}^{(3)} = \{\sigma^4(3), w_r(3 \rightsquigarrow \sigma^4(3)), w_g(3 \rightsquigarrow \sigma^4(3))\} = \{\mathbf{11}, 29, 40\}$ (the 3^{rd} down arrow from top to bottom in right half of Figure 3.3). Though it is not shown in Figure 3.3, in the same iteration, vertex 3 receives $\phi_{2+}^{(11)} = \{\sigma^4(11), w_r(11 \rightsquigarrow \sigma^4(11)), w_g(11 \rightsquigarrow \sigma^4(11))\} = \{\mathbf{16}, 33, 38\}$ from vertex $11 = \sigma^4(3)$ so that it can compute, by (3.1), $\sigma^8(3) = 16$, $w_r(3 \rightsquigarrow \sigma^8(3)) = w_r(3 \rightsquigarrow \sigma^4(3)) + w_r(11 \rightsquigarrow \sigma^4(11)) = 29 + 33 = 62$, and $w_g(3 \rightsquigarrow \sigma^8(3)) = w_g(3 \rightsquigarrow \sigma^4(3)) + w_g(11 \rightsquigarrow \sigma^4(11)) = 40 + 38 = 78$, which is precisely $\phi_{3+}^{(3)}$. It also receives $\phi_{2-}^{(5)} = \{\sigma^{-4}(5), w_r(\sigma^{-4}(5) \rightsquigarrow 5), w_g(\sigma^{-4}(5) \rightsquigarrow 5)\}$ from vertex $5 = \sigma^{-4}(3)$. Similarly, it can compute $\phi_{3-}^{(3)} = \{\sigma^{-8}(3), w_r(\sigma^{-8}(3) \rightsquigarrow 3), w_g(\sigma^{-8}(3) \rightsquigarrow 3)\} = \{\mathbf{7}, 56, 54\}$. Therefore, vertex 3 discovers vertices $16 = \sigma^8(3)$ and $7 = \sigma^{-8}(3)$ respectively. Note in this example and Figure 3.3, numerical green and red weights in knowledge sets are not in bold.

3.3.2.3 Early Halt Checking

The knowledge-discovery procedure might halt before finishing the $1+\log_2 N$ iterations. As shown in Line 16, the procedure will halt if vertex i discovers some vertex twice. More precisely, if vertex i discovers the same vertex twice in the same iteration, *i.e.,* $\sigma^{2^k}(i) = \sigma^{-2^k}(i)$ or it discovers a vertex that has been discovered in the previous iterations, *i.e.,* vertex i has already discovered $\sigma^{2^k}(i)$ (or $\sigma^{-2^k}(i)$) in the previous iterations. By Lemma 3.2.1, we conclude that vertex i can make the exact same matching decision as it would under SERENA.

Halt Checking in $O(1)$ Per Iteration. Vertex i can finish this halt checking in $O(1)$ (per port) per iteration, or $O(\log N)$ in total, using a pointer array $B[1..N]$. Here, we only need to show that the latter case described above, *i.e.,* checking whether vertex i has already discovered $\sigma^{2^k}(i)$ (or $\sigma^{-2^k}(i)$) in the previous iterations, in $O(1)$, as the checking for the former case (*i.e.,* whether $\sigma^{2^k}(i) = \sigma^{-2^k}(i)$) is obviously $O(1)$. Each array entry $B[i']$ initially points to `NULL`. At the end of each iteration (including the 0^{th} iteration), vertex i

simply checks whether $B[\sigma^{2^k}(i)] \neq \texttt{NULL}$ (or $B[\sigma^{-2^k}(i)] \neq \texttt{NULL}$). If so, then $\sigma^{2^k}(i)$ (or $\sigma^{-2^k}(i)$) has been discovered in the previous iterations. Otherwise, we update B as follows: pointing $B[\sigma^{2^k}(i)]$ and $B[\sigma^{-2^k}(i)]$ to the knowledge set $\phi_{k+}^{(i)}$ and $\phi_{k-}^{(i)}$ respectively.

Note that we need to reset the values of all N entries of B to NULL at the end of a matching computation. The time complexity of the reset is $O(\log N)$ (instead of $O(N)$) because each non-null entry of B is indexed by the identity field of a knowledge set, the total number of which is upper-bounded by $2+2\log_2 N$. Hence the total time complexity for each vertex i to finish halt checking, *i.e.,* Line 16 of Algorithm 1, is $O(1)$ (per port) per iteration, or $O(\log N)$ in total.

All or None Lemma. Using the similar operations as in the proof of Lemma 3.2.1, vertex i can use $O(1)$ operations to decide which is heaver between the green weight and the red weight of the cycle that vertex i belongs to. So can other vertices belonging to the same cycle (as vertex i) by using the following lemma.

Lemma 3.3.1 (All or None)**.** During the execution of the knowledge-discovery procedure in SERENADE, if any vertex i halts before finishing the $1+\log_2 N$ iterations, *i.e.,* halting because of discovering some vertex twice in Line 16 of Algorithm 1, then all other vertices belonging to the same cycle will also halt in the same iteration

Proof: See §A.3. ■

3.3.2.4 Discussions

In describing the knowledge-discovery procedure, we assume that input ports can communicate directly with each other. This is a realistic assumption, because in most real-world switch products, each line card i is full-duplex in the sense the logical input port i and the logical output port i are co-located in the same physical line card i. In this case, for example, an input port i_1 can communicate with another input port i_2 by sending information to output port i_2, which then relays it to the input port i_2 through the "local bypass",

presumably at little or no communication costs. However, SERENADE also works for the type of switches that do not have such a "local bypass," by letting an output port to serve as a relay, albeit at twice the communication costs. More precisely, in the example above, the input port i_1 can send the information first to the output port i_1, which then relays the information to the input port i_2.

3.3.3 Complexity Analysis

We now analyze the time and message complexities of the knowledge-discovery procedure.

Time Complexity. The time complexity of the knowledge-discovery procedure is (at most) $1+\log_2 N$ iterations, and that of each iteration is several operations for local computation for computing knowledge sets (Lines 14-15 of Algorithm 1) and halt checking (Line 16 of Algorithm 1). Clearly, those operations can be performed in $O(1)$.

Message Complexity. The message complexity of the knowledge-discovery procedure is $O(\log N)$ messages per vertex, since every vertex needs to send (and receive) two messages during each iteration. In every message, it suffices to only include $w_r(\cdot)-w_g(\cdot)$, the difference between the red and the green weights of the corresponding walk. Therefore, each message (*i.e.,* knowledge set) can be encoded in $C+\log_2 N$ bits, where C is the maximum number of bits needed to encode this difference.

3.3.4 Early Halt: The Ouroboros Cycles

In this section, we define the concept of an *ouroboros cycle*, and prove Lemma 3.3.2, which states that all vertices on an ouroboros cycle can halt (Line 16 of Algorithm 1) and make the exact same matching decisions as they would under SERENA, without performing the distributed binary search. Ouroboros is the ancient Greek symbol depicting a serpent devouring its own tail. We "borrow" this concept because what happens in ouroboros cycles is very similar to what is depicted by the symbol "Ouroboros".

Definition 3.3.1 (Ouroboros Cycle)**.** A cycle is said to be *ouroboros* if and only if its length ℓ is an *ouroboros number* (*w.r.t.* N), defined as a positive divisor of a number that takes one of the following three forms: (I) 2^α, (II) $2^\beta - 2^\gamma$, and (III) $2^\beta + 2^\gamma$, where α, β and γ are nonnegative integers that satisfy $\alpha \leq \lceil \log_2 N \rceil$ and $\gamma < \beta \leq \lceil \log_2 N \rceil$.

It is not hard to check that, in Figure 3.2, the leftmost cycle (of length 11) is not ouroboros (*i.e.*, non-ouroboros), but the other two are.

The following lemma shows a nice property of ouroboros cycles, whose proof can be found in §A.4.

Lemma 3.3.2 (Ouroboros Lemma)**.** Vertex i will discover twice a vertex on the same cycle (as itself) during the knowledge-discovery stage, if it is on an ouroboros cycle.

Remark. Readers may wonder if we can do away with the distributed binary search simply by running a little more iterations (say $0.5 \log_2 N$ more iterations), because more iterations means that more vertices may discover a vertex twice. Unfortunately, as shown in §A.5, there exists some numbers (cycle lengths) that are "hardcore non-ouroboros" in the sense a vertex i on a cycle of such a length ℓ needs to run exactly $\lceil \ell/2 \rceil$ iterations to discover a vertex twice.

3.4 Leader Election

We have shown that the $1 + \log_2 N$ iterations of the knowledge-discovery procedure alone is not enough for SERENADE to emulate SERENA exactly. To do so, SERENADE needs an additional distributed binary search. As mentioned in §3.2, the distributed binary search requires every non-ouroboros cycle to elect a designated vertex, which is decided through a leader election by vertices on this cycle. In this section, we describe how to embed this leader election seamlessly into the knowledge-discovery procedure of SERENADE.

3.4.1 Leader Election

We explain this process on an arbitrary combinatorial cycle of σ, focusing on the actions of an arbitrary vertex i that belongs to this cycle. We follow the standard practice [66] of making the vertex with the smallest identity (an integer between 1 and N) on this cycle the leader. Recall that in the knowledge-discovery procedure, after each (say k^{th}) iteration, vertex i discovers $\sigma^{-2^k}(i)$ that is "2^k σ-hops away" from it on the cycle. More precisely, vertex i learns $\phi_{k-}^{(i)}$, which contains the identities of the vertex $\sigma^{-2^k}(i)$, and the red and green weights of the walk $\sigma^{-2^k}(i) \rightsquigarrow i$. Our goal is to augment this k^{th} iteration to learn the vertex with the smallest identity on this walk $\sigma^{-2^k}(i) \rightsquigarrow i$, which we denote as $\mathcal{L}(\sigma^{-2^k}(i) \rightsquigarrow i)$ and call the leader of the level-k precinct right-ended at i.

Like in the knowledge-discovery procedure, we explain this augmentation inductively. The case of $k=0$ (*i.e.*, the 0^{th} iteration) is as follows: Each vertex i considers the one with smaller identity between itself and $\sigma^{-1}(i)$ to be the leader of the level-0 precinct right-ended at i.

For the k^{th} ($k \geq 1$) iteration, each vertex i only needs to augment the knowledge set it sends downstream to $i_D = \sigma^{2^{k-1}}(i)$ with $\mathcal{L}(\sigma^{(-2^{k-1})}(i) \rightsquigarrow i)$ (the leader of the level-$(k-1)$ precinct right-ended at i) in Line 12 of Algorithm 1. Meanwhile, it receives, from $i_U = \sigma^{(-2^{k-1})}(i)$, the vertex 2^{k-1} σ-hops upstream, $\mathcal{L}(\sigma^{(-2^{k-1})}(i_U) \rightsquigarrow i_U)$ (the leader of the level-$(k-1)$ precinct right-ended at i_U) in Line 13 of Algorithm 1, as i_U also augments its knowledge set. In addition, each vertex i also adds the following local computation in Line 15 of Algorithm 1.

$$\mathcal{L}(\sigma^{-2^k}(i) \rightsquigarrow i) \leftarrow \min\left\{\mathcal{L}(\sigma^{(-2^{k-1})}(i_U) \rightsquigarrow i_U), \mathcal{L}(\sigma^{(-2^{k-1})}(i) \rightsquigarrow i)\right\}$$

The following lemma concerns the correctness of and the minimum number of iterations (*i.e.,* $1+\log_2 N$) required by the above embedded leader election. This is also the reason why we choose to execute the knowledge-discovery procedure for $1+\log_2 N$ iterations. Its proof is straightforward, we omit it in this thesis.

Lemma 3.4.1. Given any non-ouroboros cycle, each vertex belonging to it will learn, through the augmented knowledge-discovery procedure, the identity of the leader for the cycle, after at most $1 + \log_2 N$ iterations. Besides, there exists some non-ouroboros cycle such that some vertices belonging to it need at least $1 + \log_2 N$ iterations to learn the identity of the leader.

3.4.2 Distribute Leaders' Decisions

Once the leader of a non-ouroboros cycle is decided, through a distributed binary search (to be described in §3.5), the leader will discover itself (through a non-empty walk). According to Lemma 3.2.1, the leader now can make the same matching decision as it would under SERENA. Then, the leader informs the switch controller of its decision on whether to choose the green or the red sub-matching, and the switch controller then broadcasts decisions of all leaders to the N vertices. Since each vertex on a non-ouroboros cycle knows the identity of its leader by Lemma 3.4.1, it will follow the decision made by its leader in choosing between the red and the green sub-matchings.

The size of this broadcast, equal to the number of *non-ouroboros* cycles in σ, is small (with overwhelming probability). For example, we will show in §A.8.1 that even when $N = 256$, the average number of non-ouroboros cycles is no more than 1.69 and in more than 99% of instances, there are no more than 4 non-ouroboros cycles per time slot.

3.5 Distributed Binary Search Stage

As mentioned above, only vertices on ouroboros cycles can make the exact same decisions as they would under SERENA, for vertices on non-ouroboros cycles, SERENADE needs an additional distributed binary search stage. In this section, we will describe the binary search stage focusing on an arbitrary non-ouroboros cycle.

```
1  Procedure BinarySearch(i, k, w_g, w_r)
2      if i (self) is L_0 then halt;
3      if σ^(-2^(k-1))(i) = L_0 then
4          w_g ← w_g − w_g(σ^(-2^(k-1))(i) ⇝ i);
5          w_r ← w_r − w_r(σ^(-2^(k-1))(i) ⇝ i);
6          BinarySearch(L_0, k−1, w_g, w_r);
7      else
8          if L_0 = L(σ^(-2^(k-1))(i) ⇝ i) then
9              BinarySearch(i, k − 1, w_g, w_r);
10         else
11             w_g ← w_g − w_g(σ^(-2^(k-1))(i) ⇝ i);
12             w_r ← w_r − w_r(σ^(-2^(k-1))(i) ⇝ i);
13             BinarySearch(σ^(-2^(k-1))(i),k−1,w_g,w_r);
```

Algorithm 2: Distributed binary search at vertex i.

3.5.1 Distributed Binary Search

Without loss of generality, we assume that $\mathcal{L}_0$ is the leader of, and i a vertex on, this non-ouroboros cycle. The objective of this distributed algorithm is to let its leader $\mathcal{L}_0$ discovers itself twice by searching a repetition (*i.e.,* other than its first occurrence as the starting point of the walk) of $\mathcal{L}_0$ along the walk $\mathcal{L}_0 \rightsquigarrow \sigma^N(\mathcal{L}_0)$, the level-$(\log_2 N)$ precinct right-ended at $\sigma^N(\mathcal{L}_0)$; this repetition must exist because N, the length of the walk $\mathcal{L}_0 \rightsquigarrow \sigma^N(\mathcal{L}_0)$, is no smaller than the length of this cycle. To this end, vertices on this non-ouroboros cycle perform a distributed binary search, guided by the leadership information each vertex obtains through the leader election. In the following, we describe the high-level ideas of this binary search algorithm, in which the detailed actions of a vertex i are captured by Algorithm 2. Unlike the knowledge-discovery procedure, during each iteration of the distributed binary search, only one vertex on this non-ouroboros cycle performs the search task, which we call *the search administrator*.

High-Level Ideas. This binary search is initiated by the vertex $\sigma^N(\mathcal{L}_0)$, who learns "who herself is" (*i.e.,* that herself is $\sigma^N(\mathcal{L}_0)$) during the last iteration of the augmented knowledge-discovery procedure; in other words, the initial *search administrator* is $\sigma^N(\mathcal{L}_0)$.

The initial *search interval* is the entire walk $\mathcal{L}_0 \rightsquigarrow \sigma^N(\mathcal{L}_0)$, also the level-$(\log_2 N)$ precinct right-ended at $\sigma^N(\mathcal{L}_0)$. The search administrator $\sigma^N(\mathcal{L}_0)$ first checks whether itself or $\sigma^{N/2}(\mathcal{L}_0)$, the middle point of the search interval, is a repetition of $\mathcal{L}_0$. If so, the entire search mission is accomplished, so the search ends. Otherwise, it checks whether there is a repetition of $\mathcal{L}_0$ in the right half of the search interval by checking whether $\mathcal{L}(\sigma^{N/2}(\mathcal{L}_0) \rightsquigarrow \sigma^N(\mathcal{L}_0))$ is equal to $\mathcal{L}_0$; note the identity of $\mathcal{L}(\sigma^{N/2}(\mathcal{L}_0) \rightsquigarrow \sigma^N(\mathcal{L}_0))$, the leader of the level-$(\log_2(N) - 1)$ precinct right-ended at $\sigma^N(\mathcal{L}_0)$, is known to $\sigma^N(\mathcal{L}_0)$, since it is one of the leadership information $\sigma^N(\mathcal{L}_0)$ learns through the leader election. If so, the same search administrator $\sigma^N(\mathcal{L}_0)$ carries on this binary search in the right half of the search interval. Otherwise, the middle point of the search interval $\sigma^{N/2}(\mathcal{L}_0)$ becomes the new search administrator and carries on this binary search in the left half.

Pseudocode Explanation. As mentioned above, the detailed actions of any search administrator i are captured by Algorithm 2. Initially i is $\sigma^N(\mathcal{L}_0)$ who assigns $\log_2 N$ to k, the 2^{nd} argument of Algorithm 2, which indicates the search interval is $\sigma^{-2^k} \rightsquigarrow i$. To ensure that $\mathcal{L}_0$ can discover itself, *i.e.,* learning the green and red weights of a non-empty walk from $\mathcal{L}_0$ to itself, search administrator i also maintains the weight information w_g, w_r (the 3^{rd} and 4^{th} arguments of Algorithm 2), which are the green and red weights of the walk from $\mathcal{L}_0$ to i. Initially, search administrator $\sigma^N(\mathcal{L}_0)$ knows the green and red weights of the walk $\mathcal{L}_0 \rightsquigarrow \sigma^N(\mathcal{L}_0)$, because they belong to the knowledge sets that $\sigma^N(\mathcal{L}_0)$ learns during the last iteration of the augmented knowledge-discovery procedure. We will not describe Algorithm 2 line-by-line, since the actions of search administrator i are the same as those of the initial search administrator we described above, except that Algorithm 2 details the operations for bookkeeping weight information.

The correctness of the distributed binary search and the number of iterations it requires, which are summarized in the following lemma, can be proved with mathematical induction. Here, we omit it for brevity.

Lemma 3.5.1. Given any non-ouroboros cycle with a length of ℓ, the distributed binary

search enables the leader $\mathcal{L}_0$ of this cycle to discover itself twice with at most $\lceil \log_2 \ell \rceil$ iterations.

3.5.2 Complexity Analysis

In this section, we analyze the time and message complexities of the distributed binary search.

Time Complexity. The time complexity (*i.e.,* number of iterations) of the distributed binary search is upper-bounded by $\lceil \log_2 \eta \rceil$ ($\leq \log_2 N$) iterations, where η is the length of the longest non-ouroboros cycle, since binary searches at different non-ouroboros cycles are performed simultaneously. Hence, it can be as large as $\log_2 N$ iterations, each of which has $O(1)$ time complexity.

Message Complexity. As explained in §3.5.1, during the binary search, on each non-ouroboros cycle a message (to maintaining the weight information) is transmitted only when the search administrator moves from one vertex to another, so the message complexity of the binary search, in the worst-case, is at most 1 message per vertex. Each message needs $\lceil \log_2 \log_2 N \rceil + C$ bits, where C is the number of bits for encoding $w_r(\cdot) - w_g(\cdot)$.

3.6 Early Stop: O-SERENADE

In this section, we present an early-stop version of SERENADE to approximately emulate SERENA, without performing the distributed binary search, in which vertices on any non-ouroboros cycle make a decision based on the (insufficient) information at hand after the augmented knowledge-discovery procedure. As will be shown in §3.7, O-SERENADE trades no degradation of delay performances for significant reduction in time complexities (*i.e.,* without performing the distributed binary search that has up to $\log_2 N$ iterations).

Decision Rule. Now we describe the decision rule of this early-stop version in an arbitrary non-ouroboros cycle. The decision rule is for the leader $\mathcal{L}_0$ of this cycle to compare the

green and the red weights of the longest such walk $\mathcal{L}_0 \rightsquigarrow \sigma^N(\mathcal{L}_0)$, and pick, on behalf of the whole cycle, the green or the red sub-matching according to the outcome of this comparison; note we cannot simply let every vertex i on this cycle to pick the green or the red edge individually based on its local view of $w_r\big(i \rightsquigarrow \sigma^N(i)\big)$ vs. $w_g\big(i \rightsquigarrow \sigma^N(i)\big)$, since these local views can be inconsistent. For example, for the leftmost cycle in Figure 3.2, it is not hard to check vertex 2 and vertex 3 have inconsistent local views: $w_g\big(2 \rightsquigarrow \sigma^{16}(2)\big) = 120 < 134 = w_r\big(2 \rightsquigarrow \sigma^{16}(2)\big)$ (vertex 2's local view) and $w_g\big(3 \rightsquigarrow \sigma^{16}(3)\big) = 130 > 112 = w_r\big(3 \rightsquigarrow \sigma^{16}(3)\big)$ (vertex 3's local view). It is clear that O-SERENADE will pick the red sub-matching in this cycle based on the local view of its leader (*i.e.,* vertex 2), which is different from the decision under SERENA. This strategy is opportunistic as it does not hesitate to pick a sub-matching that appears to be larger, even though there is a small chance this appearance is incorrect. Therefore, we call it O-SERENADE. Like SERENADE, this early-stop version also needs the switch controller to broadcast the decisions of all the leaders to the N vertices.

Rationale. The rationale behind the opportunistic strategy is as follows. When the weight difference between the red and the green sub-matchings is small, it matters little which sub-matching is picked. When the difference is large, however, this strategy likely will further inflate the already large difference and hence result in the correct sub-matching being picked.

3.7 Performance Evaluation

In this section, we evaluate, through simulations, the throughput and delay performances of O-SERENADE under various load conditions and traffic patterns specified in §1.2.6; there is no need to evaluate the throughput and delay performances of SERENADE, which exactly emulates SERENA. Note that we have also evaluated the message complexity of SERENADE, and investigated how the mean delay performance of O-SERENADE scales with respect to N, the number of (input/output) ports; these results can be found in §A.8.

3.7.1 Simulation Setup

In all our simulations, the number of input/output ports N is 64, unless otherwise stated. To measure throughput and delay accurately, we assume each VOQ has an infinite buffer size and hence there is no packet drop at any input port. Every simulation run lasts $30{,}000 \times N^2$ time slots. This duration is chosen so that every simulation run enters the steady state after a tiny fraction of this duration and stays there for the rest. The throughput and delay measurements are taken after the simulation run enters the steady state.

Like in [4, 10], we assume, in the following simulations, that the traffic arrival processes to different input ports are mutually independent, and each such arrival process is *i.i.d.* Bernoulli (*i.e.,* at any given input port, a packet arrives with a constant probability $\rho \in (0, 1)$ during each time slot). Note that we only use synthetic traffic (instead of that derived from packet traces) because, to the best of our knowledge, there is no meaningful way to combine packet traces into switch-wide traffic workloads. As mentioned earlier in §1.2.6, the four standard types of traffic patterns are used to generate the workloads of the switch. Finally, we emphasize that, every non-zero diagonal element (*i.e.*, traffic from an input port i and an output port i), in every traffic matrix we simulated on, is actually *switched* by the crossbar and consumes just as much switching resources per packet as other traffic matrix elements, and never takes advantage of the "local bypass" (see §3.3.2.4) that may exist between the input port i and the output port i.

3.7.2 Throughput Performance

Our simulation results show that O-SERENADE can achieve close to 100% throughput under all 4 traffic patterns and *i.i.d.* Bernoulli traffic arrivals: The VOQ lengths remain stable under an offered load of 0.99 in all these simulations.

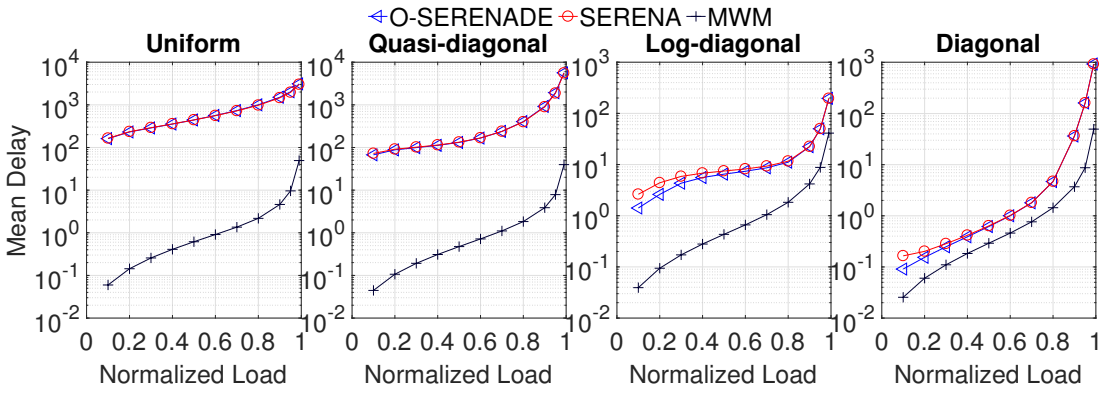


Figure 3.4: Mean delays of O-SERENADE, SERENA and MWM under the 4 traffic patterns.

3.7.3 Delay Performance

Now we shift our focus to the delay performance of O-SERENADE. We compare its delay performance only with those of SERENA and MWM. We refer readers to [4, 10] for comparisons between SERENA and some other crossbar scheduling algorithms such as iSLIP [5] and iLQF [36].

O-SERENADE vs. SERENA. Figure 3.4 shows the mean delays of the three algorithms under the 4 traffic patterns above respectively. Each subfigure shows how the mean delays (on a *log scale* along the y-axis) vary with different offered loads (along the x-axis). Figure 3.4 shows that overall O-SERENADE and SERENA perform similarly under all 4 traffic patterns and all load factors. Upon observing these simulation results, our interpretation was that the decisions made by O-SERENADE agree with the ground truth (*i.e.,* which sub-matching is indeed heavier on a non-ouroboros cycle) most of time. This interpretation was later confirmed by further simulations: They agree in between 90.57% and 99.99% of the instances.

Perhaps surprisingly, Figure 3.4 also shows that O-SERENADE performs slightly better than SERENA when the traffic load is low (say < 0.4) under log-diagonal and diagonal traffic patterns. Our interpretation of this observation is as follows. It is not hard to verify that decisions made by O-SERENADE can disagree with the ground truth, with a

non-negligible probability, only when the total green and the total red weights of a non-ouroboros cycle are very close to one another. However, in such cases, picking the wrong sub-matchings (*i.e.,* disagreeing with the ground truth) causes almost no damages. Furthermore, we speculate that it may even help O-SERENADE jump out of a local maximum (*i.e.,* have the effect of simulated annealing) and converge more quickly to a near-optimal matching (in terms of weight), thus resulting in even better delay performance.

3.8 Conclusion

In this chapter, we propose SERENADE, a parallel iterative algorithm that can provably, with a time complexity of only $O(\log N)$ per port, exactly emulate SERENA, a centralized algorithm with $O(N)$ time complexity. We also propose an early-stop version of SERENADE, called O-SERENADE, which only approximately emulates SERENA. Through extensive simulations, we demonstrate that O-SERENADE can achieve close to 100% throughput. We also demonstrate that O-SERENADE has delay performances either similar as or better than those of SERENA, under various load conditions and traffic patterns.

CHAPTER 4

QPS

4.1 Queue-Proportional Sampling (QPS)

In this section, we first describe the QPS proposing strategy in detail. Then we explain how to augment iSLIP [5] and SERENA [4, 10] using QPS. We next compare QPS with ShakeUp [67], another "add-on" technique that can be used to augment iterative crossbar scheduling algorithms such as iSLIP and iLQF [36]. In §B.1, we discuss a QPS variant called FQPS, which samples a VOQ with a probability proportional to a function of the VOQ length.

4.1.1 The QPS Proposing Strategy

In all QPS-augmented crossbar scheduling algorithms, the first step is for input ports and output ports to perform one iteration of message exchanges to generate a starter matching. This iteration consists of two phases, namely, a proposing phase and an accepting phase.

```
1 Procedure QPS-Propose()
2     Sample an output port j with probability m_j / m
3     Send m_j (length of VOQ j) to output port j
```

Algorithm 3: Proposing phase at input port 1.

Proposing Phase. In this phase, each input port proposes to exactly one output port – decided by the QPS strategy – unless it has no packet to transmit. Algorithm 3 shows the pseudocode of the QPS proposing strategy at input port 1; that at any other input port is identical. Denote as m_1, m_2, $\cdots$, m_N the respective lengths of N VOQs at input port 1, and as m their total (*i.e.*, $m \triangleq \sum_{k=1}^{N} m_k$). Input port 1 simply samples an output port j with probability m_j/m (Line 2), *i.e.,* proportional to the length of the corresponding VOQ;

it then proposes to output port j (Line 3), with the value m_j that will be used in the next phase.

```
1 Procedure Accept()
2     if one or more proposals are received then
3         Accept the one with largest VOQ length
```

Algorithm 4: Accepting phase at output port 1.

Accepting Phase. We adopt the same accepting strategy as in SERENA: "longest VOQ first". The pseudocode of the accepting phase, at output port 1, is shown in Algorithm 4; that at any other output port is identical. The action of output port 1 depends on the number of proposals it receives. If it receives exactly one proposal from an input port, it will accept the proposal and (tentatively) match with the input port. However, if it receives proposals from multiple input ports, it will accept the proposal accompanied with the highest VOQ length, with ties broken uniformly at random.

The time complexity of this accepting strategy is $O(1)$ in practice although in theory an output port could receive up to N proposals and have to compare their accompanying VOQ lengths. This is because the probability for an output port to receive proposals from more than several (say 5) input ports is tiny, and even if this rare event happens, the output port can ignore/drop all proposals beyond the first several (say 5) without affecting the quality of the final matching much. In our evaluations, we indeed set this threshold to 5.

We have also considered and experimented with another accepting strategy: accepting each competing proposal with a probability proportional to the length of the corresponding VOQ, which we refer to as *Proportional Accepting* (PA). The advantage of PA over "longest VOQ first" above is that when the switch is severely overloaded (*i.e.,* with offered load $> 100\%$), PA could provide better fairness to competing input ports and help prevent certain starvation situations. For example, consider the pathological scenario in which, for a fairly long period of time (say 1 minute), packets destined for an output j would arrive at input ports i_1 and i_2 with rates 1.2 and 0.1 respectively. Under "longest VOQ first", the output

port j would keep accepting proposals from input port i_1 (because its VOQ length is longer, assuming the two respective VOQs are empty before this period) and hence starve input port i_2, whereas under PA, the output port j would accept proposals from input port i_2 with roughly $1/13$ probability.

However, we prefer "longest VOQ first" over PA because, as we will show in §B.5.3, the former generally has better mean delay performance, albeit slightly, and guarantees almost the same fairness and lack of starvation, under all *admissible* workloads. We believe the primary mission of a crossbar scheduling algorithm is to deliver excellent performance under *admissible* workloads; such "grace under fire" (proportional fairness and lack of starvation even when severely overloaded) is a secondary consideration and can be better achieved through other "knobs or levers" orthogonal to switching such as congestion control, packet scheduling, or traffic policing/shaping. This said, we prove in §B.4 that QPS-SERENA with PA can also achieve 100% throughput just like QPS-SERENA with "longest VOQ first", in case the former is preferred in certain application scenarios.

Message Complexity. The message complexity of each "propose-accept" iteration is $O(1)$ messages per input or output port, because each input/output port transmits no more than one message during the propose/accept phase.

4.1.2 Augmenting iSLIP and SERENA

Now we describe, in QPS-iSLIP and QPS-SERENA respectively, how iSLIP and SERENA are augmented using QPS. We also describe iLQF [36] in this section, because it is closely related to iSLIP, and its performance will be compared against QPS-iSLIP in §4.4.

4.1.2.1 iSLIP, QPS-iSLIP, and iLQF

The iSLIP algorithm is a parallel iterative algorithm that computes an approximate MCM (Maximum Cardinality Matching) via multiple iterations of message exchanges between the input and output ports. Each iteration consists of three stages: request, grant, and

accept. In the request stage, each input port sends requests to all output ports whose corresponding VOQs are not empty. In the grant stage, each output port, upon receiving requests from multiple input ports, grants to one in a round-robin order. This round-robin order is enforced through a *grant pointer* that records the identifier of the input port – to whom a grant was accepted in the first iteration – during the most recent time slot when this situation occurred. Finally, in the accept stage, each input port, upon receiving accepts from multiple output ports, accepts one in a round-robin order, enforced similarly through an *accept pointer*.

QPS-iSLIP can be viewed as adding a "0^{th} iteration" to iSLIP. In this 0^{th} iteration, QPS is executed to generate a starter matching. Then iSLIP is called to match only those input/output ports not matched in the 0^{th} iteration, through multiple request-grant-accept iterations. We specify that in QPS-iSLIP, it is those ports matched in the 1^{st} iteration (by iSLIP), not those matched in the 0^{th} iteration (by QPS), who update the values of their grant or accept pointers. The rationale is that the aforementioned objective of enforcing the round-robin order is not accomplished in the QPS iteration.

iLQF [36] operates in the same way as iSLIP, except that (1) it is aware of the edge weights (*i.e.,* lengths of VOQs), and (2) it favors the request or grants with the heaviest weight (*i.e.,* greedy) in the grant or accept stage respectively. Hence, iLQF can be viewed as a greedy approach to approximately compute the MWM. iLQF generally performs better than iSLIP, but has a higher time complexity of $O(N)$ per port (compared to $O(\log^2 N)$ for iSLIP). We show in §4.4 that our QPS-iSLIP algorithm has a similar performance as iLQF, but the same per-port complexity as iSLIP.

4.1.2.2 SERENA and QPS-SERENA

As described earlier in §3.1, SERENA derives a starter matching from the arrival graph. This starter matching, which is typically partial, is then populated into a full matching by pairing the unmatched vertices in the bipartite graph in a round-robin manner. SERENA

then combines, using a MERGE procedure, this full matching with the matching used in the previous time slot, to arrive at a new matching that is at least as heavy as both matchings. This new matching will then be used for the current time slot. We omit the details of this MERGE procedure, since it is not related to how QPS augments SERENA. Finally, to precisely specify QPS-SERENA, it suffices to note that the only difference between QPS-SERENA and SERENA is that QPS-SERENA uses a QPS-generated starter matching, instead of one derived from the arrival graph.

4.1.3 QPS vs. ShakeUp

As we have shown, QPS is used mainly as an "add-on" to certain crossbar scheduling algorithms. In the literature, the only other add-on technique that we are aware of is ShakeUp [67]. ShakeUp is a set of randomized algorithms designed to boost the performance of certain *iterative* crossbar scheduling algorithms, such as iSLIP and iLQF. It does so by preventing these iterative algorithms from getting stuck at (locally) maximal matchings during their iterative executions. ShakeUp is typically used as follows: a ShakeUp-augmented crossbar scheduling algorithm alternates between an iteration of the underlying crossbar scheduling algorithm (*e.g.,* iSLIP) and a ShakeUp iteration.

There are two types of ShakeUp algorithms: unweighted and weighted. The unweighted ShakeUp is designed to augment crossbar scheduling algorithms that do not consider VOQ lengths in their decision-making, such as Parallel Iterative Matching (PIM) [16] and iSLIP [5]. In each unweighed ShakeUp iteration, unmatched input ports are first permuted in a random order. From this (random) order, each unmatched input port sends a request to an output port *uniformly at random* (*i.e.,* unweighted) chosen from the set of output ports to which the corresponding VOQs are nonempty. An output port, upon receiving such a request, must now pair with this input port, even if it was already paired with another input port. If an output port receives multiple requests during the same ShakeUp iteration, it selects one of them uniformly at random. The iSLIP scheme augmented this way was

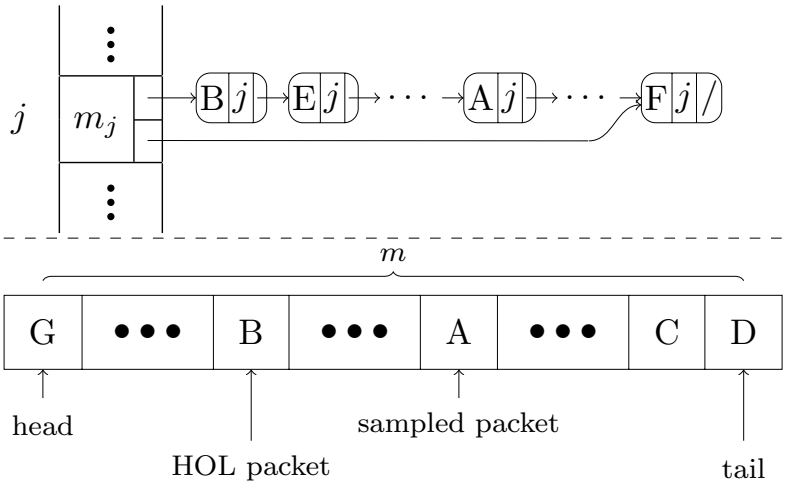
called SLIP-SHAKE in [67]. In §4.4, we will compare the its performance (renamed to iSLIP-ShakeUp) with that of QPS-iSLIP.

The weighted ShakeUp [67] is designed to augment crossbar scheduling algorithms that incorporate VOQ lengths in their decision-making, such as iLQF [36]. In each weighed ShakeUp iteration, each unmatched input port, one after another in the above-mentioned randomly order, sends a request to an output port with a probability proportional to the length of the corresponding VOQ.

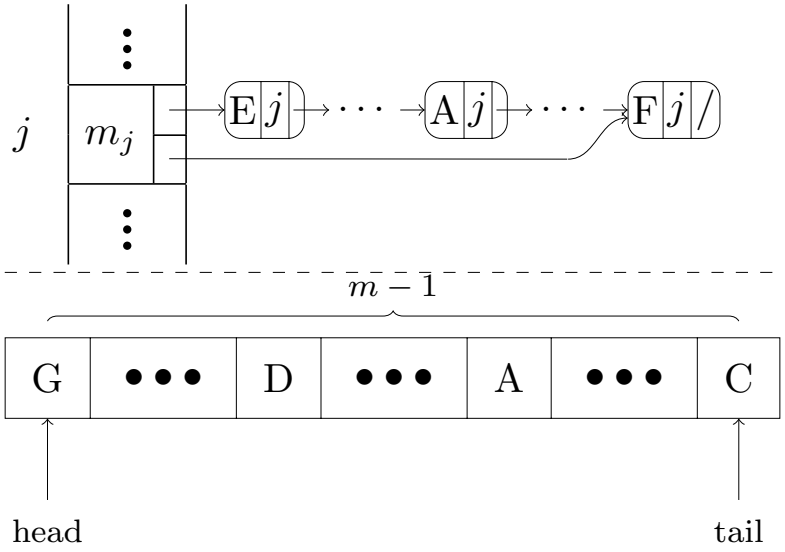
Admittedly, weighted ShakeUp's proposing strategy sounds very similar to our QPS strategy. However, there are four key differences: how they are used, how widely applicable they are, their intended purpose, and how they are implemented. First, in ShakeUp, only unmatched input ports execute this strategy to "shake up" an existing suboptimal matching, whereas in QPS, all input ports execute the strategy at the very beginning to generate a starter matching for other crossbar scheduling algorithms to build on. In a sense, ShakeUp is designed for "post-processing" whereas QPS is designed for "pre-processing". Second, while our QPS scheme can easily augment a non-iterative algorithm such as SERENA, it is not known whether ShakeUp, weighted or unweighted, can do the same. Third, it was never suggested in [67] that this (weighted) strategy might be suitable for "weight-oblivious" crossbar scheduling algorithms such as PIM or iSLIP; only the unweighed ShakeUp was "prescribed" for PIM or iSLIP. Last, unlike in our work, there was no mention of how the queue-proportional proposing strategy could be carried out in $O(1)$ time (per port), and no data structure was proposed for doing so [67].

4.2 QPS Implementation

In this section, we describe the data structure and algorithm that allows an input port to sample a VOQ in the queue-proportional manner (*i.e.,* Line 2 of Algorithm 3), and, if needed, to remove the Head-of-Line (HOL) packet of any VOQ (for receiving switching service), both with $O(1)$ (per port) time complexity. This data structure is extremely simple,



(a) Before scheduling



(b) After scheduling

Figure 4.1: Illustrating the action of the QPS data structures on a single input port.

although we have so far not been able to find anything sufficiently similar in the literature.

The memory overhead of the QPS data structure is no more than $20\,\text{bytes}$ per packet; the detailed "accounting" is shown in §B.2. Assuming an average packet size of $500\,\text{bytes}$, the amount of memory consumed by the QPS data structure is no more than 4% of what is needed for storing the actual packets. This is a modest space overhead ratio to pay, for the significant improvements in switching performance.

4.2.1 Overview of The Sampling Algorithm

We first provide a high-level overview of the sampling algorithm. It consists of two steps. In the first step, we sample a packet, out of all packets currently queued at the input port, uniformly at random. Specifically, if there are a total of m packets across all N VOQs at the input port, each packet is sampled with probability $1/m$. With such uniform sampling, the j^{th} VOQ, which has length m_j, will have one of packets sampled with probability m_j/m. This is precisely the QPS behavior called for in Line 2 of Algorithm 3.

Suppose a packet is thus sampled. A part of the second step is to find out which VOQ this packet belongs to so that the input port can propose to the corresponding output port with its queue length (see Line 3 of Algorithm 3). However, more effort is still required. Since all crossbar scheduling algorithms serve packets in a VOQ strictly in the FIFO order, if this proposal is successful (*i.e.,* accepted by the output port), and the input and output port pair is eventually a part of the final matching, the HOL packet of this VOQ, which may or may not be the sampled packet, needs to be located and serviced. Hence, the other part of the second step is to locate the HOL packet of this VOQ.

Before going into the details, we list two other basic operations that this data structure needs to also support. The first operation is that any new incoming packet must be recorded in the data structure so that it is logically "added to the end of the VOQ that it belongs to". The second operation is that, when the scheduling algorithm eventually decides to pair the input port with a different output port than was proposed to, which could happen due to either the proposal being rejected or the initially accepted proposal being overridden by the scheduling algorithm (*e.g.,* during SERENA's MERGE operation in the case of QPS-SERENA), the HOL packet of the (new) corresponding VOQ needs to be located and removed for receiving the switching service. Both operations can be supported with $O(1)$ complexity, as will be shown next.

4.2.2 The Detailed Data Structure

We show that the two steps of the QPS proposing strategy can be performed in $O(1)$ time, at any input port, via a main and an auxiliary data structures, that are the same for all input ports. Figure 4.1a and Figure 4.1b present the data structures, *at a single input port*, before and after the HOL packet of its j^{th} VOQ is chosen for (switching) service. The top half and bottom half of the figures show the main and the auxiliary data structures respectively.

Main Data Structure. The main data structure is an array of N records, corresponding to the N VOQs at the input port. Each record j (*i.e.,* array entry j) is associated with a linked list, which corresponds to (pointers to) packets queued at a VOQ in the order they arrived, starting with the HOL packet. Each node in the linked list contains two pointers encoded as "$\langle letter \rangle$" (*e.g.,* A); one points to the actual packet (*e.g.,* packet A) in the packet buffer (not shown in the figure) and the other to the corresponding entry (*e.g.,* entry A) in the auxiliary data structure, which we refer to as a *back pointer*.

For simplicity, Figure 4.1 shows only record j (corresponding to VOQ j). Each record contains a head and a tail pointers that point to the head node and the tail node of the linked list respectively. The head pointer is needed for locating and for removing the head node (*i.e.,* the HOL packet) in $O(1)$ time; it is also needed for locating and replacing the array entry that corresponds to the HOL packet in the auxiliary data structure. The tail pointer is needed for inserting a newly arrived packet to the "end of the VOQ" (*i.e.,* the first basic operation) in $O(1)$ time.

Auxiliary Data Structure. The bottom half of Figure 4.1 shows the auxiliary data structure used for performing the sampling. Suppose there are a total of m packets queued across all N VOQs at the input port. The auxiliary data structure is simply an array of m entries, each of which is a pointer that points to a distinct (packet) node (*e.g.,* node A) in one of the N linked lists in the main data structure.

Despite arrivals and departures of packets over time, the auxiliary data structure always

occupies a contiguous block of array entries, the boundaries of which are identified by a head and a tail pointer as shown in the bottom half of Figure 4.1. This contiguity allows any array entry (packet) to be sampled uniformly at random in $O(1)$ time, an aforementioned key step of QPS. Hence this contiguity needs to be maintained in the event of packet arrivals and departures. The case of a packet arrival is easier: the entry corresponds to the new packet is inserted after the current tail position, and the tail pointer updated. The case of a packet departure is only slightly trickier: if the departing packet leaves a "hole" in the block, the tail entry is moved to fill this hole, and the tail pointer updated.

In the case of a packet departure, the (packet) node in the main data structure that is pointed to by the former tail entry (now moved to "fill the hole") needs to have its *back pointer* updated to the offset of the former hole, where the former tail entry now is. This is clearly an $O(1)$ procedure. A similar procedure can be used to support the second basic operation in $O(1)$ time.

An Illustrative Example. To see how the main and the auxiliary data structures work together to facilitate QPS, consider the example shown in Figure 4.1. In Figure 4.1a, the packet A was sampled out of m packets in the auxiliary data structure. However, it is not the HOL packet, so its destination (output) port (*i.e.,* VOQ identifier) is checked, which turns out to be j. By accessing the j^{th} record in the main data structure, which corresponds to VOQ j, the HOL packet is packet B. Now, the input port proposes to match with output port j. In Figure 4.1b, if the proposal is accepted by, and the input port is eventually matched to, output port j, packet B will depart (for output port j) in the current time slot. The head pointer in the j^{th} record of the main data structure is updated to (point to) E, the new HOL packet. These operations, *i.e.,* the search for the HOL packet, and the updates to both data structures, all take $O(1)$ time.

4.3 Stability Proof of QPS-SERENA

In this section, we prove that the QPS-SERENA algorithm is stable (*i.e.*, can achieve 100% throughput) under any arrival processes that are admissible and satisfy certain mild conditions. In §4.3.1, we introduce some background information and notations that we need in the stability proofs. In §4.3.2, we describe a theorem used in [11] to prove the stability of the TASS algorithm. Unfortunately, this theorem is not applicable to QPS-SERENA, because QPS-SERENA in general does not satisfy the so-called *Property P*, a condition required by the theorem. In §4.3.3, we state a stronger theorem that requires only a weaker condition than *Property P*, which is satisfied by QPS-SERENA.

4.3.1 Background and Notations

We first define three $N \times N$ matrices $Q(t)$, $A(t)$, and $S(t)$. Let $Q(t) = \big(q_{ij}(t)\big)$ be the queue length matrix where $q_{ij}(t)$ is the length of the j^{th} VOQ at input port i during time slot t. Let $A(t) = \big(a_{ij}(t)\big)$ be the traffic arrival matrix where $a_{ij}(t)$ is the number of packets arriving at the input port i destined for output port j during time slot t, which can be viewed as the counting process associated with underlying traffic arrival process. Let $S(t) = \big(s_{ij}(t)\big)$ be the schedule (matching) matrix for time slot t output by the crossbar scheduling algorithm. As we explained earlier, each $S(t)$ is a 0-1 matrix in which $s_{ij}(t) = 1$ if and only if input port i is matched with output j during time slot t. Then, the queue length matrix Q evolves over time as follows. For $\forall 1 \leq i, j \leq N$,

$$q_{ij}(t+1) = [q_{ij}(t) + a_{ij}(t) - s_{ij}(t)]^+ \tag{4.1}$$

where $[\,\cdot\,]^+$ is defined as $\max\{\,\cdot\,, 0\}$. With a slight abuse of the notation, we rewrite (4.1), into the matrix form, as $Q(t+1) = [Q(t) + A(t) - S(t)]^+$.

Like in [68], we assume that, for each $1 \leq i, j \leq N$, $\{a_{ij}(t)\}_{t=0}^{\infty}$ is a sequence of *i.i.d.* random variables, and the second moment of their common distribution ($= \mathbb{E}\big[a_{ij}^2(0)\big]$) is finite. Note that, the same or even stronger assumptions (*e.g., i.i.d.* Bernoulli arrivals) were

made for proving the stabilities of TASS [11] and SERENA [4, 10] respectively. For ease of presentation, we refer to such an $A(t)$ as an *i.i.d.* arrival (counting) process in the sequel of this chapter.

Now we flatten the $N \times N$ matrices Q, A, and S into N^2-dimensional vectors in the row-major order, *i.e.,* the first row of the matrix becomes the first N scalars in the vector, the second row becomes the next N scalars, and so on. Now that Q, A, and S are vectors, we can take their inner products, denoted as $\langle \cdot, \cdot \rangle$, in the following derivations. For example, $\langle S(t), Q(t) \rangle$ is the weight of the schedule (matching) $S(t)$, *w.r.t.* the queue length vector $Q(t)$, at time slot t.

4.3.2 TASS, SERENA, and Their Stability

4.3.2.1 The Adaptive and Non-Degenerative Family

The idea of TASS [11], shown below, is very simple: generate a "fresh" (*i.e.,* independent of all other random vectors) random matching $R(t)$, compare its weight with that of $S(t-1)$, the matching used in the previous time slot, and use the winner as the matching for the current time slot (*i.e.,* $S(t)$). Here $R(t)$ is a random vector whose distribution is parameterized only by the current VOQ length vector $Q(t)$. Amazingly, such a simple adaptive algorithm can achieve 100% throughput, albeit at the cost of higher delays.

$$S(t) = \begin{cases} R(t) & \text{if } \langle R(t), Q(t) \rangle \geq \langle S(t-1), Q(t) \rangle \\ S(t-1) & \text{otherwise} \end{cases} \tag{4.2}$$

Note that the TASS algorithm is also by definition (*i.e.,* (4.2)) non-degenerative, defined next.

Definition 4.3.1. A scheduling algorithm is *non-degenerative* if it guarantees that for any time slot $t \geq 1$, we have

$$\langle S(t), Q(t) \rangle \geq \langle S(t-1), Q(t) \rangle$$

4.3.2.2 *Generalized Algorithm Family* $\widetilde{\Pi}$

Denote Π as the family of adaptive algorithms defined by (4.2). For the TASS' stability proof and theorem to apply also to SERENA, we need to generalize the family of Π to $\widetilde{\Pi}$ that is defined by

$$S(t) = \mathcal{F}\big(R(t), S(t-1), Q(t)\big) \tag{4.3}$$

where $\mathcal{F}$ is an operator, the resulting $S(t)$ satisfies the *non-degenerative* property defined above, and $R(t)$ is a random schedule whose probability distribution is a function only of $Q(t)$. To ease proving our result, we also force $S(t) = R(t)$ *when all queues (VOQs) are empty* at time slot t, *i.e.,* to "forget the previous schedule $S(t-1)$" and reset to the "default random schedule" $R(t)$.

In TASS, this $\mathcal{F}$ is clearly the "MAX operator", that is, choosing the heavier schedule *w.r.t.* $Q(t)$, between $R(t)$ and $S(t-1)$. In SERENA, this $\mathcal{F}$ is the MERGE operator, that is, $S(t) = MERGE\big(R(t), S(t-1), Q(t)\big)$. As we explained in §4.1.2.2, the MERGE operator combines two matchings into one that is at least as heavy, *w.r.t.* $Q(t)$, as either, so the SERENA algorithm, like TASS, is also *non-degenerative*. Hence, SERENA also belongs to this extended family $\widetilde{\Pi}$. *Now it is clear that QPS-SERENA also belongs to* $\widetilde{\Pi}$ because it differs from SERENA only in how the random schedule $R(t)$ is computed, and in QPS-SERENA this $R(t)$ is generated in the "$Q(t)$-proportional" manner (so its probability distribution is a function only of $Q(t)$).

We claim that, given any crossbar scheduling algorithm $\pi \in \widetilde{\Pi}$, the joint queueing and scheduling process $\big\{\big(Q(t), S(t)\big)\big\}_{t=0}^{\infty}$, resulting from π and any *i.i.d.* arrival process $A(t)$ (not necessarily admissible), is a Markov chain. This property is clear from the following two facts. First, by (4.3), $S(t)$ is a function of only $Q(t)$ and $S(t-1)$ (note $R(t)$ is a function only of $Q(t)$). Second, by (4.1), $Q(t)$ is a function of only $Q(t-1)$, $S(t-1)$, and the random packet arrival vector $A(t)$ that is independent of all other random vectors.

4.3.2.3 *Stability Theorem for Family* $\widetilde{\Pi}$

The following theorem, concerning the stability of the family of crossbar scheduling algorithms $\widetilde{\Pi}$, was proven in [11].

Theorem 4.3.1. For any (randomized) algorithm $\pi \in \widetilde{\Pi}$ that satisfies Property P, defined next, and under any admissible *i.i.d.* arrival process $A(t)$ (defined in §4.3.1), the joint queueing and scheduling process $\big\{\big(Q(t), S(t)\big)\big\}_{t=0}^{\infty}$ is an ergodic Markov chain, and as a consequence, the queueing process $\{Q(t)\}_{t=0}^{\infty}$ converges in distribution to a random vector $\widehat{Q}$. Furthermore,

$$\mathbb{E}[\|\widehat{Q}\|_1] < \infty$$

where $\| \cdot \|_1$ is the 1-norm.

Fix a randomized crossbar scheduling algorithm π. Let $W(t) \triangleq \langle S(t), Q(t)\rangle$ be the weight of the schedule output by π at time slot t. Denote as W_Q the weight of the MWM *w.r.t.* a queue length vector Q, *i.e.,* $W_Q \triangleq \max_S\{\langle S, Q\rangle\}$. Let S_Q be one of the schedules that attain this maximum weight (*i.e.*, $\langle S_Q, Q\rangle = W_Q$).

Definition 4.3.2 (Property P [11])**.** A crossbar scheduling algorithm π satisfies Property P if at any time slot t,

$$\mathbb{P}\big[W(t) = W_{Q(t)}\big] \geq \delta$$

where $\delta > 0$ is a constant independent of the time slot t and the queue length vector $Q(t)$.

In other words, π satisfies *Property P* if, at any time slot t, the schedule $S(t)$ output by π is a MWM with at least a constant probability δ. Both TASS and SERENA satisfy *Property P* because there is a constant (*w.r.t.* $Q(t)$) probability for $R(t)$ to be a MWM in both cases, and when this happens, $S(t)$ remains a MWM after a "MAX" or "MERGE" operation. Since both TASS and SERENA also belong to family $\pi \in \widetilde{\Pi}$, Theorem 4.3.1 implies that both can achieve 100% throughput.

4.3.3 Stability of QPS-SERENA

Although QPS-SERENA also belongs to family $\widetilde{\Pi}$, Theorem 4.3.1 is not applicable to QPS-SERENA, because it can be shown that QPS-SERENA does not satisfy *Property P*. We establish a stronger theorem that allows us to prove that QPS-SERENA can achieve 100% throughput. More specifically, we first show in Lemma 4.3.1 that QPS-SERENA satisfies a weaker condition called (ϵ, δ)-*MWM*, defined next[1]. Then we show in Theorem 4.3.2 that this weaker condition, combined with the $\widetilde{\Pi}$ family membership, is sufficient for a crossbar scheduling algorithm to achieve 100% throughput.

Definition 4.3.3. A crossbar scheduling algorithm π is called (ϵ, δ)-*MWM*, if $\forall \epsilon > 0$, there exists a constant $0 < \delta \leq 1$ s.t.

$$\mathbb{P}\big[W(t) \geq (1-\epsilon)W_{Q(t)}\big] \geq \delta$$

where δ is a constant independent of the time slot t and the queue length vector $Q(t)$. Note this δ can depend on ϵ and other (constant) system parameters such as N. Here, $W(t)$ and $W_{Q(t)}$ are similarly defined as before.

In other words, an algorithm π is called (ϵ, δ)-*MWM* if, at any time slot t, the schedule $S(t)$ output by π is within $(1-\epsilon)$ of the optimal (*i.e.,* MWM) with at least a constant probability δ. This condition is clearly weaker than *Property P*, which requires $S(t)$ to be optimal (*i.e.,* MWM) with at least a constant probability.

The following lemma (Lemma 4.3.1) shows that QPS alone is (ϵ, δ)-*MWM*. Since at any time slot t, QPS-SERENA merges $S(t-1)$ with the schedule $R(t)$ output by QPS, resulting in a schedule $S(t)$ that is at least as heavy as $R(t)$, QPS-SERENA is also (ϵ, δ)-*MWM*. Therefore, by Theorem 4.3.2 below, we conclude that QPS-SERENA can achieve 100% throughput.

Lemma 4.3.1. QPS is (ϵ, δ)-MWM.

[1]Note that, the definition of (ϵ, δ)-*MWM* is quite different than that of the **1-APRX** (to MWM) defined in [69].

Proof: See §B.4. ■

Theorem 4.3.2. For every algorithm $\pi \in \widetilde{\Pi}$ that is (ϵ, δ)-*MWM*, the conclusion of Theorem 4.3.1 (*i.e.*, convergence to a stationary distribution with finite first moment) continues to hold, under admissible *i.i.d.* arrivals.

Proof: See §B.3. ■

Remarks. Like Theorem 4.3.2 above, Theorem 1 in [54] also establishes stability with conditions weaker than that are needed in Theorem 4.3.1. However, they weaken different parts of the assumptions made in Theorem 4.3.1, and hence their proofs are very different. Theorem 4.3.2 above weakens *Property P* in Theorem 4.3.1 above to (ϵ, δ)-MWM. In contrast, Theorem 1 in [54] requires *Property P*, but weakens the non-degenerative requirement (see Definition 4.3.1) in Theorem 4.3.1 above, by allowing it to be violated with a tiny probability.

4.4 Performance Evaluation

In this section, we compare the performance of two QPS-augmented algorithms, QPS-iSLIP and QPS-SERENA, against the iterative Longest Queue First (iLQF) [36], iSLIP-ShakeUp (iSLIP augmented by ShakeUp [67]), and the two original algorithms, iSLIP [5] and SERENA [4]. We evaluate, through simulations, their throughputs and delays under various load conditions and traffic patterns. Maximum Weighted Matching (MWM) is also simulated to provide a benchmark for these comparisons.

The evaluation results show conclusively that QPS-iSLIP and QPS-SERENA outperform iSLIP and SERENA respectively in both throughput and delay. They also show that QPS-iSLIP brings about the same amount of performance improvement to iSLIP as iLQF, even though QPS-iSLIP is far less computationally expensive ($O(\log^2 N)$ per port) than iLQF ($O(N)$ per port), thus giving the "same bang for less buck". Furthermore, they show QPS-iSLIP overall performs better than iSLIP-ShakeUp.

4.4.1 Simulation Setup

In all our simulations, we set the number of input/output ports $N = 32$. Note that we have also investigated how the mean delay performance of various crossbar scheduling algorithms scales with respect to N; these results are shown in §B.5.2. For the accurate measurement of throughput and delay, each VOQ is assumed to have infinite buffer, so that there is no packet drop at any input port. Every simulation run lasts $6{,}000 \times N^2$ ($= 6.144 \times 10^6$) time slots. This duration is chosen so that every simulation run enters the steady state after a tiny fraction of this duration and stays there for the rest. The throughput and delay measurements are taken after the simulation run enters the steady state.

We initially assume *i.i.d.* Bernoulli traffic arrivals: the distributions of arrivals to different input ports are *i.i.d.*, and in each time slot, there is a probability $\rho \in (0, 1)$ that a packet will arrive. We will then look at bursty traffic arrivals further below. As mentioned earlier in §1.2.6, the four standard types of traffic patterns are used for generating the switch's workloads.

In both iSLIP and iLQF, the total number of iterations in a time slot is usually set to $\log_2 N$. However, to achieve a fair comparison between iSLIP, iLQF, and QPS-iSLIP, in simulating these algorithms, the total number of iterations in a time slot is set to $1 + \log_2 N$. For instance, with QPS-iSLIP, this means that we ran 1 iteration of QPS followed by $\log_2 N$ iterations of iSLIP. In doing so, we emphasize that the outperformance of QPS-iSLIP does not come from an extra iteration. Note that, with $1 + \log_2 N$ iterations, the complexity of both iSLIP and QPS-iSLIP remains $O(\log^2 N)$ per port and that of iLQF remains $O(N)$ per port.

For iSLIP-ShakeUp, we alternate between an iSLIP iteration and a ShakeUp iteration *also* for a total of $\log_2 N + 1$ iterations (*i.e.,* $\frac{\log_2 N+1}{2}$ iterations for each). This algorithmic setting and parameter setting both follow the guidelines provided in [67] for iSLIP-ShakeUp, and the throughput numbers we have obtained (shown in Table 4.1) match those reported in [67].

Table 4.1: Maximum achievable throughput.

Traffic	Uniform	Quasi-diagonal	Log-diagonal	Diagonal
iSLIP	100.00%	81.70%	83.85%	83.47%
QPS-iSLIP	100.00%	99.38%	96.46%	88.36%
iSLIP-ShakeUp	99.98%	91.08%	92.73%	92.41%
iLQF	100.00%	99.41%	96.47%	89.32%

We consider two performance metrics: throughput and delay. We measure two types of delays: the mean delay and the 95^{th} percentile delay. The 95^{th} percentile delay is the delay value exceeded by exactly 5% of the packets. This 95^{th} percentile delay gauges whether a crossbar scheduling algorithm sacrifices the delay performance of packets in the longest VOQs when evacuating other VOQs. In our simulations, the 95^{th} percentile delay is measured by using the high dynamic range (HDR) histograms [70].

4.4.2 Throughput Performance

We have measured the maximum achievable throughputs of iSLIP, QPS-iSLIP, iSLIP-ShakeUp and iLQF, under the 4 different traffic patterns and an offered load close to 100%. The results are presented in Table 4.1. We do not include the throughputs of MWM, SERENA and QPS-SERENA in Table 4.1 because they provably achieve 100% throughput.

There are three important observations from Table 4.1. First, for all traffic patterns except the uniform, or non-uniform traffic patterns, where iSLIP does poorly, QPS-iSLIP significantly boosts the throughput performance of iSLIP, increasing it by an additive term of 0.1768, 0.1261, and 0.0489 for the quasi-diagonal, log-diagonal, and diagonal traffic patterns respectively. Moreover, for non-uniform traffic, the throughput of QPS-iSLIP are very close to those of iLQF, which is much more expensive computationally. Second, the throughput of QPS-iSLIP is higher than that of iSLIP-ShakeUp under all traffic patterns except the diagonal. Third, just like iSLIP, QPS-iSLIP can achieve 100% throughput under the uniform traffic.

We highlight a subtle fact that may sound counterintuitive to some readers: That a

switch (running a scheduling algorithm) has a (maximum achievable) throughput of $\mu < 1$ when the offered load is 100% does not imply that the switch is stable under any offered load (say ρ) smaller than μ. This is because the extra $1 - \rho$ "switching resource" freed up by the reduced offered load may not all be efficiently utilized by the scheduling algorithm to clear up the longest queues. For example, iSLIP-ShakeUp is not stable under the quasi-diagonal traffic when the offered load is 90% (see the corresponding missing point in Figure 4.2 (1^{st} row, 2^{nd} from left)), even though its throughput under 100% offered load, *i.e.,* the maximum achievable throughput, is 91.08%. In the sequel, we use the terms "load", "normalized load", "offered load", "traffic load" and "load factor" interchangeably.

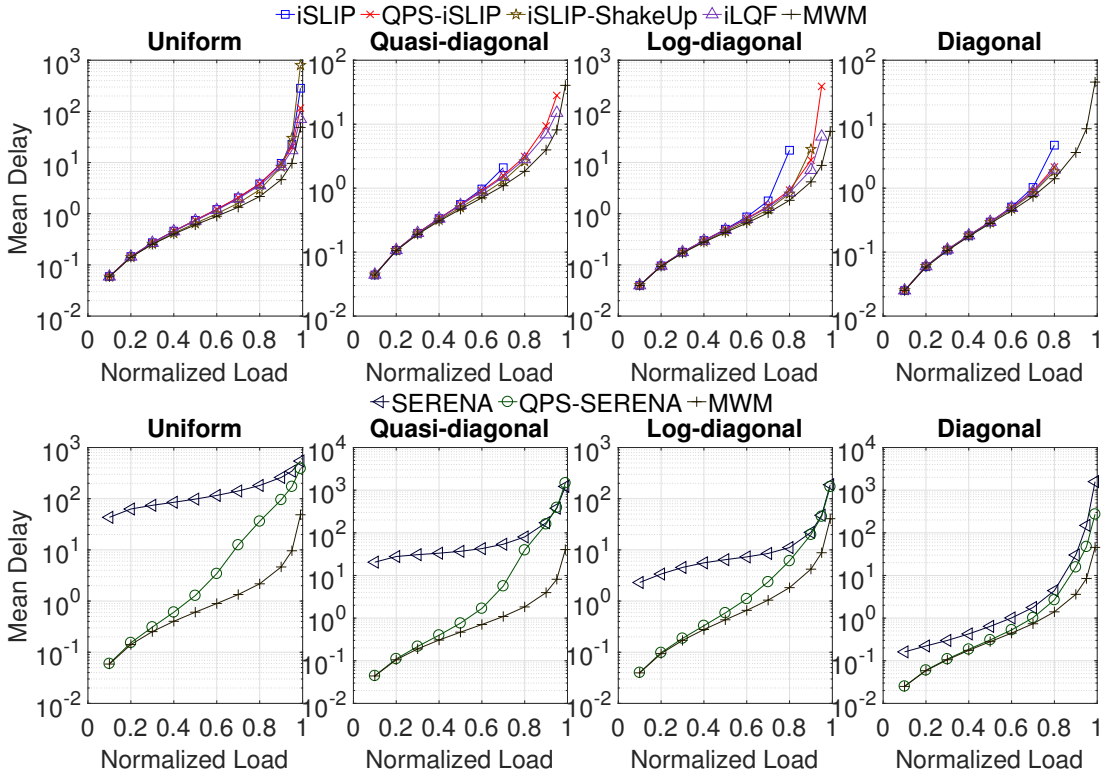


Figure 4.2: Mean delays under *i.i.d.* Bernoulli traffic arrivals with the 4 traffic patterns.

4.4.3 Delay Performance

4.4.3.1 Bernoulli Arrivals

Figure 4.2 (the 1^{st} row) presents the mean delays of iSLIP, QPS-iSLIP, iSLIP-ShakeUp, iLQF, and MWM under the 4 different traffic patterns. Since iSLIP, QPS-iSLIP, iSLIP-ShakeUp, and iLQF generally cannot achieve 100% throughput, we only measure their delay performance under the offered loads that make them stable; in all figures in the sequel, each "missing point" on a curve indicates that the corresponding scheduling algorithm is not stable under the corresponding offered load.

Figure 4.2 (the 1^{st} row) clearly shows that QPS-iSLIP has lower mean delays than iSLIP under all traffic patterns, especially when the load factor is high (*e.g.,* 80%); we note that the differences between the curves unfortunately look smaller on a log scale (on the y-axis) than they actually are. In addition, the mean delays of QPS-iSLIP are very close to those of iLQF, the more expensive algorithm computationally, under all traffic patterns and load factors.

Figure 4.2 (the 1^{st} row) also shows that QPS-iSLIP has either similar or slightly higher mean delays than iSLIP-ShakeUp under all traffic patterns, when the traffic load is low to moderate. However, when the traffic load is high (say $> 80\%$), the iSLIP-ShakeUp either becomes unstable or has higher mean delays than QPS-iSLIP, under all traffic patterns.

Figure 4.2 (the 2^{nd} row) presents the mean delays of SERENA, QPS-SERENA, and MWM under the 4 different traffic patterns. We can see that QPS-SERENA outperforms SERENA under all traffic patterns for all load factors. More specifically, QPS-SERENA outperforms SERENA by a wide margin, under uniform and diagonal traffic patterns for all load factors; it does so also under quasi-diagonal and log-diagonal traffic patterns for load factors that are not too high (≤ 0.8).

Figure 4.2 (the 2^{nd} row) also shows that the relative difference of the mean delay between QPS-SERENA and SERENA generally becomes larger as the traffic load becomes

lighter. This phenomena is due to the choice of the starter matching. In SERENA, the starter matching is the arrival graph, and when the load is light, the arrival graph does not provide enough "cue" for the scheduling algorithm to select the longest VOQs. QPS-SERENA, on the other hand, has a better starter matching that accounts for the VOQ lengths under any load conditions, and thus beats SERENA in mean delay. The outperformance of QPS-SERENA over SERENA reinforces our message about the importance of choosing a good starter matching.

Figure 4.3 (the 1^{st} row) shows the 95^{th} percentile delays of iSLIP, QPS-iSLIP, iSLIP-ShakeUp, iLQF, and MWM under the 4 different traffic patterns. Due to the presence of delay values that are very close to 0, which would severely "deform" all the curves if they were plotted in a log scale on the y-axis, Figure 4.3 is plotted in the linear scale on the y-axis. Figure 4.3 (the 1^{st} row) shows that QPS-iSLIP and iLQF achieve much lower 95^{th} percentile delays than iSLIP and iSLIP-ShakeUp, especially under heavy loads.

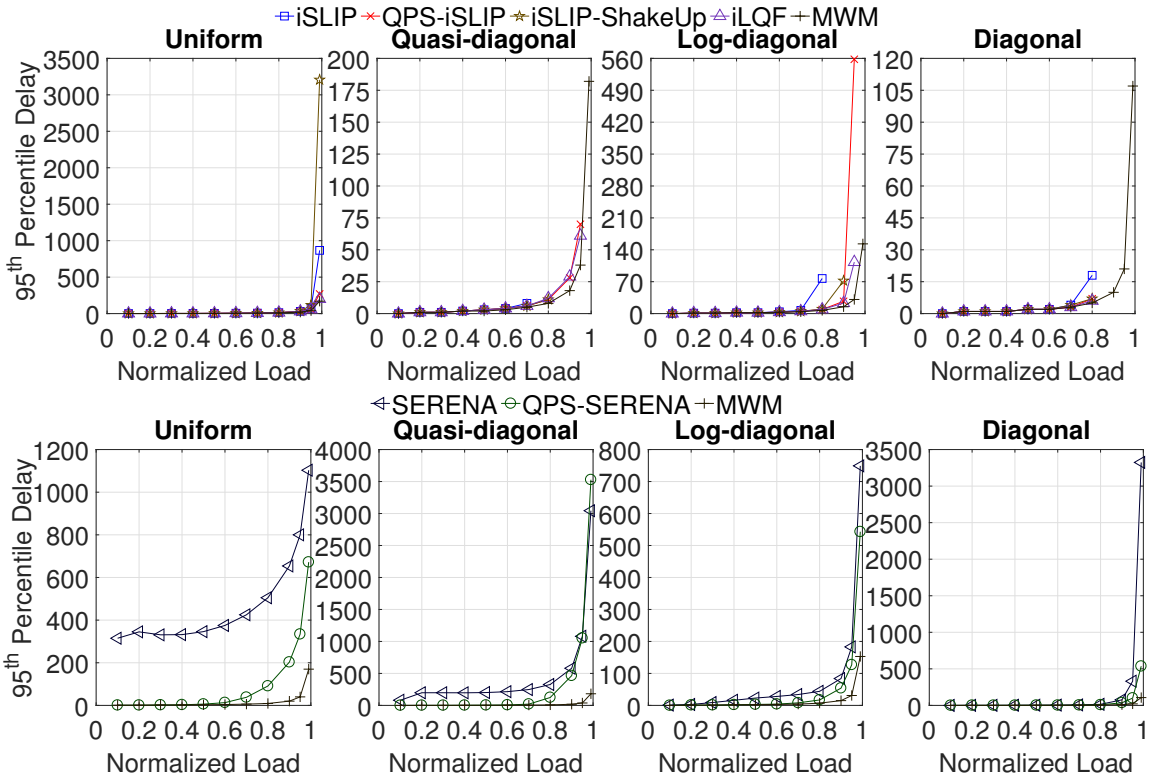


Figure 4.3: 95^{th} percentile delay under *i.i.d.* Bernoulli traffic arrivals with the 4 traffic patterns.

Figure 4.3 (the 2^{nd} row) shows the 95^{th} percentile delays of QPS-SERENA, SERENA, and MWM under the 4 different traffic patterns. Again QPS-SERENA outperforms SERENA by a wide margin under all 4 traffic patterns for almost all load factors.

4.4.3.2 Bursty Arrivals

In real networks, packet arrivals are likely to be bursty. In this section, we evaluate the performance of these scheduling algorithms under bursty traffic, generated by a two-state ON-OFF arrival process described in [4]. The durations of each ON (burst) stage and OFF (no burst) stage are geometrically distributed: the probabilities the ON and OFF state lasts for $t \geq 0$ time slots are given by

$$P_{ON}(t) = p(1-p)^t \text{ and } P_{OFF}(t) = q(1-q)^t,$$

with the parameters $p, q \in (0, 1)$ respectively. As such, the average duration of the ON and OFF states are $(1-p)/p$ and $(1-q)/q$ time slots respectively.

In an OFF state, an incoming packet's destination (*i.e.,* output port) is generated according to the corresponding traffic pattern. In an ON state, all incoming packet arrivals to an input port would be destined to the same output port, thus simulating a burst of packet arrivals. By adjusting p, we can control the desired average burst size while by adjusting q, we can control the load of the traffic.

We first compare QPS-iSLIP against iSLIP, iSLIP-ShakeUp, iLQF, and MWM, with average burst sizes ranging from 8 to 1024 packets, on an offered load of 0.75. We use this load factor because iSLIP is not stable under certain load matrices when the offered load is larger than or equal to 0.8.

The simulation results are shown in Figure 4.4 (the 1^{st} row). We can see that QPS-iSLIP beats iSLIP, and is on par with iLQF and QPS-ShakeUp, under all traffic patterns for all burst sizes. Furthermore, QPS-iSLIP beats iSLIP by a wide margin, under quasi-diagonal and log-diagonal traffic patterns. In fact, the starter matching generated by QPS for iSLIP is so superior that QPS-iSLIP is only slightly worse than MWM in the mean

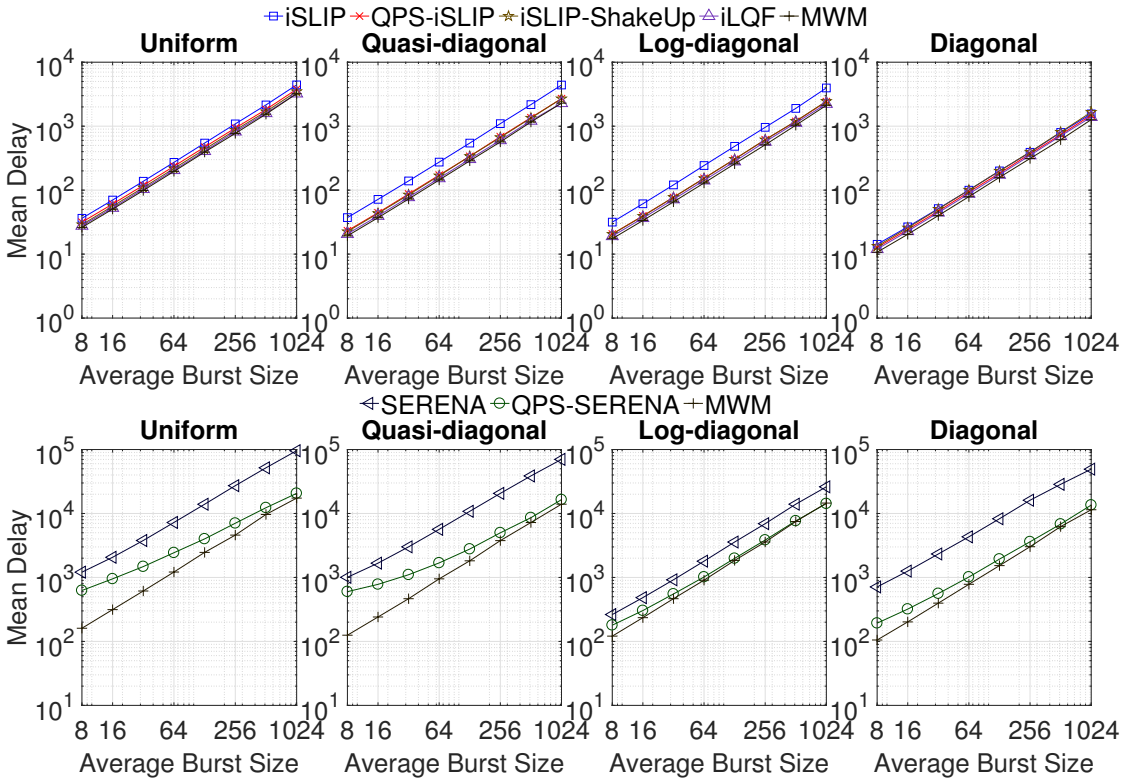


Figure 4.4: Mean delays under bursty traffic arrivals with the 4 traffic patterns.

delay performance under all traffic patterns for all burst sizes.

We then evaluate QPS-SERENA's mean delay performance against SERENA's and MWM's. Figure 4.4 (the 2^{nd} row) presents the results with average burst sizes ranging from 8 to 1024 packets under an offered load of 0.95, under the 4 traffic patterns respectively. Performance under other heavy loads, such as at 0.9, is similar to this case.

We can see from Figure 4.4 (the 2^{nd} row) that the mean delay increases for all scheduling algorithms when the burst size increases, under all 4 traffic patterns, which is not surprising. However, Figure 4.4 (the 2^{nd} row) also clearly shows that QPS-SERENA handles highly bursty traffic much better than SERENA, as we will elaborate next.

We make the following two observations from Figure 4.4 (the 2^{nd} row). First, QPS-SERENA outperforms SERENA by an increasingly wider margin, in both absolute and relative terms, as the burst size becomes larger. Second, the gap between QPS-SERENA and MWM shrinks rapidly as the burst size becomes larger. Our explanation for the first

observation is that, because QPS-SERENA obtains information directly from the *current* lengths of the VOQs, rather than indirectly from the *current* arrivals, QPS-SERENA reacts to the rapid build-up of packets in a VOQ from a *past* traffic burst much more promptly than SERENA. For the second observation, the reason is as follows. When the burst size increases, the longest one or two VOQs at every input port account for an increasingly higher percentage of all packets queued at the input port, and hence have an increasingly higher chances of being sampled by QPS, so the resulting starter matching becomes increasingly closer to an MWM.

4.5 Conclusion

In this chapter, we propose a new proposing strategy, called queue-proportional sampling (QPS), that generates superior starter matchings than all other known strategies. We use QPS to augment two existing crossbar scheduling algorithms, namely SERENA and iSLIP. We show that the augmented algorithms, namely QPS-SERENA and QPS-iSLIP, outperform the original algorithms by a wide margin, under various load conditions and traffic patterns. These performance enhancements come at virtually no additional computational cost due to QPS being an $O(1)$ algorithm (per port). Finally, to prove that QPS-SERENA can achieve 100% throughput, we have proved a new and stronger stability theorem.

CHAPTER 5

QPS-r

5.1 The QPS-r Algorithm

The QPS-r algorithm simply runs r iterations of QPS (see Chapter 4) to arrive at a matching. so its time complexity per port is exactly r times those of QPS. Since r is a small constant, it is $O(1)$, same as that of QPS.

Recall that, QPS was used in Chapter 4 as an "add-on" to augment other crossbar scheduling algorithms as follows. It generates a starter matching, which is then populated (*i.e.*, adding more edges to it) and refined, by other crossbar scheduling algorithms such as iSLIP [5] and SERENA [4, 10], into a final matching. To generate such a starter matching, QPS needs to run only one iteration, which consists of two phases, namely, a proposing phase and an accepting phase. The details of the two phases were described in §4.1.1.

The QPS-r Scheme. The QPS-r scheme simply runs r QPS iterations. In each iteration, each input port that is not matched yet, first proposes to an output port according to the QPS proposing strategy (see §4.1); each output port that is not matched yet, accepts a proposal (if it has received any) according the "longest VOQ first" accepting strategy. Hence, if an input port has to propose multiple times (once in each iteration), due to all its proposals (except perhaps the last) being rejected, the identities of the output ports it "samples" (*i.e.,* proposes to) during these iterations are samples with replacement, which more precisely are *i.i.d.* random variables with a queue-proportional distribution.

At the first glance, sampling with replacement may appear to be an obviously suboptimal strategy for the following reason. There is a nonzero probability for an input port to propose to the same output port multiple times, but since the first (rejected) proposal implies this output port has already accepted "someone else" (a proposal from another input

port), all subsequent proposals to this output port will surely go to waste. For this reason, sampling without replacement (*i.e.*, avoiding all output ports proposed to before) may sound like an obviously better strategy. However, it is really not, since compared to sampling with replacement, it has a much higher time complexity of $O(\log N)$, but improves the throughput and delay performances only slightly according to our simulation studies.

5.2 Throughput and Delay Analysis

In this section, we show that QPS-1 (*i.e.*, running a single QPS iteration) delivers exactly the same provable throughput and delay guarantees as maximal matching algorithms. When $r > 1$, QPS-r clearly should have better throughput and delay performances than QPS-1, as more input and output ports can be matched up during subsequent iterations, although we are not able to derive stronger bounds.

5.2.1 Preliminaries

In this section, we introduce the notation and assumptions that will later be used in our derivations. We define three $N \times N$ matrices $Q(t)$, $A(t)$, and $D(t)$. Let $Q(t) \triangleq \big(q_{ij}(t)\big)$ be the queue length matrix where each $q_{ij}(t)$ is the length of the j^{th} VOQ at input port i during time slot t. With a slight abuse of notation, we refer to this VOQ as q_{ij} (without the t term).

We define $Q_{i*}(t)$ and $Q_{*j}(t)$ as the sum of the i^{th} row and the sum of the j^{th} column respectively of $Q(t)$, *i.e.*, $Q_{i*}(t) \triangleq \sum_j q_{ij}(t)$ and $Q_{*j}(t) \triangleq \sum_i q_{ij}(t)$. With a similar abuse of notation, we define Q_{i*} as the VOQ set $\{q_{i1}, q_{i2}, \cdots, q_{iN}\}$ (*i.e.*, those on the i^{th} row), and Q_{*j} as $\{q_{1j}, q_{2j}, \cdots, q_{Nj}\}$ (*i.e.*, those on the j^{th} column).

Now we introduce a concept that lies at the heart of our derivations: neighborhood. For each VOQ q_{ij}, we define its neighborhood as $Q_{i*} \bigcup Q_{*j}$, the set of VOQs on the i^{th} row or the j^{th} column. We denote this neighborhood as $Q_{ij}^{\dagger}$, since it has the shape of a cross. Figure 5.1 illustrates $Q_{ij}^{\dagger}$, where the row and column in the shadow are the VOQ sets Q_{i*}

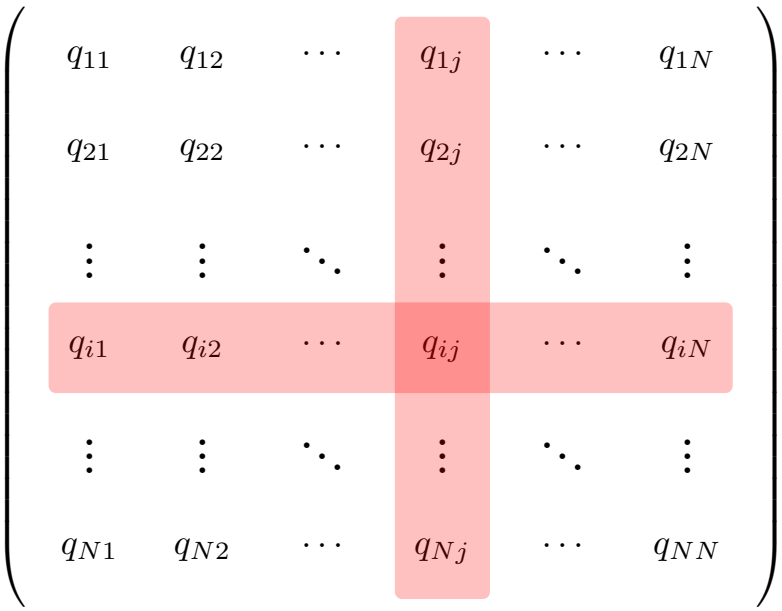


Figure 5.1: Illustration of neighborhood of q_{ij}, *i.e.,* $Q_{ij}^{\dagger}$.

and Q_{*j} respectively. $Q_{ij}^{\dagger}$ can be viewed as the *interference set* of VOQs for VOQ q_{ij} [19, 20], as no other VOQ in $Q_{ij}^{\dagger}$ can be active (*i.e.,* transmit packets) simultaneously with q_{ij}. We define $Q_{ij}^{\dagger}(t)$ as the total length of all VOQs in (the set) $Q_{ij}^{\dagger}$ at time slot t, that is

$$Q_{ij}^{\dagger}(t) \triangleq Q_{i*}(t) - q_{ij}(t) + Q_{*j}(t). \tag{5.1}$$

Here we need to subtract the term $q_{ij}(t)$ so that it is not double-counted (in both $Q_{i*}(t)$ and $Q_{*j}(t)$).

Let $A(t) = \big(a_{ij}(t)\big)$ be the traffic arrival matrix where $a_{ij}(t)$ is the number of packets arriving at the input port i destined for output port j during time slot t. For ease of exposition, we assume that, for each $1 \le i, j \le N$, $\{a_{ij}(t)\}_{t=0}^{\infty}$ is a sequence of *i.i.d.* random variables, the second moment of their common distribution ($= \mathbb{E}\big[a_{ij}^2(0)\big]$) is finite, and this sequence is independent of other sequences (for a different i and/or j). Our analysis, however, holds for more general arrival processes (*e.g.*, Markovian arrivals) that were considered in [19, 20], as we will elaborate shortly. Let $D(t) = \big(d_{ij}(t)\big)$ be the departure matrix for time slot t output by the crossbar scheduling algorithm. Similar to S (the matching matrix described in§1.2.2), $D(t)$ is a 0-1 matrix in which $d_{ij}(t) = 1$ if and only if a packet departs from q_{ij}

during time slot t. For any i, j, the queue length process $q_{ij}(t)$ evolves as follows:

$$q_{ij}(t+1) = q_{ij}(t) - d_{ij}(t) + a_{ij}(t). \tag{5.2}$$

Recall from §1.2.5 that $\Lambda = \big(\lambda_{ij}\big)$ is the (normalized) traffic rate matrix (associated with $A(t)$) where λ_{ij} is normalized (to the percentage of the line rate of an input/output link) mean arrival rate of packets to VOQ q_{ij}. With $a_{ij}(t)$ being an *i.i.d.* process, we have $\lambda_{ij} = \mathbb{E}\big[a_{ij}(0)\big]$.

As mentioned before, we will prove in this section that, same as the maximal matching algorithms, QPS-1 is stable *under any traffic arrival process* $A(t)$ whose rate matrix Λ satisfies $\rho < 1/2$ (*i.e.*, can provably attain at least 50% throughput, or half of the maximum), where ρ (defined in (1.1)) is the maximum load factor imposed on any input or output port by the traffic matrix Λ. We also derive the average delay bound for QPS-1, which we show is *order-optimal* (*i.e.*, independent of switch size N).

Similar to $Q^{\dagger}_{ij}(t)$, we define $A^{\dagger}_{ij}(t)$ as the total number of packet arrivals to all VOQs in the neighborhood set $Q^{\dagger}_{ij}$:

$$A^{\dagger}_{ij}(t) \triangleq A_{i*}(t) - a_{ij}(t) + A_{*j}(t), \tag{5.3}$$

where $A_{i*}(t)$ and $A_{*j}(t)$ are similarly defined as $Q_{i*}(t)$ and $Q_{*j}(t)$ respectively. $D^{\dagger}_{ij}(t)$, $D_{i*}(t)$, and $D_{*j}(t)$ are similarly defined, so is $\Lambda^{\dagger}_{ij}(t)$. We now state some simple facts concerning $D(t)$, $A(t)$, and Λ as follows.

Fact 5.2.1. Given any crossbar scheduling algorithm, for any i, j, we have, $D_{i*}(t) \leq 1$ (at most one packet can depart from input port i during time slot t), $D_{*j}(t) \leq 1$, and $D^{\dagger}_{ij}(t) \leq 2$.

Fact 5.2.2. Given any *i.i.d.* arrival process $A(t)$ and its rate matrix is Λ whose maximum load factor is defined in (1.1), for any i, j, we have $\mathbb{E}[A^{\dagger}_{ij}(t)] = \Lambda^{\dagger}_{ij} \leq 2\rho$.

The following fact is slightly less obvious.

Fact 5.2.3. Given any crossbar scheduling algorithm, for any i, j, we have

$$d_{ij}(t) D^{\dagger}_{ij}(t) = d_{ij}(t). \tag{5.4}$$

Fact 5.2.3 holds because, as mentioned earlier, no other VOQ in $Q_{ij}^{\dagger}$ (see Figure 5.1) can be active simultaneously with q_{ij}. More precisely, if $d_{ij}(t) = 1$ (*i.e.*, VOQ q_{ij} is active during time slot t) then $D_{ij}^{\dagger}(t) \triangleq D_{i*}(t) - d_{ij}(t) + D_{*j}(t) = 1 - 1 + 1 = 1$; otherwise $d_{ij}(t)D_{ij}^{\dagger}(t) = 0 \cdot D_{ij}^{\dagger}(t) = 0 = d_{ij}(t)$.

5.2.2 Why QPS-1 Is Just as Good?

The provable throughput and delay bounds of maximal matching algorithms were derived from a "departure inequality" (to be stated and proved next) that all maximal matchings satisfy. This inequality, however, is not in general satisfied by matchings generated by QPS-1. Rather, QPS-1 satisfies a much weaker form of departure inequality, which we discover is fortunately barely strong enough for proving the same throughput and delay bounds.

Property 5.2.1 (Departure Inequality, stated as Lemma 1 in [20, 19])**.** If during a time slot t, the crossbar schedule is a *maximal matching*, then each departure process $D_{ij}^{\dagger}(t)$ satisfies the following inequality

$$q_{ij}(t)D_{ij}^{\dagger}(t) \geq q_{ij}(t). \tag{5.5}$$

Proof: We reproduce the proof of Property 5.2.1 with a slightly different approach for this thesis to be self-contained. Suppose the contrary is true, *i.e.*, $q_{ij}(t)D_{ij}^{\dagger}(t) < q_{ij}(t)$. This can only happen when $q_{ij}(t) > 0$ and $D_{ij}^{\dagger}(t) = 0$. However, $D_{ij}^{\dagger}(t) = 0$ implies that no nonempty VOQ (edge) in the neighborhood $Q_{ij}^{\dagger}$ (see Figure 5.1) is a part of the matching. Then this matching cannot be maximal (a contradiction) since it can be enlarged by the addition of the nonempty VOQ (edge) q_{ij}. ■

Clearly, the departure inequality (5.5) above implies the following much weaker form of it:

$$\sum_{i,j} \mathbb{E}\big[q_{ij}(t)D_{ij}^{\dagger}(t)\big] \geq \sum_{i,j} \mathbb{E}\big[q_{ij}(t)\big]. \tag{5.6}$$

In the rest of this section, we prove the following lemma:

Lemma 5.2.1. The matching generated by QPS-1, during any time slot t, satisfies the much weaker "departure inequality" (5.6).

Before we prove Lemma 5.2.1, we introduce an important definition and state four facts about QPS-1 that will be used later in the proof. In the following, we will run into several innocuous possible $\frac{0}{0}$ situations that all result from queue-proportional sampling, and we consider all of them to be 0.

We define $\alpha_{ij}(t)$ as the probability of the event that the proposal from input port i to output port j is accepted during the accepting phase, conditioned upon the event that input port i did propose to output port j during the proposing phase. With this definition, we have the first fact

$$\mathbb{E}\big[d_{ij}(t) \mid Q(t)\big] = \frac{q_{ij}(t)}{Q_{i*}(t)} \cdot \alpha_{ij}(t), \tag{5.7}$$

since both sides (note $d_{ij}(t)$ is a 0-1 random variable) are the probability that i proposes to j and this proposal is accepted. Summing over j on both sides, we obtain the second fact

$$\mathbb{E}\big[D_{i*}(t) \mid Q(t)\big] = \sum_j \frac{q_{ij}(t)}{Q_{i*}(t)} \cdot \alpha_{ij}(t). \tag{5.8}$$

The third fact is that, for any output port j,

$$\mathbb{E}\big[D_{*j}(t) \mid Q(t)\big] = 1 - \prod_i \big(1 - \frac{q_{ij}(t)}{Q_{i*}(t)}\big). \tag{5.9}$$

In this equation, the LHS is the conditional *probability* ($D_{*j}(t)$ is also a 0-1 random variable) that at least one proposal is received and accepted by output port j, and the second term on the RHS of (5.9) is the probability that no input port proposes to output port j (so j receives no proposal). This equation holds since when j receives one or more proposals, it will accept one of them (the one with the longest VOQ).

The fourth fact is that, for any i, j,

$$\alpha_{ij}(t) \geq \prod_{k \neq i} \big(1 - \frac{q_{kj}(t)}{Q_{k*}(t)}\big). \tag{5.10}$$

This inequality holds because when input port i proposes to output port j, and no other input port does, j has no choice but to accept $i's$ proposal.

5.2.3 Proof of Lemma 5.2.1

Now we are ready to prove Lemma 5.2.1. It suffices to show that for any i and j, we have

$$\sum_{i,j} \mathbb{E}\big[q_{ij}(t) D_{ij}^{\dagger}(t) \mid Q(t)\big] \geq \sum_{i,j} q_{ij}(t) \tag{5.11}$$

because with (5.11), we have

$$\begin{aligned}
&\sum_{i,j} \mathbb{E}\big[q_{ij}(t) D_{ij}^{\dagger}(t)\big] \\
=&\mathbb{E}\Big[\mathbb{E}\big[\sum_{i,j} q_{ij}(t) D_{ij}^{\dagger}(t) \mid Q(t)\big]\Big] \\
\geq&\mathbb{E}\big[\sum_{i,j} q_{ij}(t)\big] \\
=&\sum_{i,j} \mathbb{E}\big[q_{ij}(t)\big].
\end{aligned}$$

By the definition of $D_{ij}^{\dagger}(t) \triangleq D_{i*}(t) - d_{ij}(t) + D_{*j}(t)$, we have,

$$\begin{aligned}
&\sum_{i,j} \mathbb{E}\big[q_{ij}(t) D_{ij}^{\dagger}(t) \mid Q(t)\big] \\
=&\sum_{i,j} q_{ij}(t) \mathbb{E}\big[D_{i*}(t) \mid Q(t)\big] - \sum_{i,j} q_{ij}(t) \mathbb{E}\big[d_{ij}(t) \mid Q(t)\big] \\
&+ \sum_{i,j} q_{ij}(t) \mathbb{E}\big[D_{*j}(t) \mid Q(t)\big].
\end{aligned} \tag{5.12}$$

Focusing on the first term on the RHS of (5.12) and using (5.8), we have,

$$\begin{aligned}
&\sum_{i,j} q_{ij}(t) \mathbb{E}\big[D_{i*}(t) \mid Q(t)\big] \\
=&\sum_{i} Q_{i*}(t) \mathbb{E}\big[D_{i*}(t) \mid Q(t)\big] \\
=&\sum_{i} Q_{i*}(t) \Big(\sum_{j} \frac{q_{ij}(t)}{Q_{i*}(t)} \cdot \alpha_{ij}(t)\Big) \\
=&\sum_{i,j} q_{ij}(t) \alpha_{ij}(t).
\end{aligned} \tag{5.13}$$

Focusing the second term on the RHS of (5.12) and using (5.7), we have

$$-\sum_{i,j} q_{ij}(t) \mathbb{E}\big[d_{ij}(t) \mid Q(t)\big] = -\sum_{i,j} q_{ij}(t) \alpha_{ij}(t) \frac{q_{ij}(t)}{Q_{i*}(t)}. \tag{5.14}$$

Hence, the sum of the first two terms in (5.12) is equal to

$$\begin{aligned}
&\sum_{i,j} q_{ij}(t)\alpha_{ij}(t)\Big(1-\frac{q_{ij}(t)}{Q_{i*}(t)}\Big) \\
\geq &\sum_{i,j} q_{ij}(t)\Big(\prod_{k\neq i}\big(1-\frac{q_{kj}(t)}{Q_{k*}(t)}\big)\Big)\big(1-\frac{q_{ij}(t)}{Q_{i*}(t)}\big) \qquad (5.15)\\
= &\sum_{i,j} q_{ij}(t)\prod_{i}\big(1-\frac{q_{ij}(t)}{Q_{i*}(t)}\big) \\
= &\sum_{i,j} q_{ij}(t)\Big(1-\mathbb{E}\big[D_{*j}(t)\mid Q(t)\big]\Big). \qquad (5.16)
\end{aligned}$$

Note that (5.15) is due to (5.10) and (5.16) is due to (5.9). We now arrive at (5.11), when adding the third and last term in (5.12) to the RHS of (5.16).

5.2.4 Throughput Analysis

In this section we prove, through Lyapunov stability analysis, the following theorem (*i.e.,* Theorem 5.2.1) which states that any crossbar scheduling algorithm that satisfies the weaker departure inequality (5.6), including QPS-1 as shown in Lemma 5.2.1, can attain at least 50% throughput. The same throughput bound was proved in [21], through fluid limit analysis, for maximal matching algorithms using the (stronger) departure inequality (5.5) which as stated earlier is not in general satisfied by matchings generated by QPS-1.

Theorem 5.2.1. Let $\{Q(t)\}_{t=0}^{\infty}$ be the queueing process of a switching system that is an irreducible Markov chain. Let the departure process of $\{Q(t)\}_{t=0}^{\infty}$ satisfy the weaker "departure inequality" (5.6). Then whenever its maximum load factor $\rho < 1/2$, the queueing process is stable in the following sense: (I) The Markov chain $\{Q(t)\}_{t=0}^{\infty}$ is positive recurrent and hence converges to a stationary distribution $\bar{Q}$; (II) The first moment of $\bar{Q}$ is finite.

Proof. Here we prove only (I), since Theorem 5.2.2 that we will shortly prove implies (II). We define the following Lyapunov function of $Q(t)$: $L\big(Q(t)\big) = \sum_{i,j} q_{ij}(t)Q_{ij}^{\dagger}(t)$, where $Q_{ij}^{\dagger}(t)$ is defined earlier in (5.1). This Lyapunov function was first introduced in [19] for

the delay analysis of maximal matching algorithms for wireless networking. By the Foster-Lyapunov stability criterion [71, Proposition 2.1.1], to prove that $\{Q(t)\}_{t=0}^{\infty}$ is positive recurrent, it suffices to show that, there exists a constant $B>0$ such that whenever the total queue (VOQ) length $\|Q(t)\|_1 > B$ (because it is not hard to verify that the complement set of states $\{Q(t) : \|Q(t)\|_1 \leq B\}$ is finite and the drift is bounded whenever $Q(t)$ belongs to this set), we have

$$\mathbf{E}\big[L\big(Q(t+1)\big) - L\big(Q(t)\big) \mid Q(t)\big] \leq -\epsilon, \tag{5.17}$$

where $\epsilon > 0$ is a constant. It is not hard to check (for more detailed derivations, please refer to [19]),

$$\begin{aligned} &L\big(Q(t+1)\big) - L\big(Q(t)\big) \\ =&2\sum_{i,j} q_{ij}(t)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big) + \sum_{i,j}\big(a_{ij}(t) - d_{ij}(t)\big)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big). \end{aligned} \tag{5.18}$$

Hence the drift (LHS of (5.17)) can be written as

$$\begin{aligned} &\mathbf{E}\big[L\big(Q(t+1)\big) - L\big(Q(t)\big) \mid Q(t)\big] \\ =&\mathbf{E}\Big[2\sum_{i,j} q_{ij}(t)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big) \mid Q(t)\Big] \\ &\quad + \mathbf{E}\Big[\sum_{i,j}\big(a_{ij}(t) - d_{ij}(t)\big)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big) \mid Q(t)\Big]. \end{aligned} \tag{5.19}$$

Now we claim the following two inequalities, which we will prove shortly.

$$\mathbf{E}\big[2\sum_{i,j} q_{ij}(t)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big) \mid Q(t)\big] \leq 2(2\rho - 1)\|Q(t)\|_1. \tag{5.20}$$

$$\mathbf{E}\big[\sum_{i,j}\big(a_{ij}(t) - d_{ij}(t)\big)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big) \mid Q(t)\big] \leq CN^2. \tag{5.21}$$

With (5.20) and (5.21) substituted into (5.19), we have

$$\mathbf{E}\big[L\big(Q(t+1)\big) - L\big(Q(t)\big) \mid Q(t)\big] \leq 2(2\rho - 1)\|Q(t)\|_1 + CN^2.$$

where $C > 0$ is a constant. Since $\rho < 1/2$, we have $2\rho - 1 < 0$. Hence, there exist $B, \epsilon > 0$ such that, whenever $\|Q(t)\|_1 > B$,

$$\mathbf{E}\big[L\big(Q(t+1)\big) - L\big(Q(t)\big) \mid Q(t)\big] \leq -\epsilon.$$

Now we proceed to prove (5.20).

$$\begin{aligned}
&\mathbf{E}\Big[2\sum_{i,j} q_{ij}(t)\big(A^{\dagger}_{ij}(t) - D^{\dagger}_{ij}(t)\big) \mid Q(t)\Big] \\
=&2\Big(\sum_{i,j}\mathbf{E}\Big[q_{ij}(t)A^{\dagger}_{ij}(t) \mid Q(t)\Big] - \sum_{i,j}\mathbf{E}\Big[q_{ij}(t)D^{\dagger}_{ij}(t) \mid Q(t)\Big]\Big) \\
\leq&2\Big(2\rho\sum_{i,j}\mathbf{E}\Big[q_{ij}(t) \mid Q(t)\Big] - \sum_{i,j}\mathbf{E}[q_{ij}(t) \mid Q(t)]\Big) \qquad (5.22)\\
=&2(2\rho-1)\|Q(t)\|_1. \qquad (5.23)
\end{aligned}$$

In the above derivations, inequality (5.22) holds due to (5.11), $A(t)$ being independent of $Q(t)$ for any t, and Fact 5.2.2 that $\mathbf{E}[A^{\dagger}_{ij}(t)] \leq 2\rho$.

Now we proceed to prove (5.21), which upper-bounds the conditional expectation $\mathbf{E}\big[\big(a_{ij}(t) - d_{ij}(t)\big)\big(A^{\dagger}_{ij}(t) - D^{\dagger}_{ij}(t)\big) \mid Q(t)\big]$. It suffices however to upper-bound the unconditional expectation $\mathbf{E}\big[\big(a_{ij}(t) - d_{ij}(t)\big)\big(A^{\dagger}_{ij}(t) - D^{\dagger}_{ij}(t)\big)\big]$, which we will do in the following, since we can obtain the same upper bounds on $\mathbf{E}[D^{\dagger}_{ij}(t)]$ and $\mathbf{E}[d_{ij}(t)]$ (2 and 1 respectively) whether the expectations are conditional (on $Q(t)$) or not. Note the other two terms $A^{\dagger}_{ij}(t)$ and $a_{ij}(t)$ are independent of (the condition) $Q(t)$.

As for any i, j, $a_{ij}(t)$ is *i.i.d.*, we have,

$$\begin{aligned}
&\mathbf{E}\big[\big(a_{ij}(t) - d_{ij}(t)\big)\big(A^{\dagger}_{ij}(t) - D^{\dagger}_{ij}(t)\big)\big] \qquad (5.24)\\
=&\mathbf{E}[a_{ij}(t)A^{\dagger}_{ij}(t) - d_{ij}(t)A^{\dagger}_{ij}(t) - a_{ij}(t)D^{\dagger}_{ij}(t) + d_{ij}(t)D^{\dagger}_{ij}(t)] \\
=&\mathbf{E}[a^2_{ij}(t)] - \lambda^2_{ij} + \lambda_{ij}\Lambda^{\dagger}_{ij} - \mathbf{E}[d_{ij}(t)]\Lambda^{\dagger}_{ij} - \lambda_{ij}\mathbf{E}[D^{\dagger}_{ij}(t)] + \mathbf{E}[d_{ij}(t)D^{\dagger}_{ij}(t)] \\
=&\mathbf{E}[a^2_{ij}(t)] - \lambda^2_{ij} + \lambda_{ij}\Lambda^{\dagger}_{ij} - \mathbf{E}[d_{ij}(t)]\Lambda^{\dagger}_{ij} - \lambda_{ij}\mathbf{E}[D^{\dagger}_{ij}(t)] + \mathbf{E}[d_{ij}(t)]. \qquad (5.25)
\end{aligned}$$

In arriving at (5.25), we have used (5.2). The RHS of (5.25) can be bounded by a constant $C > 0$ due to the following assumptions and facts: $\mathbf{E}[a^2_{ij}(t)] = \mathbf{E}[a^2_{ij}(0)] < \infty$ for any t, $d_{ij}(t) \leq 1$, $D^{\dagger}_{ij}(t) \leq 2$, $\lambda_{ij} \leq \rho < 1/2$, and $\Lambda^{\dagger}_{ij} \leq 2\rho < 1$. Therefore, we have (by applying $\sum_{i,j}$ to both (5.24) and the RHS of (5.25))

$$\sum_{i,j}\mathbf{E}\Big[\big(a_{ij}(t) - d_{ij}(t)\big)\big(A^{\dagger}_{ij}(t) - D^{\dagger}_{ij}(t)\big)\Big] \leq CN^2.$$

□

Remarks. Now that we have proved that $\{Q(t)\}_{t=0}^{\infty}$ is positive recurrent. Hence, we have, in steady state, for any $1 \leq i, j \leq N$, $\mathbf{E}[d_{ij}(t)] = \lambda_{ij}$. Therefore, we have, in steady state, for any $1 \leq i, j \leq N$

$$\mathbf{E}\big[\big(a_{ij}(t) - d_{ij}(t)\big)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big)\big] = \sigma_{ij}^2 - \lambda_{ij}\Lambda_{ij}^{\dagger} + \lambda_{ij}, \tag{5.26}$$

where $\sigma_{ij}^2 = \mathbf{E}[a_{ij}^2(t)] - \lambda_{ij}^2$ is the variance of $a_{ij}(t)$, because (5.25) can be simplified as the RHS of (5.26) in steady state.

The following lemma (Lemma 5.2.2), in combination with Lemma 5.2.1, shows that Theorem 5.2.1 applies to QPS-1. Therefore, QPS-1 can attain at least 50% throughput under any *i.i.d.* arrival process.

Lemma 5.2.2. Under any *i.i.d.* arrival process whose maximum load factor $\rho < 1/2$, when using QPS-1 as the crossbar scheduling algorithm, the resulting queueing process $\{Q(t)\}_{t=0}^{\infty}$ is an irreducible Markov chain.

Proof. $\{Q(t)\}_{t=0}^{\infty}$ is clearly a Markov chain, since in (5.2), the term $d_{ij}(t)$ is a function of $Q(t)$ and $a_{ij}(t)$ is a random variable independent of $Q(t)$. The reasoning for the irreducibility of this Markov is the same as in §B.3. □

5.2.5 Delay Analysis

In this section, we derive the bound on the expected total queue length $\mathbf{E}[\|\bar{Q}\|_1]$ (readily convertible to the corresponding delay bound using Little's Law) for QPS-1 under *i.i.d.* traffic arrivals using the following moment bound lemma (*i.e.,* Lemma 5.2.3) [71, Proposition 2.1.4]. This bound, shown in (5.28), is identical to that derived in [19, 20, Section III.B] for maximal matchings under *i.i.d.* traffic arrivals. Note this equivalence is not limited to *i.i.d.* traffic arrivals: It can be shown that the delay analysis results for general Markovian arrivals derived in [19, 20] for maximal matchings (using the stronger "departure inequality" (5.5)) hold also for QPS-1.

Lemma 5.2.3. Suppose that $\{Y_t\}_{t=0}^{\infty}$ is a positive recurrent Markov chain with countable state space $\mathcal{Y}$. Suppose V, f, and g are non-negative functions on $\mathcal{Y}$ such that,

$$V(Y_{t+1}) - V(Y_t) \leq -f(Y_t) + g(Y_t), \text{ for all } Y_t \in \mathcal{Y}. \tag{5.27}$$

Then $\mathbf{E}[f(\bar{Y})] \leq \mathbf{E}[g(\bar{Y})]$, where $\bar{Y}$ is a random variable with the stationary distribution of the Markov chain $\{Y_t\}_{t=0}^{\infty}$.

Now we derive the following bound on $\mathbf{E}[\|\bar{Q}\|_1]$, which is stronger than the part (II) of Theorem 5.2.1.

Theorem 5.2.2. Under the same assumptions and definitions as in Theorem 5.2.1, we have

$$\mathbf{E}[\|\bar{Q}\|_1] \leq \frac{1}{2(1-2\rho)} \sum_{i,j} \big(\sigma_{ij}^2 - \lambda_{ij}\Lambda_{ij}^{\dagger} + \lambda_{ij}\big). \tag{5.28}$$

Proof. We define function V in Lemma 5.2.3 as L, the Lyapunov function used in the proof of Theorem 5.2.1 and Y_t as the queue length matrix $Q(t)$. Let $f(Y_t) \triangleq -2\sum_{i,j} q_{ij}(t)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big) + h(Y_t)$, where $h(Y_t) \triangleq 2\sum_{i,j} q_{ij}(t)A_{ij}^{\dagger}(t) + \sum_{i,j} a_{ij}(t)D_{ij}^{\dagger}(t)$, and $g(Y_t) \triangleq \sum_{i,j}\big(a_{ij}(t) - d_{ij}(t)\big)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big) + h(Y_t)$. It is not hard to check that both $f(\cdot)$ and $g(\cdot)$ are non-negative functions and inequality (5.27) holds for V, Y_t, f, g defined above. Furthermore, we have proved before, $\{Q(t)\}_{t=0}^{\infty}$ is a positive Markov chain whenever the maximum load factor $\rho < 1/2$. Hence, we have, in steady state,

$$\begin{aligned}
&-2(2\rho-1)\mathbf{E}[\|\bar{Q}\|_1] \\
\leq &\mathbf{E}\big[-2\sum_{i,j} q_{ij}(t)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big)\big] &(5.29)\\
= &\mathbf{E}[f(\bar{Y})] - \mathbf{E}[h(\bar{Y})] \\
\leq &\mathbf{E}[g(\bar{Y})] - \mathbf{E}[h(\bar{Y})] &(5.30)\\
= &\mathbf{E}\big[\big(a_{ij}(t) - d_{ij}(t)\big)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big)\big] \\
= &\sum_{i,j}\big(\sigma_{ij}^2 - \lambda_{ij}\Lambda_{ij}^{\dagger} + \lambda_{ij}\big). &(5.31)
\end{aligned}$$

In the above derivation, inequality (5.29) is because taking expectation on both sides of (5.20), we have $\mathbf{E}\big[2\sum_{i,j} q_{ij}(t)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big)\big] \leq 2(2\rho - 1)\mathbf{E}[\|Q(t)\|_1]$, thus, in steady state,

$$-2(2\rho - 1)\mathbf{E}[\|\bar{Q}\|_1] \leq \mathbf{E}\big[-2\sum_{i,j} q_{ij}(t)\big(A_{ij}^{\dagger}(t) - D_{ij}^{\dagger}(t)\big)\big].$$

Inequality (5.30) is due to Lemma 5.2.3, and equality (5.31) is due to (5.26).

Therefore, we have, in steady state,

$$\mathbf{E}[\|\bar{Q}\|_1] \leq \frac{1}{2(1-2\rho)} \sum_{i,j} \big(\sigma_{ij}^2 - \lambda_{ij}\Lambda_{ij}^{\dagger} + \lambda_{ij}\big).$$

□

Since, as explained in the proof of Lemma 5.2.2, $\{Q(t)\}_{t=0}^{\infty}$ is an irreducible Markov chain under *i.i.d.* arrivals when the maximum load factor $\rho < 1/2$, Theorem 5.2.2 applies to QPS-1. Hence we obtain,

Corollary 5.2.1. The bound on $\mathbf{E}[\|\bar{Q}\|_1]$ as stated in (5.28) holds under QPS-1 scheduling, whenever the arrival process is *i.i.d.* and the maximum load factor $\rho < 1/2$.

It is not hard to check (by applying Little's Law) that the average delay (experienced by packets) is bounded by a constant independent of N (*i.e., order-optimal*) for a given maximum load factor $\rho < 1/2$, since the variance $\sigma_{ij}^2 = \mathbf{E}[a_{ij}^2(0)] - \lambda_{ij}^2$ for any i, j is finite (as we have assumed in §5.2.1 that the second moment $\mathbf{E}[a_{ij}^2(0)]$ is finite). For the special case of *i.i.d.* Bernoulli arrivals (when $\sigma_{ij}^2 = \lambda_{ij} - \lambda_{ij}^2$), this bound (the RHS) can be further tightened to $\frac{\sum_{i,j}\lambda_{ij}}{1-2\rho}$. This implies, by Little's Law, the following "clean" bound: $\bar{\omega} \leq \frac{1}{1-2\rho}$ where $\bar{\omega}$ is the expected delay averaged over all packets transmitting through the switch.

5.3 Performance Evaluation

In this section, we evaluate, through simulations, the performance of QPS-r under various load conditions and traffic patterns. We compare its performance with that of iSLIP [5], a refined and optimized representative parallel maximal matching algorithm (adapted for

switching). The performance of the MWM (Maximum Weighted Matching) is also included in the comparison as a benchmark. Our simulations show conclusively that QPS-1 performs very well inside the provable stability region (more precisely, with no more than 50% offered load), and that QPS-3 has comparable throughput and delay performances as iSLIP, which has much more expensive computationally.

5.3.1 Simulation Setup

In our simulations, we first fix the number of input/output ports, N to 64. Later, in section 5.3.3 we investigate how the mean delay performances of these algorithms scale with respect to N. To measure throughput and delay accurately, we assume each VOQ has an infinite buffer size and hence there is no packet drop at any input port. Each simulation run is guided by the following stopping rule [72, 73]: The number of time slots simulated is the larger between $500N^2$ and that is needed for the difference between the estimated and the actual average delays to be within 0.01 time slots with probability at least 0.98.

We assume in our simulations that each traffic arrival matrix $A(t)$ is *i.i.d.* Bernoulli with its traffic rate matrix Λ being equal to the product of the offered load and a traffic pattern matrix (defined in §1.2.6). Similar Bernoulli arrivals were studied in [4, 5]. Later, we will also look at bursty arrivals in §5.3.4. Note that only synthetic traffic (instead of that derived from packet traces) is used in our simulations because, to the best of our knowledge, there is no meaningful way to combine packet traces into switch-wide traffic workloads. As mentioned earlier in §1.2.6, the four standard types of normalized (with each row or column sum equal to 1) traffic patterns are used.

5.3.2 Throughput and Delay Performances

We first compare the throughput and delay performances of QPS-1 (1 iteration), QPS-3 (3 iterations), iSLIP ($\log_2 64$ = 6 iterations), and MWM (length of VOQ as the weight measure). Figure 5.2 shows their mean delays (in number of time slots) under the afore-

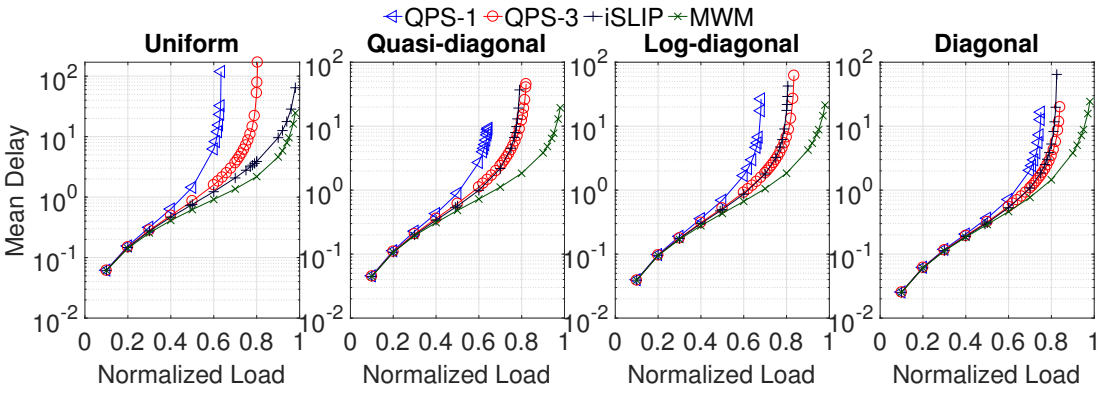


Figure 5.2: Mean delays of QPS-1, QPS-3, iSLIP, and MWM under the 4 traffic patterns.

mentioned four traffic patterns respectively. Each subfigure shows how the mean delay (on a *log scale* along the y-axis) varies with the offered load (along the x-axis). We make three observations from Figure 5.2. First, Figure 5.2 clearly shows that, when the offered load is no larger than 0.5, QPS-1 has low average delays (*i.e.*, more than just being stable) that are close to those of iSLIP and MWM, under all four traffic patterns. Second, the maximum sustainable throughputs (where the delays start to "go through the roof" in the subfigures) of QPS-1 are roughly $0.634, 0.645, 0.681$, and 0.751 respectively, under the four traffic patterns respectively; they are all comfortably larger than the 50% provable lower bound. Third, the throughput and delay performances of QPS-3 and iSLIP are comparable: The former has slightly better delay performances than the latter under all four traffic patterns except the uniform.

5.3.3 How Mean Delay Scales with N

Figure 5.3 shows how the mean delays of QPS-3, iSLIP (running $\log_2 N$ iterations given any N), and MWM scale with the number of input/output ports N, under the four different traffic patterns. With one exception, we have simulated the following different values of N: $N = 8, 16, 32, 64, 128, 256, 512, 1{,}024$. The exception is that we did not obtain the delay values for MWM (not a "main character" in our story) for $N = 1,024$, as it proved to be prohibitively expensive computationally to do so. In all these plots, the offered load is 0.75, which is quite high compared to the maximum sustainable throughputs of QPS-3 and iSLIP

(shown in Figure 5.2) under these four traffic patterns. Figure 5.3 shows that the mean delays of QPS-3 are slightly lower (*i.e.*, better) than those of iSLIP under all traffic patterns except the uniform. In addition, the mean delay curves of QPS-3 remain almost flat (*i.e.*, constant) under log-diagonal and diagonal traffic patterns. Although they increase with N under uniform and quasi-diagonal traffic patterns, they eventually almost flatten out when N gets larger (say when $N \geq 128$). These delay curves show that QPS-3, which runs only 3 iterations, deliver slightly better delay performances, under all 4 traffic patterns except the uniform, than iSLIP (a refined and optimized parallel maximal matching algorithm adapted for switching), which runs $\log_2 N$ iterations with each iteration has $O(\log_2 N)$ time complexity.

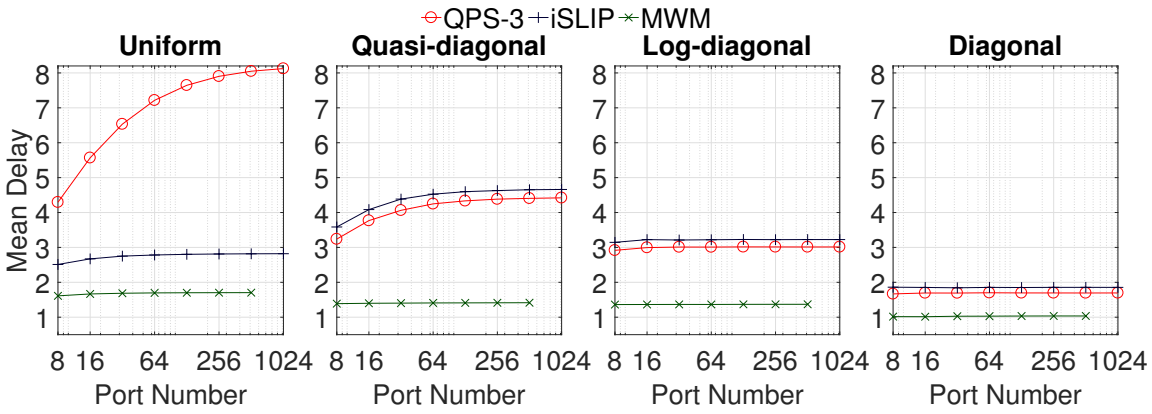


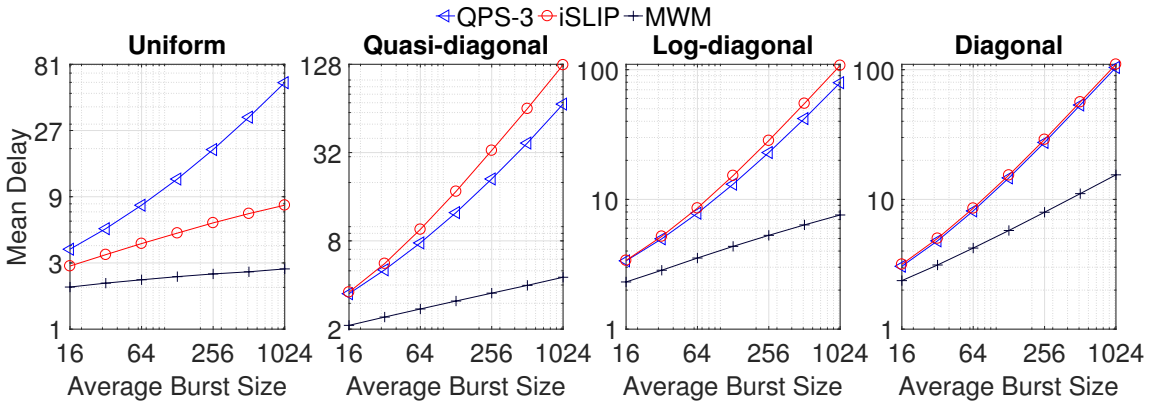
Figure 5.3: Mean delays scaling with number of ports N for QPS-3, iSLIP, and MWM.

5.3.4 Bursty Arrivals

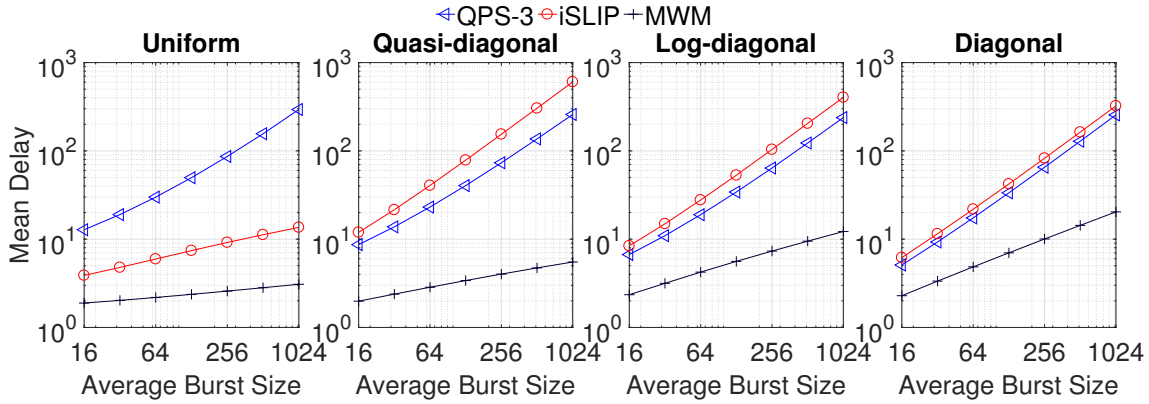
In real networks, packet arrivals are likely to be bursty. In this section, we evaluate the performances of QPS-3, iSLIP and MWM under bursty traffic arrivals, generated by a two-state ON-OFF arrival process that we have described in §4.4.3.2.

We evaluate their mean delay performances with the average burst size ranging from 16 to 1,024 packets. We have simulated various offered loads, but here we only present the results, shown in Figure 5.4, under an offered load of 0.49 (Figure 5.4a) and that of 0.75 (Figure 5.4b). Figure 5.4 clearly shows that QPS-3 outperforms iSLIP (under all traffic

patterns except the uniform), by an increasingly wider margin in both absolute and relative terms as the average burst size becomes larger, under both offered loads.



(a) Offered load: 0.49



(b) Offered load: 0.75

Figure 5.4: Mean delays under bursty traffic arrivals with the 4 traffic patterns.

5.4 Conclusion

In this chapter, we propose QPS-r, a parallel iterative crossbar scheduling algorithm with $O(1)$ time complexity per port. We prove, through Lyapunov stability analysis, that it achieves the same throughput and delay guarantees in theory, and demonstrate through simulations that it has comparable performances in practice as the family of maximal matching algorithms (adapted for switching); maximal matching algorithms are much more expensive computationally (at least $O(\log N)$ iterations and a total of $O(\log^2 N)$ per-port time

complexity). These salient properties make QPS-r an excellent candidate algorithm that is fast enough computationally and can deliver acceptable throughput and delay performances for high-link-rate high-radix switches.

CHAPTER 6

SB-QPS

6.1 Batch Scheduling Algorithms

Since Small-Batch QPS (SB-QPS) is a batch scheduling algorithm [22, 23, 24], we first provide some background on batch scheduling. In a batch scheduling algorithm, the T matchings for a batch of T future time slots are batch-computed.

	O_1	O_2	$\cdots$	O_N
1	I_3	I_7	$\cdots$	I_1
2	I_5	–	$\cdots$	I_3
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
T	–	I_5	$\cdots$	I_2

Figure 6.1: A joint calendar. "–" means unmatched.

These T matchings form a joint calendar (schedule) of the N output ports that can be encoded as a $T \times N$ table with TN cells in it, as illustrated by an example shown in Figure 6.1. Each column corresponds to the calendar of an output port and each row a time slot. The content of the cell at the intersection of the t^{th} row and the j^{th} column is the input port that O_j is to pair with during the t^{th} time slot in this batch. Hence, each cell also corresponds to an edge (between the input and the output port pair) and each row also corresponds to a matching (under computation for the corresponding time slot). In the example shown in Figure 6.1, output port O_1 is to pair with I_3 during the 1^{st} time slot (in this batch), I_5 during the 2^{nd} time slot, and is unmatched during the T^{th} time slot.

At each input port, all packets that were in queue before a cutoff time (for the current batch), including those that belong to either the current batch, or previous batches but could not be served then, are waiting to be inserted into the respective calendars (*i.e.,* columns of cells) of the corresponding output ports. The design objective of a batch scheduling

algorithm is to pack as many such packets across the N input ports as possible into the TN cells in this joint calendar. After the computation of the current joint calendar is completed, the T matchings in it will be used as the crossbar configurations for a batch of T future time slots. In the meantime, the switch is switching packets according to the T matchings specified in a past joint calendar that was computed earlier.

6.2 The SB-QPS Algorithm

In this section, we describe in detail SB-QPS, a batch scheduling algorithm that uses a small constant batch size T that is independent of N. SB-QPS is a parallel iterative algorithm: The input and output ports run T QPS-like iterations (request-accept message exchanges) to collaboratively pack the joint calendar. The operation of each iteration is extremely simple: Input ports request for cells in the joint calendar, and output ports accept or reject the requests. More precisely, each iteration of SB-QPS, like that of QPS (see Chapter 4), consists of two phases: a proposing phase and an accepting phase.

Proposing Phase. We adopt the same proposing strategy as in QPS: In this phase, each input port, unless it has no packet to transmit, proposes to *exactly one* output port that is decided by the QPS strategy. Here, we will only describe the operations at input port 1; those at any other input port are identical. Like in §4.1, we denote by $m_1, m_2, \cdots, m_N$ the respective queue lengths of the N VOQs at input port 1, and by m their sum (*i.e.*, $m \triangleq \sum_{k=1}^{N} m_k$). At first, input port 1 simply samples an output port j with probability m_j/m, *i.e.,* proportional to m_j, the length of the corresponding VOQ; it then sends a proposal to output port j. The content of the proposal in SB-QPS is slightly different than that in QPS. In QPS, the proposal contains only the VOQ length information (*i.e.,* the value m_j), whereas in SB-QPS, it contains also the following availability information (of input port 1): Out of the T time slots in the batch, what (time slots) are still available for input port 1 to pair with an output port? The time complexity of this QPS operation, carried out using the data structure described in §4.2, is $O(1)$ per input port.

Accepting Phase. The accepting phase (in SB-QPS) at an output port is quite different than that in QPS. Whereas the latter allows at most one proposal to be accepted at any output port (as QPS is a part of a regular crossbar scheduling algorithm that is concerned with only a single time slot at a time), the former allows an output port to accept multiple (up to T) proposals (as each output port has up to T cells in its calendar to be filled). Here, we describe the accepting phase at output port 1; that at any other output port is identical. The operations at output port 1 depend on the number of proposals it receives. If output port 1 receives exactly one proposal from an input port (say input port i), it tries to accommodate this proposal using an accepting strategy we call *First Fit Accepting* (FFA). The FFA strategy is to match in this case input port i and output port 1 at the earliest time slot (in the batch of T time slots) during which both are still available (for pairing); if they have "schedule conflicts" over all T time slots, this proposal is rejected. If output port 1 receives proposals from multiple input ports, then it first sorts (with ties broken arbitrarily) these proposals in a descending order according to their corresponding VOQ lengths, and then tries to accept each of them using the FFA strategy.

In SB-QPS, opportunities – in the form of proposals from input ports – can arise, throughout the time window (up to T time slots long) for computing the join calendar, to fill any of its TN cells. As explained earlier, this "capturing every opportunity" to fill the joint calendar allows a batch scheduling algorithm to produce matchings of much higher qualities than a regular crossbar scheduling algorithm that is based on the same underlying bipartite matching algorithm can. Indeed, SB-QPS, the batch scheduling algorithm that is based on the QPS bipartite matching primitive, significantly outperforms QPS-1, the regular crossbar scheduling algorithm that is based on QPS, as we will show in §6.3.

Time Complexity. The time complexity for the accepting phase at an output port is $O(1)$ on average, although in theory it can be as high as $O(N \log N)$ since an output port can receive up to N proposals and have to sort them based on their corresponding VOQ lengths. Like in §4.1, this time complexity can be made $O(1)$ even in the worst case by letting the

output port drop ("knock out") all proposals except the earliest few (say 3) to arrive. In this work, we indeed set this threshold to 3 and find that it has a negligible effect on the quality of resulting matchings.

We now explain how to carry out an FFA operation in $O(1)$ time. In SB-QPS, we encode the availability information of an input port i as a T-bit-long bitmap $B_i[1..T]$, where $B_i[t] = 1$ if input port i is available (i.e., not already matched with an output port) at time slot t and $B_i[t] = 0$ otherwise. The availability information of an output port o is similarly encoded into a T-bit-long bitmap $B_o[1..T]$. When input port i sends a proposal, which contains the availability information $B_i[1..T]$, to output port o, the corresponding FFA operation is for the output port o to find the first bit in the bitmap $(B_i \& B_o)[1..T]$ that has value 1, where $\&$ denotes bitwise-AND. Since the batch size T in SB-QPS is a small constant (say T=32), both bitmaps can fit into a single CPU word and "finding the first 1" is an instruction on most modern CPUs.

To summarize, the worst-case time complexity of SB-QPS is $O(T)$ per input or output port for the joint calendar consisting of T matchings, since SB-QPS runs T iterations and each iteration has $O(1)$ worst-case time complexity per input or output port. Hence the worst-case time complexity for computing each matching is $O(1)$ per input or output port.

Message Complexity. The message complexity of each "propose-accept" iteration is $O(1)$ messages per input or output port, because each input port sends at most one proposing message per iteration and each output port sends out at most 3 acceptance messages (where 3 is the "knockout" threshold explained above). Each proposing message is $T + \lceil \log_2 W \rceil$ bits long (T bits for encoding the availability information and $\lceil \log_2 W \rceil$ bits for encoding the corresponding VOQ length), where W is the longest possible VOQ length. Each acceptance message is $\lceil log_2 T \rceil$ bits long (for encoding the time slot the pairing is to be made). To summarize, the worst-case message complexity of SB-QPS for computing each of the T matchings is $O(1)$ per port.

We have considered and experimented with two other accepting strategies. One is to

accept as many as possible proposals, which we refer to as *Maximum Fit Accepting* (MFA). The other is to maximize the total weight of accepted proposals that is defined as the total length of the VOQs between the input and output pairs corresponding to the accepted proposals, which we refer to as *Maximum Weight Fit Accepting* (MWFA). Unlike FFA that tries to accommodate proposals one after another, MFA and MWFA consider all proposals jointly and maximizes the number of or the weight of proposals that can be accepted. Intuitively, MFA and MWFA should produce higher-quality matchings (measured by the resulting throughput performances) than FFA. However, as we will show in §C.1.3, that FFA generally has better (throughput and delay) performances. Therefore, we prefer FFA.

6.3 Performance Evaluation

In this section, we evaluate, through simulations, the throughput and delay performances of SB-QPS under various load conditions and traffic patterns. SB-QPS is compared against Fair-Frame [23]. The batch size T of Fair-Frame is configured following the guidance provided in [23]. More precisely, $T = \lceil \log(2N/\delta_{min}) / \log(1/\gamma) \rceil$, where δ_{min} was chosen to minimize the delay bound, $\gamma \triangleq \rho e^{1-\rho}$, and ρ is the maximum load factor. Since ρ is usually not known in practice, we set ρ=0.9 in our simulations so that Fair-Frame is stable whenever the maximum load factor is no more than 0.9, *i.e.,* the maximum sustainable throughput of Fair-Frame is at least 0.9. This is fair because, as we will show later, the maximum sustainable throughputs of our SB-QPS are around 0.9 under all simulated traffic patterns. SB-QPS is also compared against QPS-1 (QPS-r with r=1 iteration). This is a fair comparison because QPS-1, like our SB-QPS, runs only a single iteration to compute a matching. The MWM algorithm (with the VOQ length as the weight measure), which, as mentioned in Chapter 2, delivers near-optimal delay performance, is also compared against as a benchmark.

6.3.1 Simulation Setup

In our simulations, we fix the number of input and output ports N to 64. Later, in §C.1.1, we investigate how the mean delay performances of these algorithms scale with respect to N. To accurately measure throughput and delay, we assume that each VOQ has an infinite buffer size, so no packet is dropped at any input port. Each simulation run is guided by the following stopping rule [72, 73]: The number of time slots simulated is the larger between $500N^2$ and that is needed for the difference between the estimated and the actual average delays to be within 0.01 with probability at least 0.98.

We assume in our simulations that each traffic arrival matrix is *i.i.d.* Bernoulli with its traffic rate matrix equal to the product of the offered load and a traffic pattern matrix. Similar Bernoulli arrivals were studied in [4, 5]. Later in §C.1.2, we will look at bursty traffic arrivals. Note that only synthetic traffic (instead of that derived from packet traces) is used in our simulations because, to the best of our knowledge, there is no meaningful way to combine packet traces into switch-wide traffic workloads. The four standard types of normalized (with each row or column sum equal to 1) traffic patterns, described earlier in §1.2.6, are used.

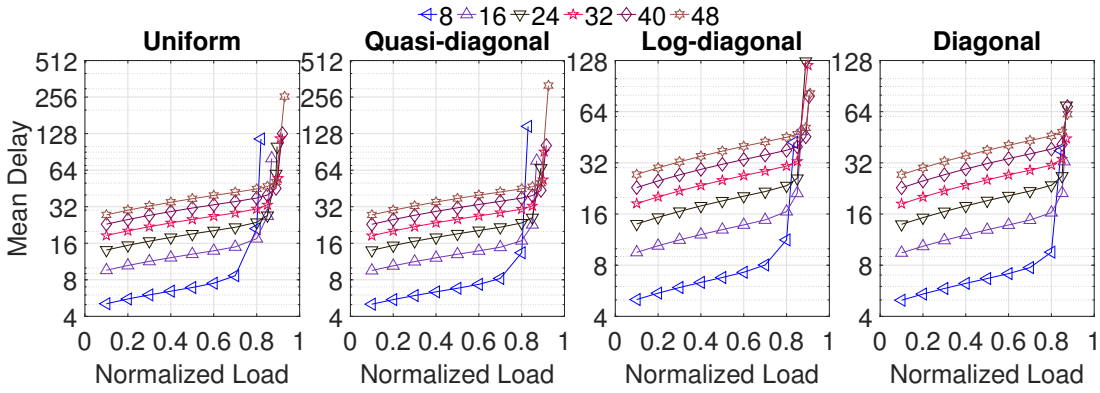


Figure 6.2: Mean delays of SB-QPS with different batch sizes under the 4 traffic patterns.

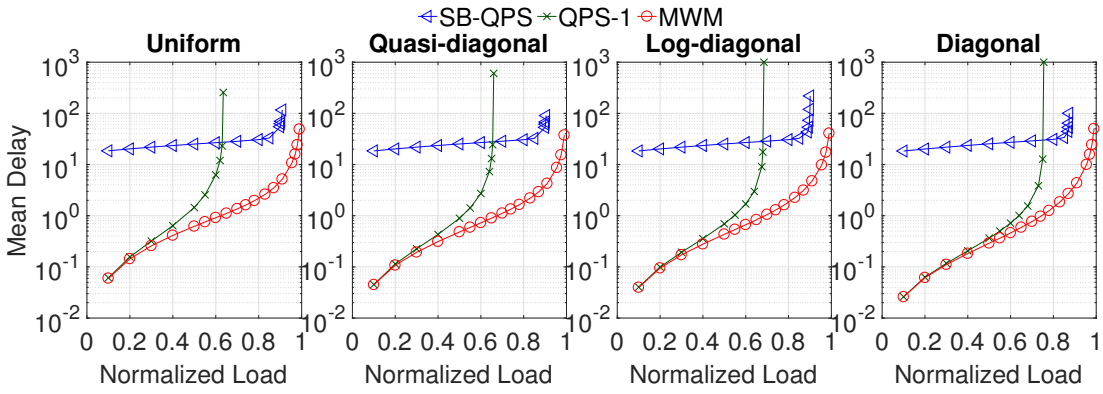
6.3.2 How Large Should Batch Size T Be?

When implementing SB-QPS, we have to first decide on the value of batch size T. As explained earlier in §1.3.4, for SB-QPS, a larger batch size T generally results in matchings of higher qualities and hence leads to better throughput performances. However, a larger T results in longer batching delays and hence can lead to worse overall delay performances for SB-QPS. In addition, since the availability information in a proposal message is T bits long, a larger T leads to a higher message complexity for SB-QPS.

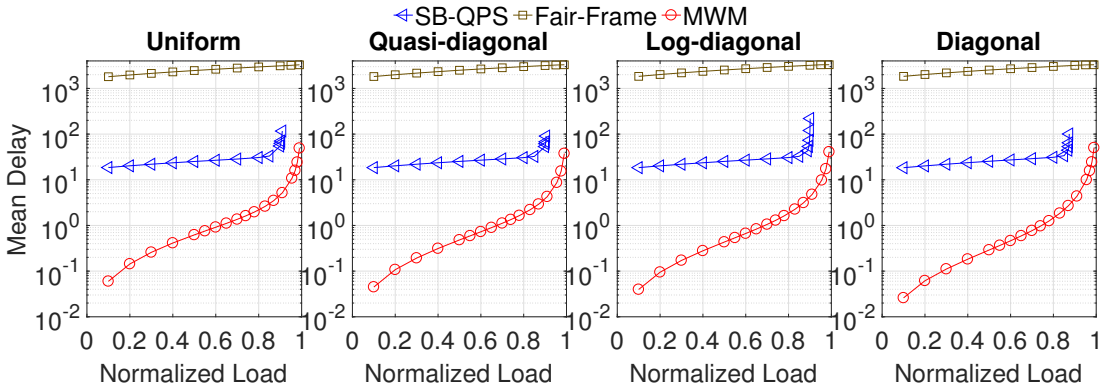
We use simulations to decide the value of T. In these simulations, we have investigated different values of T varying between 8 and 64. Here, to avoid too many curves in a single figure making it hard to read, we only present the simulation results with six different batch sizes: 8, 16, 24, 32, 40 and 48. Figure 6.2 presents the mean delays, more precisely, the average queueing delays in number of time slots, of SB-QPS with the six different batch sizes under the aforementioned four different traffic patterns. Each subfigure shows how the mean delay (on a *log scale* along the y-axis) varies with the offered load (along the x-axis). Figure 6.2 clearly shows that the larger the batch size T is, the better the throughput performance is. Figure 6.2 also shows that T=32 appears to be a nice performance-cost tradeoff: The throughput gain when increasing T beyond 32 (say to 40) are marginal. Hence, we set the batch size T to 32 in the rest of §6.3. SB-QPS clearly deserves its name (small-batch) since this tiny batch size of 32 is much smaller than that of most other batch scheduling algorithm.

6.3.3 Throughput and Delay Performances

Figure 6.3a presents the mean delays of SB-QPS, QPS-1, and MWM under the aforementioned 4 standard traffic patterns. We make three observations from Figure 6.3a. First, Figure 6.3a clearly shows that the mean delays of SB-QPS increase much slower with respect to the offered load than those of QPS-1 and MWM when the offered load is not very high (say < 0.8). Second, when the offered load is low to moderate (say < 0.6),



(a) Comparison against QPS-1 and MWM.



(b) Comparison against Fair-Frame and MWM.

Figure 6.3: Mean delays under *i.i.d.* Bernoulli traffic arrivals with the 4 traffic patterns.

because of the batching delays, the mean delays of SB-QPS are much higher than those of QPS-1 and MWM under all four traffic patterns. Third, SB-QPS significantly improves the maximum sustainable throughputs (where the delays start to "go through the roof" in each subfigure in Figure 6.3a) of QPS-1. More precisely, the maximum sustainable throughputs of SB-QPS are roughly $0.910, 0.905, 0.901$, and 0.874 under the uniform, quasi-diagonal, log-diagonal, and diagonal traffic patterns respectively; those of QPS-1 are only $0.634, 0.645, 0.681$, and 0.751. Hence, SB-QPS increases the throughputs by an additive term of around $0.276, 0.26, 0.22$ and 0.123 for the uniform, quasi-diagonal, log-diagonal, and diagonal traffic patterns respectively.

Figure 6.3b presents the mean delays of SB-QPS, Fair-Frame, and MWM under the 4 standard traffic patterns. We can see that, due to the much smaller batch size of SB-QPS (T =32) relative to that of Fair-Frame (T=3,279), SB-QPS outperforms Fair-Frame

significantly under all traffic patterns for all load factors except those close to the maximum sustainable throughputs of SB-QPS.

6.4 Conclusion

In this chapter, we propose a batch scheduling algorithm called SB-QPS that significantly reduces the batch size without sacrificing the throughput performance much, and achieves a time complexity of $O(1)$ per matching computation per port via parallelization. We show, through simulations, that the throughput performances of SB-QPS are much better than those of QPS-1, the state-of-the-art regular crossbar scheduling algorithm based on the same underlying bipartite matching algorithm.

Appendices

APPENDIX A

APPENDIX FOR CHAPTER 3

A.1 Parallelized Population

As explained in §3.1, the new random matching $A'(t)$ derived from the *arrival graph*, which is in general a partial matching, has to be populated into a full matching $R(t)$ before it can be merged with $S(t-1)$, the matching used in the previous time slot. SERENADE parallelizes this POPULATE procedure, *i.e.,* the round-robin pairing of unmatched input ports in $A'(t)$ with unmatched output ports in $A'(t)$, so that the time complexity for each input port is $O(\log N)$, as follows.

Suppose that each unmatched port (input port or output port) knows its own ranking, *i.e.,* the number of unmatched ports up to itself (including itself) from the first one (we will show later how each unmatched port can obtain its own ranking). Then, each unmatched input port needs to obtain the identity of the unmatched output port with the same ranking. This can be done via 3 message exchanges as follows. Each pair of unmatched input and output ports "exchange" their identities through a "broker". More precisely, the j^{th} unmatched input port (*i.e.,* unmatched input port with ranking j) first sends its identity to input port j (*i.e.,* the broker). Then, the j^{th} unmatched output port also sends its identity to input port j (*i.e.,* the broker). Finally, input port j (*i.e.,* the broker) sends the identity of the output port with ranking j to the input port (with ranking j). Thus, the input port learns the identity of the corresponding output port. Note that, since every pair of unmatched input port and output port has its unique ranking, thus they would have different "brokers". Therefore, all pairs can simultaneously exchange their messages without causing any congestion (*i.e.,* a port sending or receiving too many messages).

It remains to parallelize the computation of ranking each port (input port or output port).

This problem can be reduced to the parallel prefix sum problem [74] as follows. Here, we will only show how to compute the rankings of input ports in parallel; that for output ports is identical. Let $B[1..N]$ be a bitmap that indicates whether input port i is unmatched (when $B[i]=1$) or not (when $B[i]=0$). Note that, this bitmap is distributed, that is, each input port i only has a single bit $B[i]$. For $i=1,2,\cdots,N$, denote as r_i the ranking of input port i. It is clear that $r_i=\sum_{k=1}^{i} B[k]$, for any $1\leq i\leq N$. In other words, the N terms r_1, r_2, $\cdots$, r_N are the prefix sums of the N terms $B[1]$, $B[2]$, $\cdots$, $B[N]$. Using the Ladner-Fischer parallel prefix-sum algorithm [64], we can obtain these N prefix sums r_1, r_2, $\cdots$, r_N in $O(\log N)$ time (per port) using $2N$ processors (one at each input or output port).

A.2 Proof of Lemma 3.2.1

We need only to consider the following two cases.

(1) **The two walks are in the same "rotational" direction**. Without loss of generality, we assume the two walks are $i\rightsquigarrow\sigma^{\beta}(i)$ and $i\rightsquigarrow\sigma^{\gamma}(i)$ respectively, where $0\leq\beta<\gamma$, and $\sigma^{\beta}(i)=\sigma^{\gamma}(i)=j$. By applying the operator $\sigma^{-\beta}$ to both sides of the equation $\sigma^{\beta}(i)=\sigma^{\gamma}(i)$, we have $i=\sigma^{\gamma-\beta}(i)$. Hence, ℓ divides $(\gamma-\beta)$, where ℓ is the length of the cycle to which vertices i,j belong. Suppose $\kappa\ell=\gamma-\beta$, where $\kappa>0$ is an integer. Then, we have the $(\gamma-\beta)$-edge-long walk $\sigma^{\beta}(i)\rightsquigarrow\sigma^{\gamma}(i)$ coils around the cycle (of length ℓ) exactly κ times, and so the green weight $w_g\big(\sigma^{\beta}(i)\rightsquigarrow\sigma^{\gamma}(i)\big)$ (or red weight) is κ times of that of the cycle. Since vertex i can obtain the green weight (or the red weight) of the walk $\sigma^{\beta}(i)\rightsquigarrow\sigma^{\gamma}(i)$ via subtracting $w_g(i\rightsquigarrow\sigma^{\beta}(i))$ from $w_g(i\rightsquigarrow\sigma^{\gamma}(i))$, *i.e.,*

$$\begin{cases} w_g\big(\sigma^{\beta}(i)\rightsquigarrow\sigma^{\gamma}(i)\big) & \leftarrow w_g\big(i\rightsquigarrow\sigma^{\gamma}(i)\big)-w_g\big(i\rightsquigarrow\sigma^{\beta}(i)\big) \\ w_r\big(\sigma^{\beta}(i)\rightsquigarrow\sigma^{\gamma}(i)\big) & \leftarrow w_r\big(i\rightsquigarrow\sigma^{\gamma}(i)\big)-w_r\big(i\rightsquigarrow\sigma^{\beta}(i)\big) \end{cases} \tag{A.1}$$

it knows whether $w_g(i\rightsquigarrow\sigma^{\ell}(i))$ (the green weight of the cycle) or $w_r(i\rightsquigarrow\sigma^{\ell}(i))$ (the red weight of the cycle) is larger.

(2) **The two walks are in opposite directions**. Without loss of generality, we assume the two walks are $\sigma^{-\beta}(i) \rightsquigarrow i$ and $i \rightsquigarrow \sigma^{\gamma}(i)$ respectively where $\beta, \gamma > 0$, and $\sigma^{-\beta}(i) = \sigma^{\gamma}(i) = j$. By applying the operator σ^{β} to both sides of the equation $\sigma^{-\beta}(i) = \sigma^{\gamma}(i)$, we have $i = \sigma^{\gamma+\beta}(i)$. So ℓ divides $(\gamma+\beta)$, the rest reasoning is the same as before.

A.3 Proof of Lemma 3.3.1

Suppose that vertex i discovers vertex j after the k_1^{th} and k_2^{th} iteration respectively. Then we have $\sigma^{m_1}(i) = \sigma^{m_2}(i) = j$ where $m_1 = 2^{k_1}$ if i discovers j through Line 14 during the k_1^{th} iteration, otherwise $m_1 = -2^{k_1}$. Similarly, $m_2 = \pm 2^{k_2}$. Thus, we have $i = \sigma^{m_2 - m_1}(i)$. Therefore, there exists some positive integer κ such that $|m_2 - m_1| = \kappa\ell$, where ℓ is the length of the cycle.

For any other vertex x on the same cycle, we have $x = \sigma^{m_2 - m_1}(x)$. Thus, $\sigma^{m_2}(x) = \sigma^{m_1}(x) \triangleq y$. Therefore, x also discovers y twice.

A.4 Proof of Lemma 3.3.2

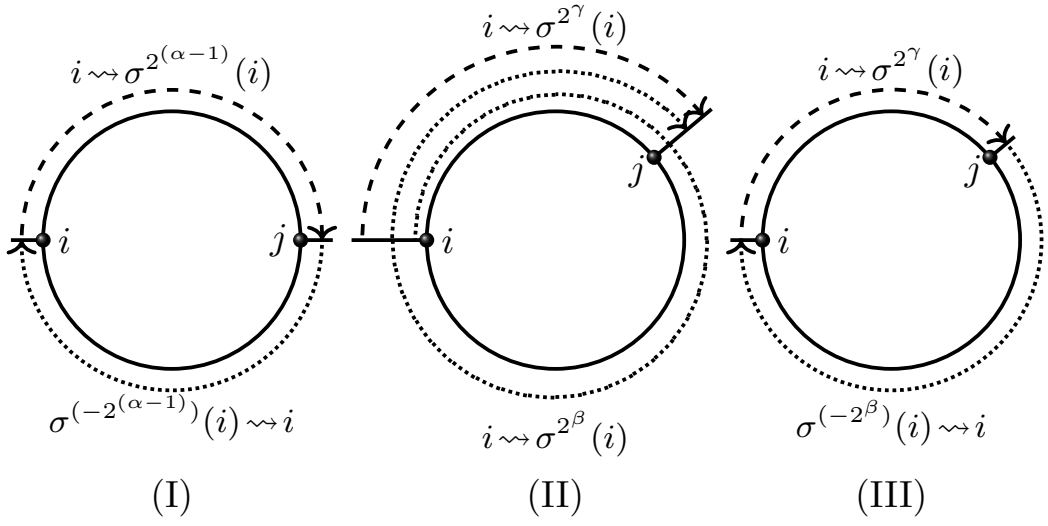


Figure A.1: Illustration of three cases corresponding to cycle lengths belonging to the three forms of the ouroboros numbers.

Here we only show the proof sketch for brevity. It is not hard to show that if the cycle,

to which vertex i belongs, has a length of an ouroboros number, *i.e.,* a divisor of numbers in the three forms defined in Definition 3.3.1, then vertex i will discover a vertex j on the same cycle twice via one of the three cases illustrated in Figure A.1. For example, if the cycle length ℓ is a divisor of form (I) defined in Definition 3.3.1, then it has to be a power of 2. Without loss of generality, we assume $\ell = 2^\alpha$, where nonnegative integer $\alpha \leq \log_2 N$. In the cases of $\alpha \geq 1$, it is clear that, in the $(\alpha-1)^{th}$ iteration, vertex i discovers $\sigma^{2^{\alpha-1}}(i)$ and $\sigma^{(-2^{\alpha-1})}(i)$, which turn out to be the same vertex. The case of $\alpha = 0$ is slightly different: After the 0^{th} iteration, vertex i discovers $\sigma(i)$ and $\sigma^{-1}(i)$, which are both equal to i. Therefore, (I) shown in Figure A.1 happens. Similarly, we can show that if the cycle length ℓ is a divisor of a number in form (II) (or (III)), then (II) (or (III)) shown in Figure A.1 happens.

Note that for (II) and (III) in Figure A.1, both the two walks can coil around the cycle for one or more times, and the directions of the two walks can be reversed. For example, for (III) in Figure A.1, the two walks could also be $\sigma^{-2^\gamma}(i) \rightsquigarrow i$ and $i \rightsquigarrow \sigma^{2^\beta}(i)$, and both of them could coil around the cycle for one or more times, *i.e.,* they are longer than the cycle.

A.5 Why Not Use More Than $1 + \log_2 N$ Iterations?

Fix a vertex i. Note that in the knowledge-discovery procedure, each iteration results in two new vertices being discovered by vertex i and hence increases the chance of a vertex being discovered twice by i. Hence, if we run more than $1+\log_2 N$ iterations, then vertex i may discover a vertex twice even if it is on a non-ouroboros cycle (as defined in Definition 3.3.1). In other words, with additional iterations, some non-ouroboros numbers may become "effective ouroboros numbers". Readers may wonder if we can do away with the distributed binary search simply by running a little more iterations (say $0.5 \log_2 N$ more iterations). Unfortunately, as shown in Table A.1, there exists some numbers (cycle lengths) that are "hardcore non-ouroboros" in the sense a vertex i on a cycle of such a length ℓ needs to run exactly $\lceil \ell/2 \rceil$ iterations to discover a vertex twice. In fact, it is a

long-standing open problem in mathematics whether there exists infinite number of what we call "hardcore non-ouroboros" numbers here. More precisely, it is a special case of the Artin's Conjecture [75], which, if put into our context, asks whether there are infinitely many prime numbers p such that, it takes a vertex i on a cycle of length p exactly $\lceil p/2 \rceil$ iterations to discover a vertex on the same cycle twice.

Table A.1: Examples of "hardcore non-ouroboros" numbers.

ℓ	61	131	239	509	1019
Iterations	31	66	120	255	510

A.6 SERENADE vs. MIX

In this section, we describe the three variants of MIX [54] in detail. The first variant, which is centralized and idealized, computes the total green and red weights of each cycle or path by "linearly" traversing the cycle or path. Hence it has a time complexity of $O(N)$, where N is the number of nodes in a wireless network. This idealized variant is however impractical because it requires the complete knowledge of the connectivity topology of the wireless network.

The second variant removes this infeasible requirement and hence is practical. It estimates and compares the *average* green and red weights of each cycle or path (equivalent to comparing the total green and red weights) using a synchronous iterative gossip algorithm proposed in [63]. In this gossip algorithm, each node (say X) is assigned a green (or red) weight that is equal to the weight of the edge that uses X as an endpoint and belongs to the matching used in the previous time slot (or in the new random matching); in each iteration, each node attempts to pair with a random neighbor and, if this attempt is successful, both nodes will be assigned the same red (or green) weight equal to the average of their current red (or green) weights. The time complexity of each MERGE is $O(l^2 N \log N)$, since this gossip algorithm requires $O(l^2 N \log N)$ iterations [54] for the average red (or green)

weight estimate to be close to the actual average with high probability. Here l is the length of the longest path or cycle.

The third (practical) variant, also a gossip-based algorithm, employs the aforementioned "idempotent trick" (see §A.7) to estimate and compare the *total* green and red weights of each cycle or path. This idempotent trick reduces the convergence time (towards the actual total weights) to $O(l)$ iterations, but as mentioned earlier requires each pair of neighbors to exchange a large number ($O(N \log N)$ to be exact) of exponential random variables during each message exchange. Since l is usually $O(N)$ in a random graph, the time complexity of this algorithm can be considered $O(N)$.

A.7 An Idempotent Trick

As mentioned in §A.6, there is an alternative solution to the consistency problem that does not require leader election, using a standard "idempotent trick" that was used in [54] to solve a similar problem. To motivate this trick, we zoom in on the example shown in Figure 3.2. Both the consistency problem and the absolute correctness problem above can be attributed to the fact that the (green or red) weights of some edges are accounted for (*i.e.*, added to the total) κ times, while those of others $\kappa - 1$ times. For example, in $3 \rightsquigarrow \sigma^{16}(3)$, the (green or red) weights of edges $(3, 4)$, $(4, 14)$, and *etc* are accounted for 2 times, while those of edges $(2, 8)$, $(8, 5)$, and *etc* 1 times. Since the "+" operator is not idempotent (so adding a number to a counter κ times is not the same as adding it $\kappa - 1$ times), the total (green or red) weight of the walk obtained this way does not perfectly track that of the cycle.

The "idempotent trick" is to use, instead of the "+" operator, a different and idempotent operator MIN to arrive at an estimation of the total green (or red) weight; the MIN is idempotent in the sense the minimum of a multi-set (of real numbers) M is the same as that of the set of distinct values in M. The idempotent trick works, in this O-SERENADE context, for a set of edges e_1, e_2, ..., e_Z that comprise a non-ouroboros cycle with green weights

w_1, w_2, ..., w_Z respectively, as follows; the trick works in the same way for the red weights. Each edge e_ζ "modulates" its green weight w_ζ onto an exponential random variable X_ζ with distribution $F(x) = 1 - e^{-x/w_\zeta}$ (for $x > 0$) so that $E[X_\zeta] = w_\zeta$. Then the green weight of every walk W on this non-ouroboros cycle can be encoded as $MIN\{X_\zeta | e_\zeta \in W\}$. It is not hard to show that we can compute this MIN encoding of every 2^d-edge-long walk $i \rightsquigarrow \sigma^{2^d}(i)$ by O-SERENADE in the same inductive way we compute the "+" encoding. For example, under the MIN encoding, Line 10 in Algorithm 1 becomes "Send to i_U the value $MIN\{X_\zeta | e_\zeta \in i \rightsquigarrow \sigma^{2^{k-1}}(i)\}$". However, unlike the "+" encoding, which requires the inclusion of only 1 "codeword" in each message, the MIN encoding requires the inclusion of $O(N \log N)$ *i.i.d.* "codewords" in each message in order to ensure sufficient estimation accuracy [54].

A.8 More Performance Evaluations

Table A.2: Average per-port message complexities of SERENADE (bytes).

Traffic patterns	Uniform			Quasi-diagonal			Log-diagonal			Diagonal		
N	64	128	256	64	128	256	64	128	256	64	128	256
light ($\rho = 0.1$)	29.79	40.04	51.29	25.86	36.2	47.7	12.47	14.32	16.23	8.14	9.03	9.98
moderate ($\rho = 0.6$)	34.79	44.84	55.73	31.08	40.55	50.82	21.61	25.69	29.86	14.51	16.82	19.37
high ($\rho = 0.95$)	35.21	45.26	56.13	28.65	37.19	46.50	20.07	23.66	27.47	15.70	18.38	21.50

A.8.1 Message Complexities

In this section, we investigate the empirical message complexities of SERENADE. Those of O-SERENADE are not presented here, as they are similar.

Per-Port Message Complexities. Table A.2 shows the numerical results of the average per-port message complexities (in bytes) of SERENADE for $N = 64$, 128, and 256 for the 4 traffic patterns described above under low, moderate, and high offered loads. As explained in §3.3.3, it suffices to only include $w_r(\cdot) - w_g(\cdot)$, the difference between the red and green weights, in each message. This difference can be encoded in 15 bits (with a single "sign"

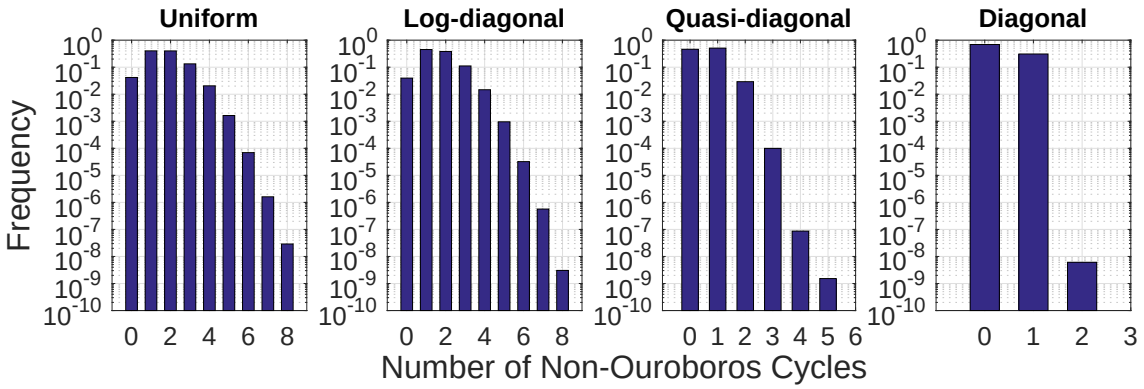


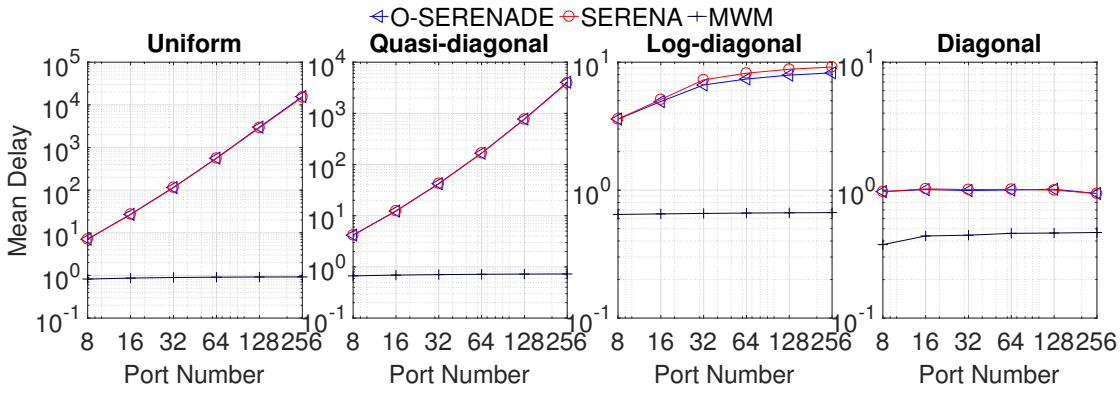
Figure A.2: Histogram for number of non-ouroboros cycles in SERENADE ($N = 256$, $\rho = 0.6$).

bit), as we assume that each weight can fit in 14 bits (*i.e.,* no more than 16,384 packets). The worst-cast message complexities of SERENADE, described in §3.3.3 and §3.5.2, are $44.25, 53.25, 63$ bytes per port for $N = 64$, 128, and 256 respectively. Comparing them against those values in Table A.2, we can see that the average message complexities (per port), under all load factors or traffic patterns, are lower than the worst cases.

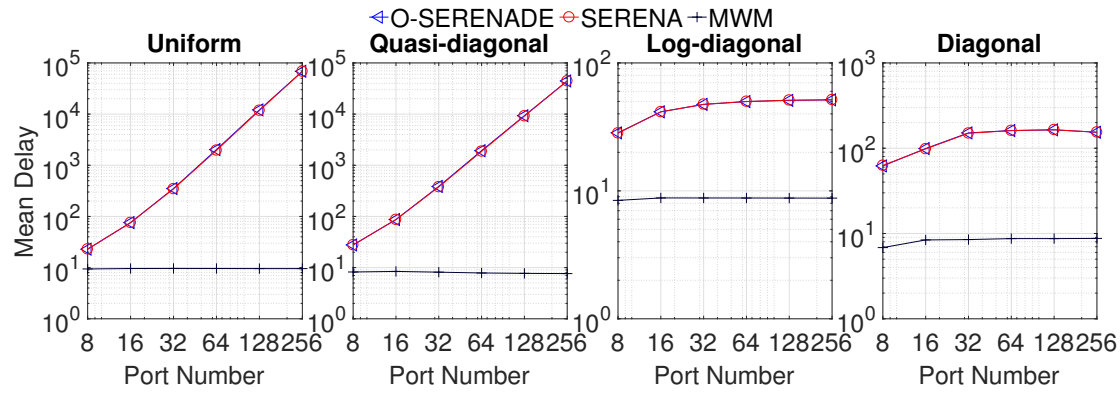
Number of Non-Ouroboros Cycles. We also measure the number of non-ouroboros cycles in each time slot in SERENADE, of which the histograms (on a *log scale* along the y-axis) for $N = 256$, and an offered load of $\rho = 0.6$ under the four different traffic patterns are shown in Figure A.2. The average numbers of non-ouroboros cycles are $1.69, 1.61, 0.57, 0.31$ under the uniform, quasi-diagonal, log-diagonal, and diagonal traffic patterns respectively. It is not hard to check that under any of the four different traffic patterns, in more than 99% of instances, there are no more than 4 non-ouroboros cycles per time slot.

A.8.2 How Mean Delay Scales with N

In this section, we investigate how the mean delay of O-SERENADE scales with the number of input/output ports N under *i.i.d.* Bernoulli traffic. We have simulated the following different N values: $N = 8, 16, 32, 64, 128, 256$. Figure A.3 compares the mean delays for O-SERENADE against those for SERENA under the 4 different traffic patterns with a moderate offered load of 0.6 and a heavy offered load of 0.95. As a benchmark, we also show



(a) Offered load: 0.6



(b) Offered load: 0.95

Figure A.3: Mean delays scaling with number of ports N for O-SERENADE, SERENA, and MWM under the 4 traffic patterns.

those of MWM. It shows that the scaling behaviors of O-SERENADE are almost the same as that of SERENA in terms of mean delays, for all values of N. More precisely, under the log-diagonal and the diagonal traffic patterns, O-SERENADE achieves near-optimal scaling (*i.e.,* nearly constant independent of N) of mean delay, whereas under the uniform and quasi-diagonal traffic patterns, the mean delay grows roughly quadratically with N (*i.e.*, $O(N^2)$ scaling).

APPENDIX B

APPENDIX FOR CHAPTER 4

B.1 QPS Variants

The success we have with QPS leads us to wonder if we can obtain better switching performance by using other proportional sampling strategies. For example, instead of setting sampling probabilities proportional to the lengths of the VOQs, we may set them proportional to the squares of the lengths of the VOQs. More generally, we can set the sampling probabilities proportional to any arbitrary function $f(\cdot)$ of the lengths of the VOQs. We refer to this family of strategies as FQPS, where QPS is a special case (using a linear weight function $f(x) = x$).

To some readers, FQPS may sound similar to MWM-α [29]. They are, however, fundamentally different. The MWM-α work studies the performances of MWM when the weight of a VOQ queue of length x is set to x^α; it does not care how a MWM is computed. FQPS, on the other hand, is about how to better generate a starter matching that can result in a final matching that is as close to the MWM as possible, after a reasonable amount of further computation (*e.g.,* $O(N)$).

We have evaluated the performance of several FQPS variants through simulations. Simulation results (see §B.5.1) show that the delay performance of QPS can be slightly improved with certain nonlinear weight functions (*e.g.,* with $f(x) = x^2$). However, whereas the time complexity of QPS is $O(1)$ per packet, other FQPS variants all have a higher time complexity of $O(\log N)$ per packet. Hence we conclude that QPS overall remains the best practical solution.

B.2 Space Complexity of QPS

Each node (packet) in the main data structure contains 3 pointers (2 pointers encoded as "$\langle letter \rangle$" plus 1 for the linked list) and the index of the VOQ (the value j in every node in Figure 4.1), which needs $\log_2 N$ bits to store (typically less than 2 bytes). Each array entry (packet) in the auxiliary data structure is also a pointer. Note that, in the main data structure, we need an array entry (record) for each VOQ, not for each packet; since the maximum number of packets at an input port is typically much larger than N, the number of VOQs, the memory overhead of these array entries (record), is no more than 2 bytes per packet. Therefore, the memory overhead of the data structures is no more than 4 pointers (4 bytes each) plus 4 bytes, or 20 bytes, per packet.

B.3 Proof of Theorem 4.3.2

We have explained in §4.3 that, given any crossbar scheduling algorithm $\pi \in \widetilde{\Pi}$, its joint queueing and scheduling process $\left\{ \big(Q(t), S(t)\big) \right\}_{t=0}^{\infty}$, under any *i.i.d.* arrival processes $A(t)$ (not necessarily admissible), is a Markov chain. We claim this Markov chain is irreducible and aperiodic, when π is furthermore (ϵ, δ)-***MWM*** and $A(t)$ is furthermore admissible. Here we provide only a sketchy justification. To justify the irreducibility, we show that $Q(t)$, starting from any state (*i.e.,* queue lengths) it is currently in, will with a nonzero probability return to the "all-queues-empty" state in a finite number of time slots. To show this property, we claim that, for any integer $\tau > 0$, the switch could, with a nonzero probability, have no packet arrivals to any of its VOQs during $[t, t+\tau]$. This claim is true because, the arrival process $A(t)$ is *i.i.d.*, and for any $1 \le i \le N$ and $1 \le j \le N$, we have $\beta_{ij} \triangleq P[a_{ij}(t) = 0] > 0$ (otherwise the process $a_{ij}(t)$ is not admissible). Hence, when there are no packet arrivals during $[t, t+\tau]$, which happens with a nonzero probability, a "reasonably good" crossbar scheduling algorithm (being *non-degenerative* and (ϵ, δ)-***MWM***) can clear all the queues during $[t, t+\tau]$, with a sufficiently large τ, and return the

$Q(t)$ part of the Markov chain to the "all-queues-empty" state. As to the $S(t)$ part of the Markov chain, the algorithm resets (*i.e.,* returns) $S(t)$ to the default random schedule $R(t)$ when all queues are empty, as explained earlier. Therefore, the Markov chain is irreducible. To justify the aperiodicity of the Markov chain, we note that there is a nonzero probability for the Markov chain to stay at "all-queues-empty" for at least two consecutive time slots.

Now that the Markov chain is irreducible and aperiodic, to prove Theorem 4.3.2, it remains to show that (1) the Markov chain is positive recurrent and hence converges to a stationary distribution, and (2) the stationary distribution has a finite first moment. We accomplish both by analyzing the following Lyapunov function $V(\cdot)$ of $Y(t) = \big(Q(t), S(t)\big)$:

$$V(Y) = V_1(Y) + V_2(Y) \tag{B.1}$$

where $V_1(Y) = \|Q\|_2^2$, $V_2(Y) = \big([\langle \rho^* S_Q - S, Q\rangle]^+\big)^2$. Here, $\|\cdot\|_2$ is the 2-norm, S_Q is a schedule/matching achieving maximum weight *w.r.t.* Q, and $\rho^* = \frac{1}{2}(1+\rho)$, where $\rho < 1$ is the maximum normalized load imposed on any input or output port as defined in (1.1). It is clear that $\rho^* < 1$.

Note that, in [11], $V_1(Y)$ is defined in the same way as in this work, whereas $V_2(Y)$ is defined as $V_2(Y) \triangleq \big(\langle S_Q - S, Q\rangle\big)^2$, which is quite different than in this work. We must define $V_2(Y)$ differently here because if its definition in [11] were used instead, there would be an additional positive drift term $c_4 V_1(Y(t))$ on the RHS of (B.4) (in Lemma B.3.3) which is asymptotically larger than the negative drift term $-\epsilon_1\sqrt{V_1(Y(t))}$ on the RHS of (B.3) (in Lemma B.3.2), resulting in an overall positive drift on the RHS of (B.2) when Lemma B.3.2 and Lemma B.3.3 are combined to prove Lemma B.3.1.

The proof of Theorem 4.3.2 relies on the following drift condition of $V(Y)$.

Lemma B.3.1. If the arrivals are admissible *i.i.d.*, then there exists $B, \epsilon > 0$ such that, if $V\big(Y(t)\big) > B$, we have,

$$\mathbb{E}\big[V\big(Y(t+1)\big) - V\big(Y(t)\big) \mid Y(t)\big] < -\epsilon\|Q(t)\|_2 \tag{B.2}$$

The proof of Lemma B.3.1 in turn relies on the following two lemmas.

Lemma B.3.2. If the arrivals are admissible *i.i.d.*, then the drift of the function V_1 satisfies the following inequality

$$\mathbb{E}\big[V_1\big(Y(t+1)\big) - V_1\big(Y(t)\big) \mid Y(t)\big] \leq -\epsilon_1\sqrt{V_1\big(Y(t)\big)} + 2\sqrt{V_2\big(Y(t)\big)} + c_1 \tag{B.3}$$

Here, $\epsilon_1 = \frac{1-\rho}{N}$, $c_1 = \mathbb{E}\big[\|A(t)+\mathbf{1}\|_2^2\big]$ and $\mathbf{1}$ is the vector with all its elements equal to 1.

Lemma B.3.3. If the arrivals are admissible *i.i.d.*, then the drift of the function V_2 satisfies the following inequality

$$\mathbb{E}\big[V_2\big(Y(t+1)\big) - V_2\big(Y(t)\big) \mid Y(t)\big] \leq -\epsilon_2 V_2\big(Y(t)\big) + c_2\sqrt{V_2\big(Y(t)\big)} + c_3 \tag{B.4}$$

Here, $\epsilon_2 > 0$ is a constant, $c_2 = 4(\rho+2)N$, $c_3 = 4\mathbb{E}\big[\big(\langle\mathbf{1}, A(t)\rangle + 2N\big)^2\big]$.

B.3.1 Proof of Lemma B.3.2

By simple calculations and using (4.1), we have

$$\begin{aligned}
&\mathbb{E}\big[V_1\big(Y(t+1)\big) - V_1\big(Y(t)\big) \mid Y(t)\big] \\
=&\,\mathbb{E}\big[\|Q(t+1)\|_2^2 - \|Q(t)\|_2^2 \mid Y(t)\big] \\
\leq&\,\mathbb{E}\big[\langle A(t)-S(t), 2Q(t)\rangle \mid Y(t)\big] + \mathbb{E}\big[\|A(t)-S(t)\|_2^2 \mid Y(t)\big]
\end{aligned} \tag{B.5}$$

Here, we use the fact that $\|Q(t+1)\|_2^2 = \|[Q(t)+A(t)-S(t)]^+\|_2^2 \leq \|Q(t)+A(t)-S(t)\|_2^2$.

Focusing on the first term $\mathbb{E}\big[\langle A(t)-S(t), 2Q(t)\rangle \mid Y(t)\big]$ above, we have

$$\begin{aligned}
&\mathbb{E}\big[\langle A(t)-S(t), 2Q(t)\rangle \mid Y(t)\big] \\
=&\,\langle \Lambda - S(t), 2Q(t)\rangle \\
=&\,2\langle \Lambda - \rho^* S_{Q(t)}, Q(t)\rangle + 2\langle \rho^* S_{Q(t)} - S(t), Q(t)\rangle
\end{aligned} \tag{B.6}$$

According to Fact 1.2.1 (see (1.2)), we can decompose Λ as follows: $\Lambda = \sum_{n=1}^{K} \alpha_n M_n$, where $K \leq N^2 - 2N + 2$, $\alpha_n > 0$ for $n = 1, 2, ..., K$, and $\sum_{n=1}^{K} \alpha_n \leq \rho$.

Hence, we have

$$
\begin{aligned}
&\langle \Lambda - \rho^* S_{Q(t)}, Q(t) \rangle \\
=& \langle \sum_{n=1}^{K} \alpha_n M_n - \rho^* S_{Q(t)}, Q(t) \rangle \\
=& \langle \sum_{n=1}^{K} \alpha_n M_n - \rho^* S_{Q(t)}, Q(t) \rangle - \sum_{n=1}^{K} \alpha_n W_{Q(t)} + \sum_{n=1}^{K} \alpha_n W_{Q(t)} \\
=& \sum_{n=1}^{K} \alpha_n \big(\langle M_n, Q(t) \rangle - W_{Q(t)} \big) + (\sum_{n=1}^{K} \alpha_n - \rho^*) W_{Q(t)} \\
\leq& (\sum_{n=1}^{K} \alpha_n - \rho^*) W_{Q(t)} \qquad \text{(B.7)} \\
\leq& \big(\rho - \frac{1}{2}(1+\rho) \big) W_{Q(t)} \qquad \text{(B.8)} \\
\leq& - \frac{(1-\rho) W_{Q(t)}}{2} \qquad \text{(B.9)}
\end{aligned}
$$

Inequality (B.7) holds because $\forall 1 \leq n \leq K$ we have $\alpha_n > 0$ and $\langle M_n, Q(t) \rangle - W_{Q(t)} \leq 0$ (the weight of M_n is no more than $W_{Q(t)}$, the weight of the MWM *w.r.t.* $Q(t)$) and (B.8) is due to $\sum_{n=1}^{K} \alpha_n \leq \rho$.

Since,

$$
\begin{aligned}
W_{Q(t)} &\geq \max_{n=1,\cdots,N^2} q_n(t) \\
&\geq \sqrt{\frac{\|Q(t)\|_2^2}{N^2}} \\
&= \frac{1}{N} \sqrt{V_1\big(Y(t)\big)} \qquad \text{(B.10)}
\end{aligned}
$$

From (B.5), (B.6), (B.9) and (B.10), we have

$$
\begin{aligned}
&\mathbb{E}\big[V_1\big(Y(t+1)\big) - V_1\big(Y(t)\big) \mid Y(t) \big] \\
\leq& -(1-\rho) \frac{1}{N} \sqrt{V_1\big(Y(t)\big)} + 2 \langle \rho^* S_{Q(t)} - S(t), Q(t) \rangle + \mathbb{E}\big[\|A(t) - S(t)\|_2^2 \mid Y(t) \big] \\
\leq& -\epsilon_1 \sqrt{V_1\big(Y(t)\big)} + 2 \sqrt{V_2\big(Y(t)\big)} + c_1 \qquad \text{(B.11)}
\end{aligned}
$$

Here $\epsilon_1 = \frac{1-\rho}{N}$, $c_1 = \mathbb{E}\big[\|A(t) + \mathbf{1}\|_2^2 \big]$ and $\mathbf{1}$ is the vector with all its elements equal to 1.

B.3.2 Proof of Lemma B.3.3

By simple calculations, we have

$$\mathbb{E}[V_2\big(Y(t+1)\big) \mid Y(t)] = \mathbb{P}[\mathcal{E}] \cdot 0 + \mathbb{P}[\mathcal{E}^c] \cdot \mathbb{E}[V_2\big(Y(t+1)\big) \mid Y(t), \mathcal{E}^c] \tag{B.12}$$

Here, $\mathcal{E}$ is the event $\big\{\langle \rho^* S_{Q(t+1)} - S(t+1), Q(t+1)\rangle \leq 0\big\}$, and $\mathcal{E}^c$ is the complementary event of $\mathcal{E}$.

Since algorithm π is (ϵ, δ)-***MWM*** (see Definition 4.3.3), for $\epsilon_3 = 1 - \rho^* > 0$, there exists $\delta > 0$, such that,

$$\begin{aligned}
\mathbb{P}[\mathcal{E}^c] &= 1 - \mathbb{P}\big[\langle \rho^* S_{Q(t+1)} - S(t+1), Q(t+1)\rangle \leq 0\big] \\
&= 1 - \mathbb{P}\big[\rho^* W_{Q(t+1)} - W(t+1) \leq 0\big] \\
&= 1 - \mathbb{P}\big[W(t+1) \geq \big(1 - (1-\rho^*)\big) W_{Q(t+1)}\big] \\
&= 1 - \mathbb{P}\big[W(t+1) \geq \big(1 - \epsilon_3\big) W_{Q(t+1)}\big] \\
&\leq 1 - \delta
\end{aligned} \tag{B.13}$$

Focusing on the second term in the RHS of (B.12), we have

$$\begin{aligned}
&\mathbb{E}\big[V_2\big(Y(t+1)\big) \mid Y(t), \mathcal{E}^c\big] \\
=&\mathbb{E}\Big[\big([\langle \rho^* S_{Q(t+1)} - S(t+1), Q(t+1)\rangle]^+\big)^2 \mid Y(t), \mathcal{E}^c\Big] \\
=&\mathbb{E}\big[\big(\langle \rho^* S_{Q(t+1)} - S(t+1), Q(t+1)\rangle\big)^2 \mid Y(t), \mathcal{E}^c\big] \\
\leq&\mathbb{E}\big[\big(\langle \rho^* S_{Q(t+1)} - S(t+1), Q(t) + A(t) - S(t)\rangle + N\big)^2 \mid Y(t), \mathcal{E}^c\big] \qquad \text{(B.14)} \\
=&\mathbb{E}\big[\big(\langle \rho^* S_{Q(t+1)}, Q(t)\rangle - \langle S(t+1), Q(t)\rangle + N \\
&\qquad + \langle \rho^* S_{Q(t+1)} - S(t+1), A(t) - S(t)\rangle\big)^2 \mid Y(t), \mathcal{E}^c\big] \qquad \text{(B.15)}
\end{aligned}$$

Here, the term N in (B.14) is because

$$
\begin{aligned}
&\langle \rho^* S_{Q(t+1)} - S(t+1), Q(t+1) \rangle \\
=& \langle \rho^* S_{Q(t+1)} - S(t+1), [Q(t) + A(t) - S(t)]^+ \rangle \\
=& \langle \rho^* S_{Q(t+1)} - S(t+1), Q(t) + A(t) - S(t) \rangle + \langle \rho^* S_{Q(t+1)} - S(t+1), \chi_{\left\{Q(t)+A(t)-S(t)<0\right\}} \rangle \\
\leq& \langle \rho^* S_{Q(t+1)} - S(t+1), Q(t) + A(t) - S(t) \rangle + \langle S_{Q(t+1)}, \chi_{\left\{Q(t)+A(t)-S(t)<0\right\}} \rangle \\
\leq& \langle \rho^* S_{Q(t+1)} - S(t+1), Q(t) + A(t) - S(t) \rangle + N
\end{aligned}
$$

Here, $\chi_{\left\{Q(t)+A(t)-S(t)<0\right\}}$ is a vector whose n^{th} element/scalar takes value 1 if $q_n(t) + a_n(t) - s_n(t) < 0$, which happens only when $q_n(t) = 0, a_n(t) = 0, s_n(t) = 1$ and value 0 otherwise. The last inequality is because $\langle S_{Q(t+1)}, \mathbf{1} \rangle \leq N$, where $\mathbf{1}$ is the vector with all its elements equal to 1. In the following proof steps, we will use similar tricks to remove $[\,\cdot\,]^+$, which we may not elaborate again.

We now derive the following three inequalities that will be needed to complete our proof.

First, we have

$$
\begin{aligned}
&\langle S(t+1), Q(t) \rangle \\
\geq& \langle S(t+1), Q(t+1) - A(t) + S(t) \rangle - N && \text{(B.16)} \\
\geq& \langle S(t), Q(t+1) \rangle - \langle S(t+1), A(t) - S(t) \rangle - N && \text{(B.17)} \\
=& \langle S(t), Q(t) + A(t) - S(t) \rangle + \langle S(t), \chi_{\left\{Q(t)+A(t)-S(t)<0\right\}} \rangle - \langle S(t+1), A(t) - S(t) \rangle - N \\
\geq& \langle S(t), Q(t) + A(t) - S(t) \rangle - \langle S(t+1), A(t) - S(t) \rangle - N \\
=& \langle S(t), Q(t) \rangle - \langle S(t+1) - S(t), A(t) - S(t) \rangle - N \\
=& \langle S(t), Q(t) \rangle - \langle S(t+1) - S(t), A(t) \rangle + \langle S(t+1) - S(t), S(t) \rangle - N \\
\geq& \langle S(t), Q(t) \rangle - \langle \mathbf{1}, A(t) \rangle - 2N && \text{(B.18)}
\end{aligned}
$$

Here, the constant term N in (B.16) is due to the removal of $[\,\cdot\,]^+$, and (B.17) is due to the fact that π is *non-degenerative*, *i.e.,* $\langle S(t+1), Q(t+1) \rangle \geq \langle S(t), Q(t+1) \rangle$. The derivation of (B.18) uses the following two simple facts: $0 \leq \langle S(t+1), S(t) \rangle \leq N$ and

$0 \leq \langle S(t), S(t) \rangle \leq N.$

Second, we have

$$\langle S_{Q(t+1)}, Q(t) \rangle \leq W_{Q(t)} = \langle S_{Q(t)}, Q(t) \rangle \tag{B.19}$$

Third, we have

$$\begin{aligned}
&\langle \rho^* S_{Q(t+1)} - S(t+1), A(t) - S(t) \rangle \\
=& \langle \rho^* S_{Q(t+1)} - S(t+1), A(t) \rangle - \langle \rho^* S_{Q(t+1)} - S(t+1), S(t) \rangle \\
\leq& \langle S_{Q(t+1)}, A(t) \rangle + \langle S(t+1), S(t) \rangle \\
\leq& \langle \mathbf{1}, A(t) \rangle + N
\end{aligned} \tag{B.20}$$

Now, according to (B.18), (B.19) and (B.20), we have, conditioned upon the event $\mathcal{E}^c$,

$$\begin{aligned}
0 <& \langle \rho^* S_{Q(t+1)} - S(t+1), Q(t+1) \rangle \\
\leq& \langle \rho^* S_{Q(t+1)}, Q(t) \rangle - \langle S(t+1), Q(t) \rangle + N + \langle \rho^* S_{Q(t+1)} - S(t+1), A(t) - S(t) \rangle \\
\leq& \langle \rho^* S_{Q(t)}, Q(t) \rangle - \big(\langle S(t), Q(t) \rangle - \langle \mathbf{1}, A(t) \rangle - 2N \big) + N + \big(\langle \mathbf{1}, A(t) \rangle + N \big) \\
\leq& \langle \rho^* S_{Q(t)} - S(t), Q(t) \rangle + 2 \langle \mathbf{1}, A(t) \rangle + 4N \\
\leq& \sqrt{V_2\big(Y(t)\big)} + 2\big(\langle \mathbf{1}, A(t) \rangle + 2N \big)
\end{aligned} \tag{B.21}$$

Therefore, we have

$$\begin{aligned}
&\mathbb{E}\big[V_2\big(Y(t+1)\big) \mid Y(t), \mathcal{E}^c \big] \\
\leq& V_2\big(Y(t)\big) + 4\big(\mathbb{E}\big[\langle \mathbf{1}, A(t) \rangle \mid \mathcal{E}^c \big] + 2N \big) \sqrt{V_2\big(Y(t)\big)} + 4\mathbb{E}\big[\big(\langle \mathbf{1}, A(t) \rangle + 2N \big)^2 \mid \mathcal{E}^c \big]
\end{aligned} \tag{B.22}$$

Since $\langle \mathbf{1}, A(t) \rangle \geq 0$, we have,

$$\begin{aligned}
&\mathbb{E}\big[\langle \mathbf{1}, A(t) \rangle \mid \mathcal{E}^c \big] \mathbb{P}[\mathcal{E}^c] \\
\leq& \mathbb{E}\big[\langle \mathbf{1}, A(t) \rangle \mid \mathcal{E}^c \big] \mathbb{P}[\mathcal{E}^c] + \mathbb{E}\big[\langle \mathbf{1}, A(t) \rangle \mid \mathcal{E} \big] \mathbb{P}[\mathcal{E}] \\
=& \mathbb{E}\big[\langle \mathbf{1}, A(t) \rangle \big] \\
\leq& \rho N
\end{aligned} \tag{B.23}$$

Here, $\rho < 1$ is the maximum normalized load imposed on any input or output port as defined in (1.1).

Similarly, we have

$$\mathbb{E}\big[\big(\langle \mathbf{1}, A(t)\rangle + 2N\big)^2 \mid \mathcal{E}^c\big]\mathbb{P}[\mathcal{E}^c] \le \mathbb{E}\big[\big(\langle \mathbf{1}, A(t)\rangle + 2N\big)^2\big] \tag{B.24}$$

Substituting (B.13), (B.22), (B.23) and (B.24) into (B.12), we have

$$\mathbb{E}[V_2\big(Y(t+1)\big) \mid Y(t)] \le (1-\delta)V_2\big(Y(t)\big) + c_2\sqrt{V_2\big(Y(t)\big)} + c_3 \tag{B.25}$$

where $c_2 = 4(\rho+2)N$, $c_3 = 4\mathbb{E}\big[\big(\langle \mathbf{1}, A(t)\rangle + 2N\big)^2\big]$.

Therefore, we have

$$\mathbb{E}\big[V_2\big(Y(t+1)\big) - V_2\big(Y(t)\big) \mid Y(t)\big] \le -\delta V_2\big(Y(t)\big) + c_2\sqrt{V_2\big(Y(t)\big)} + c_3 \tag{B.26}$$

Hence, Lemma B.3.3 holds with $\epsilon_2 = \delta$.

B.3.3 Proof of Lemma B.3.1

We now proceed to prove Lemma B.3.1. Note that, the proof is the same as the proof of *Lemma 1* in [11]. We reproduce it with some minor revisions for this thesis to be self-contained.

By Lemma B.3.2 (concerning the drift of $V_1(Y)$) and Lemma B.3.3 (concerning the drift of $V_2(Y)$), the drift of $V(Y)$ satisfies

$$\begin{aligned}&\mathbb{E}\big[V\big(Y(t+1)\big) - V\big(Y(t)\big) \mid Y(t)\big]\\ \le& -\epsilon_1\sqrt{V_1\big(Y(t)\big)} + (2+c_2)\sqrt{V_2\big(Y(t)\big)} - \epsilon_2 V_2\big(Y(t)\big) + c_1 + c_3\end{aligned} \tag{B.27}$$

When $V\big(Y(t)\big) \ge B$, we have $V_1\big(Y(t)\big) \ge B - V_2\big(Y(t)\big)$, and hence

$$-\epsilon_1\sqrt{V_1\big(Y(t)\big)} \le -\frac{\epsilon_1}{2}\sqrt{V_1\big(Y(t)\big)} - \frac{\epsilon_1}{2}\sqrt{B - V_2\big(Y(t)\big)} \tag{B.28}$$

Substituting the first term in the RHS of (B.27) by the RHS of (B.28), we obtain

$$\begin{aligned}&\mathbb{E}\big[V\big(Y(t+1)\big) - V\big(Y(t)\big) \mid Y(t)\big]\\ \le& -\frac{\epsilon_1}{2}\sqrt{V_1\big(Y(t)\big)} - \frac{\epsilon_1}{2}\sqrt{B - V_2\big(Y(t)\big)} + (2+c_2)\sqrt{V_2\big(Y(t)\big)} - \epsilon_2 V_2\big(Y(t)\big) + c_1 + c_3\end{aligned}$$

It is clear that when B is large enough, we have,

$$-\frac{\epsilon_1}{2}\sqrt{B - V_2\big(Y(t)\big)} + (2+c_2)\sqrt{V_2\big(Y(t)\big)} - \epsilon_2 V_2\big(Y(t)\big) + c_1 + c_3 < 0$$

Hence,

$$\begin{aligned}
&\mathbb{E}\big[V\big(Y(t+1)\big) - V\big(Y(t)\big) \mid Y(t)\big] \\
&< -\frac{\epsilon_1}{2}\sqrt{V_1\big(Y(t)\big)} \\
&= -\frac{\epsilon_1}{2}\|Q(t)\|_2
\end{aligned} \tag{B.29}$$

Hence Lemma B.3.1 holds with $\epsilon = \frac{\epsilon_1}{2}$. Here $\epsilon_1 = \frac{1-\rho}{N}$ as specified in Lemma B.3.2 (see (B.3)).

B.3.4 Proof of Theorem 4.3.2

To prove Theorem 4.3.2, we need a theorem due to Tweedie [76], stated as follows.

Theorem B.3.1 (Tweedie [76])**.** Suppose that $\{Y_n\}_{n=0}^{\infty}$ is an aperiodic and irreducible Markov chain with countable state space $\mathcal{Y}$. Let $f(Y), g(Y)$ be real nonnegative functions such that $g(Y) \geq f(Y), Y \in D^c$, where D is a finite subset of $\mathcal{Y}$. If

$$\mathbb{E}\big[g(Y_1) \mid Y_0 = Y\big] < \infty, \quad Y \in D \tag{B.30}$$

and

$$\mathbb{E}\big[g(Y_1) \mid Y_0 = Y\big] < g(Y) - f(Y), \quad Y \in D^c \tag{B.31}$$

then the Markov chain is ergodic and

$$\mathbb{E}\big[f(\widehat{Y})\big] < \infty$$

where the random variable $\widehat{Y}$ has the steady state distribution of the Markov chain $\{Y_n\}_{n=0}^{\infty}$.

Remarks. In the above theorem, Y_0, Y_1 can be replaced by Y_n, Y_{n+1}, respectively, for any integer $n \geq 0$, since $\{Y_n\}_{n=0}^{\infty}$ is a Markov chain.

Now, we can proceed to prove Theorem 4.3.2. Note that, the proof of Theorem 4.3.2 here, using Lemma B.3.1 and Theorem B.3.1, is mostly the same as in [11].

Let $Y_t = Y(t) = \big(Q(t), S(t)\big)$. Then Y_t is an irreducible and aperiodic Markov chain (explained in §B.3). Define $f, g : \mathcal{Y} \to \mathbb{R}^+$ be such that

$$g(Y) = V(Y), f(Y) = \frac{\epsilon_1}{2}\|Q\|_2$$

where $Y = (Q, S)$ and $\epsilon_1 = \frac{1-\rho}{N}$ which is the same as in (B.29). Let $D^c = \big\{Y : V(Y) > B\big\}$, for B specified in the proof of Lemma B.3.1. It is clear that (B.30) holds from the definition of D^c. By Lemma B.3.1 (note $\epsilon = \frac{\epsilon_1}{2}$ in Lemma B.3.1), Inequality (B.31) also holds (by replacing Y_t and Y_{t+1} in (B.2) by Y_0 and Y_1 respectively). By Theorem B.3.1, we have that the Markov chain $Y(t) = \big(Q(t), S(t)\big)$ converges in distribution to $\widehat{Y} = (\widehat{Q}, \widehat{S})$, and that $\mathbb{E}\big[f(\widehat{Y})\big] < \infty$. Therefore, $\mathbb{E}\big[\|\widehat{Q}\|_2\big] = \frac{2}{\epsilon_1}\mathbb{E}\big[f(\widehat{Y})\big] < \infty$.

Given any outcome ω, the (deterministic) N^2-dimensional vector satisfies

$$\|\widehat{Q}(\omega)\|_1 \leq N\|\widehat{Q}(\omega)\|_2$$

by the *Cauchy-Schwarz inequality*.

Therefore,

$$\mathbb{E}\big[\|\widehat{Q}\|_1\big] \leq N\mathbb{E}\big[\|\widehat{Q}\|_2\big] < \infty$$

This completes the proof of Theorem 4.3.2.

B.4 Proof of Lemma 4.3.1

Let Q be the VOQ length vector at the current time t; we do not use the notation $Q(t)$ here because the proof does not involve the term t. Let S_Q be a maximum weighted matching *w.r.t.* Q, and let W_Q denote its weight. Given any $\epsilon > 0$, we derive another matching $S' \subseteq S_Q$ from S_Q as follows: remove every edge (*i.e.,* VOQ) from S_Q whose weight (*i.e.,* VOQ length) is less than $\frac{\epsilon}{N}W_Q$. Since there can be at most N edges in any matching, the weight of S' satisfies $\langle S', Q\rangle \geq W_Q - N \cdot \frac{\epsilon}{N}W_Q > (1-\epsilon)W_Q$.

Recall that in the proposing phase, QPS samples a set of edges (not necessarily a matching), which we denote as U. Next, we prove that, U contains all edges in S' (*i.e.,* $S' \subseteq U$) with at least a constant (*i.e.,* not as a function of Q) probability $\delta = \big(\frac{\epsilon}{N^2}\big)^N$. Given any edge $e = (i, j) \in S'$ (*i.e.,* j^{th} VOQ at input port i), its weight is at least $\frac{\epsilon}{N}W_Q$ since all edges lighter than that would have been removed earlier. Since the weight of any edge can be at most W_Q, the total weight of all edges (VOQs) incident on vertex (input port) i is at

most NW_Q. Hence the probability that this edge $e = (i, j)$ (*i.e.,* output port j) is sampled by input port i in the QPS proposing phase is at least $(\frac{\epsilon}{N} W_Q)/(NW_Q) = \frac{\epsilon}{N^2}$. Since every input port makes the sampling decision independently, the probability that all edges in S' are sampled during the QPS proposing phase is at least $\left(\frac{\epsilon}{N^2}\right)^{|S'|} \geq \left(\frac{\epsilon}{N^2}\right)^N$, where $|S'|$ is the number of edges in S'.

Now suppose the event $S' \subseteq U$ happens during the QPS proposing phase. We show that the final matching accepted by the output ports, during the QPS accepting phase, is at least as heavy as S'. This is however clear from the following two facts. First, given any edge $e = (i, j) \in S'$, it is either accepted by output port j or beaten by another edge (*i.e.,* proposal) e' to output port j that has a heavier (or equal) weight (VOQ length). Second, when the latter happens, since S' is a matching, e' will not compete with (and beat) any edge in S' other than e.

Remarks. Lemma 4.3.1 continues to hold if the "longest VOQ first" accepting strategy is replaced by the aforementioned proportional accepting (PA) strategy (see §4.1.1). Let $\mathcal{E}$ be the event that S' is contained in the final matching. To prove this remark, it suffices to show that there is a constant (*i.e.,* not as a function of Q) probability for $\mathcal{E}$ to happen, conditioned upon the happening of the event $S' \subseteq U$. Using the same argument as above for proving that the event $S' \subseteq U$ happens with a probability that is at least $\left(\frac{\epsilon}{N^2}\right)^N$, we can prove that $\mathcal{E}$ happens conditionally with a probability that is at least $\left(\frac{\epsilon}{N^2}\right)^N$.

B.5 More Performance Evaluations

B.5.1 Mean Delay Performance for FQPS

Here, we consider several alternative functions $f(\cdot)$ of the queue lengths for FQPS, besides the VOQ lengths (*i.e.,* $f(x) = x$) used in QPS, to see if they can deliver better mean delay performance than QPS. We present the simulation results for two types of functions:

(1) $f(x) = x^\alpha$ for $\alpha = 2, 3, 4, \infty$: inspired by the functions considered in MWM-α [29], and

(2) $f(x) = \log(x+1)$: inspired by the log-weights used in MWM-0^+ [30].

The case $\alpha = \infty$ is an extreme case in which each input port samples the longest VOQ (with ties broken uniformly randomly) and proposes to the corresponding output port.

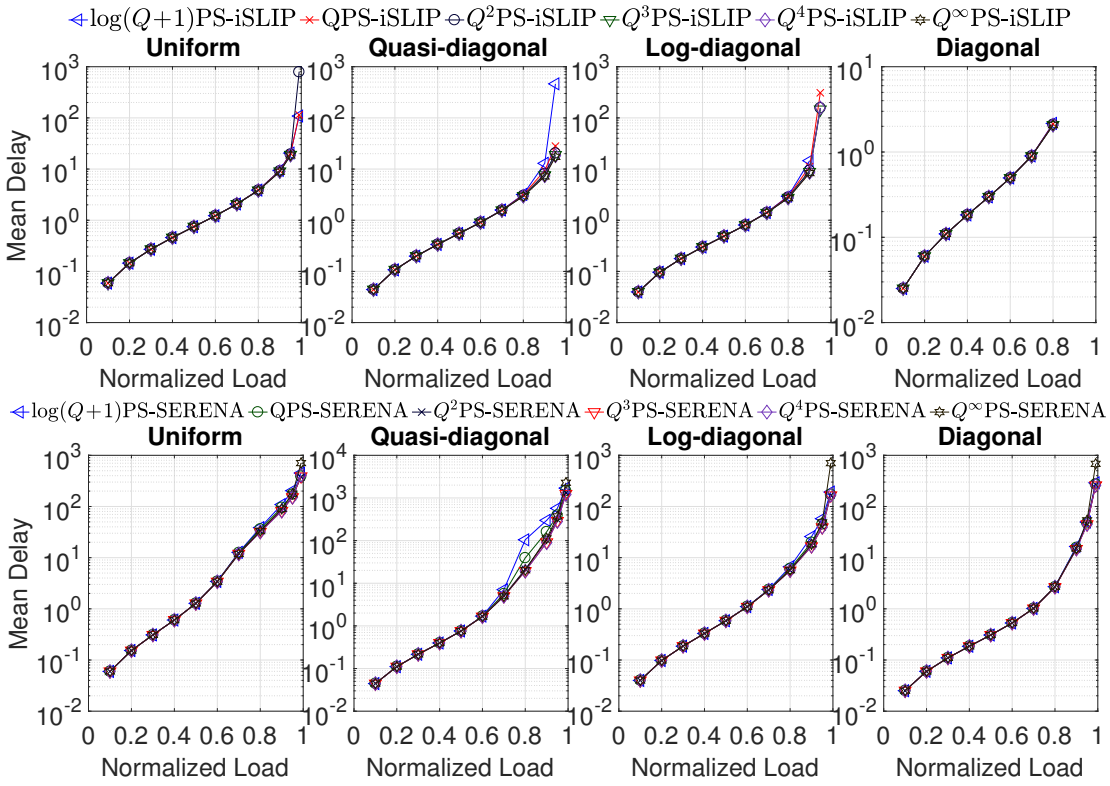


Figure B.1: Mean delays for different FQPS-iSLIP and FQPS-SERENA under the 4 traffic patterns.

Figure B.1 presents the mean delays of FQPS-augmented scheduling algorithms under the 4 traffic patterns and a range of normalized loads. By selecting a proper α, we can indeed achieve marginal improvements (*e.g.,* when $\alpha = 2, 3, 4$). However, when $\alpha \to \infty$, the mean delay increases dramatically when the load is high. This is not surprising because at high loads, a high α strategy severely penalizes short VOQs by blocking them from being serviced until they themselves become long enough, resulting in poor delay performance. Furthermore, the mean delays of the scheduling algorithms are similar when the load is light, but as the load increases, the performance gaps between the FQPS-augmented algorithms with different α values increase (though the differences remain small). Surprisingly,

unlike MWM-α where mean the delay increases as α increases [29], for FQPS, the relationship between the mean delay and α is not so straightforward. On one hand, the mean delay performance is generally slightly better in cases $\alpha = 2, 3, 4$ than that in QPS (*i.e.,* $\alpha = 1$). On the other hand, in the case $\alpha = \infty$, the mean delay performance becomes much worse than that in QPS.

We also see that, unlike in MWM-0^+ [30], using $f(x) = \log(x + 1)$ for FQPS actually increases the mean delay, as compared to QPS. The reason for this is that the use of the $\log(\cdot)$ weight function results in the probabilities of sampling the longer VOQs being very close to those of sampling the shorter queues. Such an almost weight-oblivious way of sampling intuitively does not yield good performance.

While there is slight improvement in mean delay for properly selected α under all traffic patterns, from Figure B.1, we see that the difference between QPS-SERENA (QPS-iSLIP) and the FQPS-SERENA (FQPS-iSLIP) is, at best, marginal. Implementing FQPS, however, requires more complex data structures (and more space), such as a binary search tree. Such an implementation requires $O(\log N)$ (per packet) time complexity for the operations (insertion, deletion, *etc.*). In contrast, the $O(1)$ complexity of QPS makes it a far more attractive and practical solution. To summarize, all factors considered, QPS offers the best tradeoff between performance and time/implementation complexities within the FQPS family.

B.5.2 How Mean Delay Scales with N

In the section, we investigate how the mean delays for QPS-augmented scheduling algorithms scale with the number of input/output ports N. We have simulated four different N values: $N = 16, 32, 64, 128$.

Figure B.2 (the 1^{st} row) shows the mean delays for QPS-iSLIP, iSLIP, iSLIP-ShakeUp, iLQF, and MWM under the normalized load of 0.75 (some algorithms are not stable under load factor 0.8) and the 4 different traffic patterns. From Figure B.2, we can see that all

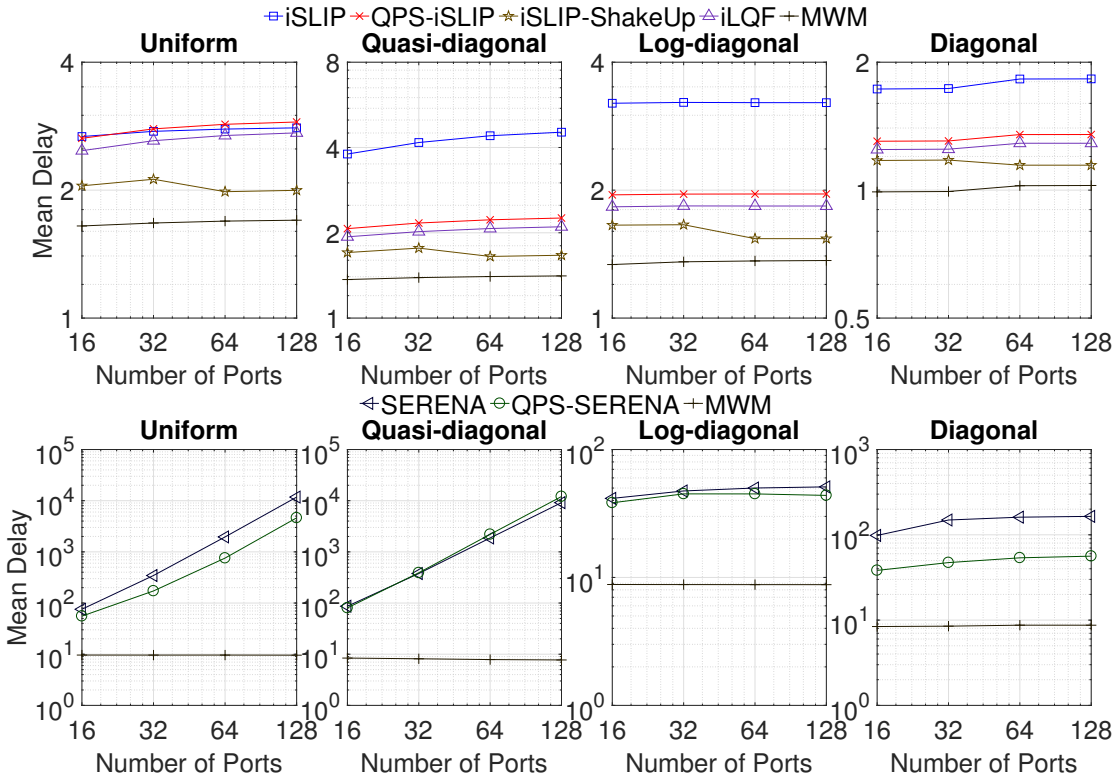


Figure B.2: Mean delays scaling with number of ports N for different scheduling algorithms under the 4 traffic patterns.

four scheduling algorithms scale quite well under all 4 traffic patterns: In every case, the mean delay nearly remains constant when N increases.

In Figure B.2 (both rows), the mean delay of MWM (under 0.75 load in the 1^{st} row and 0.95 load in the 2^{nd} row) is nearly a constant *w.r.t.* N. This scaling behavior of MWM to a certain degree confirms a theoretical result proven in [77]. It states that the average total queue length (across all N input ports) under an optimal algorithm scales linearly with N as $\frac{N}{1-\rho}$, where $\rho \in (0, 1)$ is the load factor. Suppose this total average queue length is furthermore nearly evenly distributed across the N input ports by an optimal algorithm, the mean delay (proportional to the average per-port queue length in the steady state) is expected to be nearly constant when N increases.

Figure B.2 (the 2^{nd} row) shows the mean delays for QPS-SERENA against SERENA and MWM under the normalized load of 0.95 and the 4 different traffic patterns. As we can see, QPS-SERENA outperforms SERENA and the gap increases when N increases, under

all traffic patterns except the quasi-diagonal. In addition, under the log-diagonal and the diagonal traffic patterns, both QPS-SERENA and SERENA achieve near-optimal scaling (*i.e.,* nearly constant as a function of N) of mean delay, whereas under the uniform and the quasi-diagonal traffic patterns, the mean delay grows roughly quadratically with N (*i.e.,* $O(N^2)$ scaling).

B.5.3 "Longest VOQ First" vs. Proportional Accepting

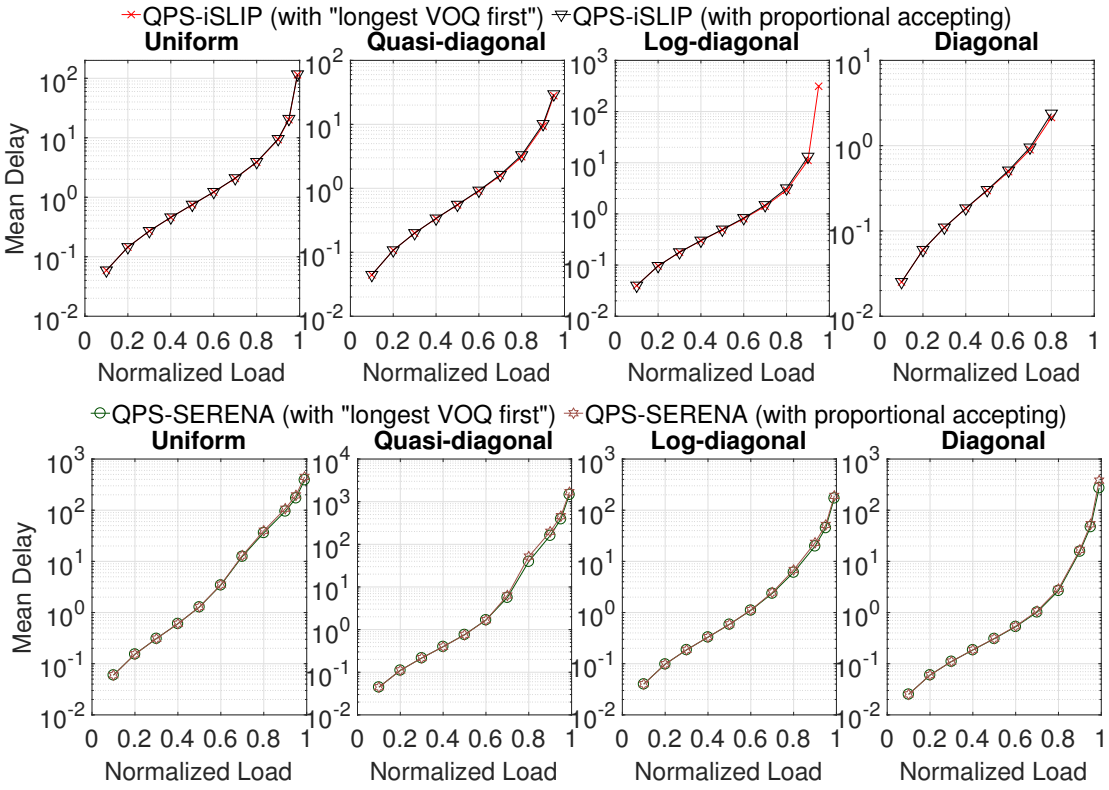


Figure B.3: Mean delays for QPS-iSLIP and QPS-SERENA with the 2 different accepting strategies under *i.i.d.* Bernoulli traffic arrivals with the 4 traffic patterns.

In this section, we compare the performance between the two different accepting strategies we proposed in §4.1.1: "longest VOQ first" and proportional accepting (PA). Figure B.3 compares QPS-iSLIP with the 2 different accepting strategies (the 1^{st} row) and QPS-SERENA with the 2 different accepting strategies (the 2^{nd} row), in terms of mean delay, under *i.i.d.* Bernoulli traffic arrivals with the 4 different traffic patterns. Similarly,

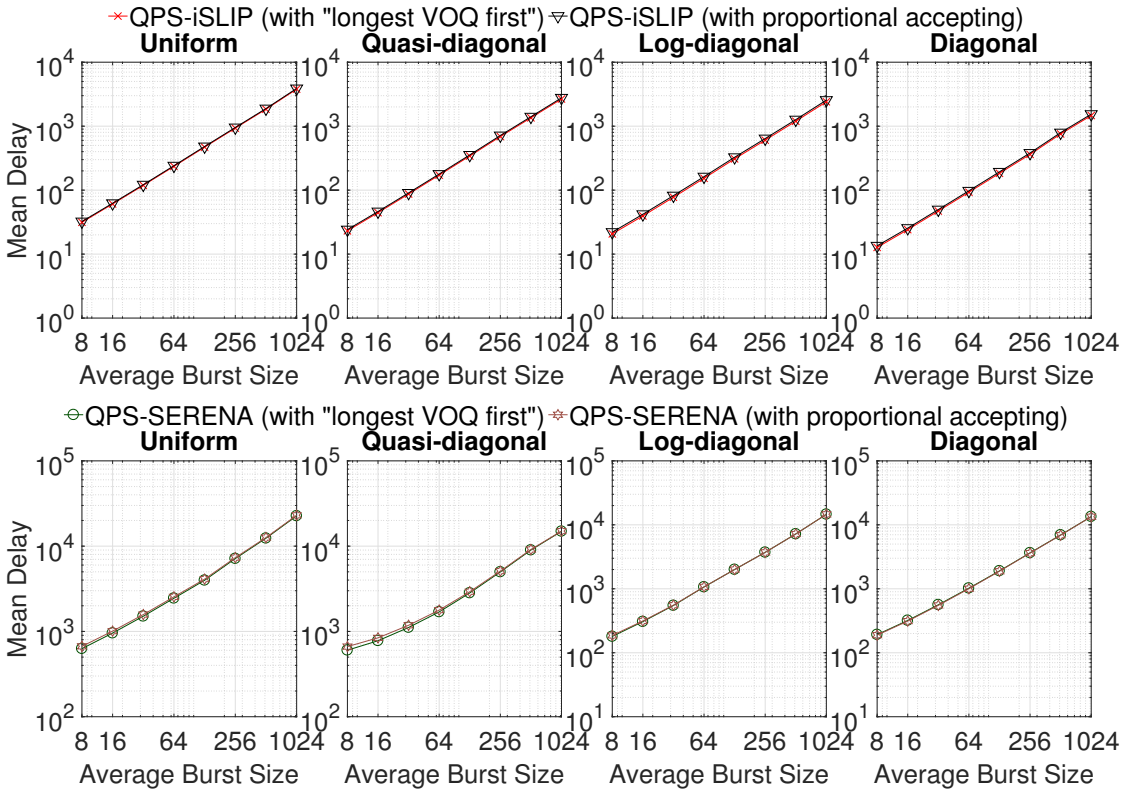


Figure B.4: Mean delays for QPS-iSLIP (offered load: 0.75) and QPS-SERENA (offered load: 0.95) with the 2 different accepting strategies under bursty traffic arrivals with the 4 traffic patterns.

in Figure B.4, the 1^{st} row compares QPS-iSLIP with the 2 different accepting strategies under bursty traffic arrivals with an offered load of 0.75, and the 2^{nd} row compares QPS-SERENA with the 2 different accepting strategies under bursty traffic with an offered load of 0.95. Figure B.3 and Figure B.4 show that PA results in either slightly worse or similar mean delay performances than "longest VOQ first" in all these scenarios.

B.5.4 QPS vs. O(1) Algorithm in [37]

Figure B.5 compares QPS-iSLIP and QPS-SERENA against the $O(1)$ scheduling algorithm in [37], in terms of mean delay, under *i.i.d.* Bernoulli traffic arrivals with the 4 different traffic patterns. Figure B.5 clearly shows that the mean delays of the $O(1)$ algorithm in [37] are between 3 and 4 orders of magnitudes larger than those of QPS-iSLIP and QPS-SERENA under all workload conditions. Note that in Figure B.5, only mean delays under offered

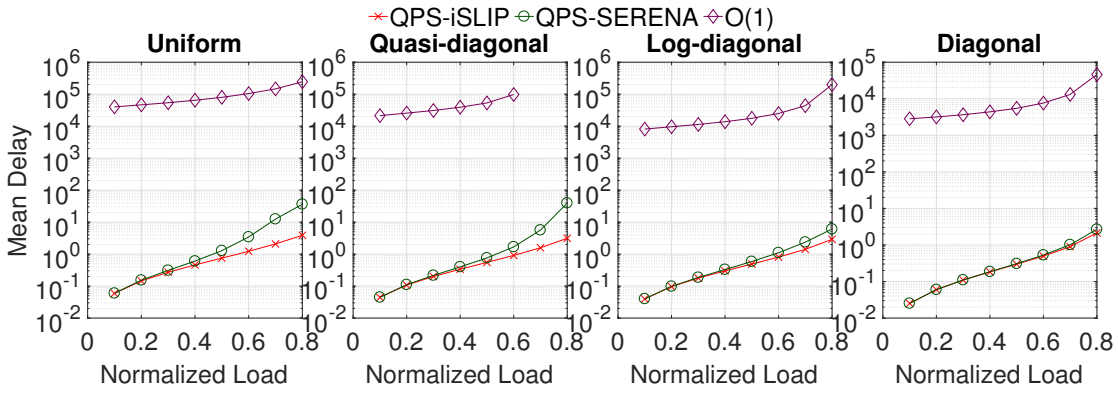


Figure B.5: Mean delays for QPS-iSLIP/QPS-SERENA against $O(1)$ algorithm in [37] under the 4 traffic patterns.

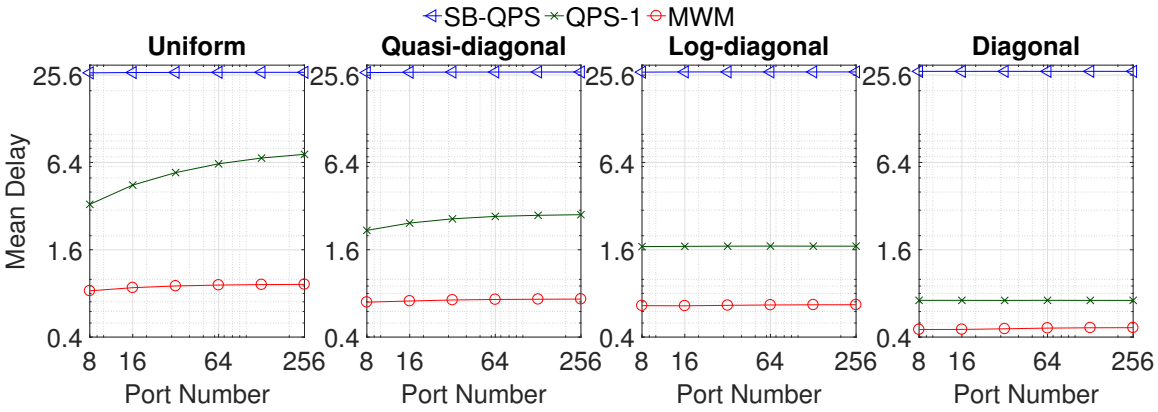
loads ≤ 0.8 (and ≤ 0.6 for quasi-diagonal load matrices) are reported for the $O(1)$ algorithm in [37], because its simulation could not reach the steady state within a reasonable amount of time, when the offered load is higher than that. As explained earlier, this phenomenon is expected, because such Glauber Dynamics based scheduling algorithms converge to the steady state very slowly when the number of ports (or wireless nodes) N is large and the traffic load is high [39]. Indeed, for the $O(1)$ algorithm to converge under an offered load of just 0.8, we had to increase the number of time slots in the simulation to at least $20{,}000 \times N^2$, which is more than three times that was necessary for any other simulation run.

APPENDIX C

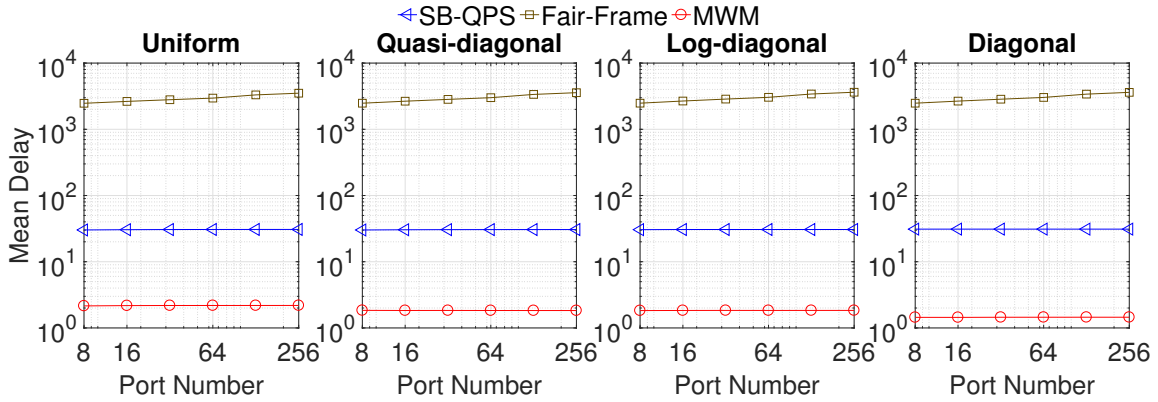
APPENDIX FOR CHAPTER 6

C.1 More Performance Evaluations

C.1.1 How Mean Delay Scales with N



(a) Comparison against QPS-1 and MWM (offered load: 0.6).



(b) Comparison against Fair-Frame and MWM (offered load: 0.8).

Figure C.1: Mean delays scaling with number of ports N under the 4 traffic patterns.

In this section, we investigate how mean delays of SB-QPS, QPS-1, Fair-Frame, and MWM scale with the number of input/output ports N under *i.i.d.* Bernoulli traffic arrivals. We have simulated seven different N values: $N = 8, 16, 32, 64, 128, 256, 512$. We have

simulated different offered loads, here we only present the results under an offered load of 0.6 (when comparing with QPS-1) and that of 0.8 (when comparing with Fair-Frame); other offered loads lead to similar conclusions. The results are shown in Figure C.1a and Figure C.1b.

Figure C.1a compares mean delays of SB-QPS, QPS-1, and MWM under the 4 different traffic patterns with an offered load of 0.6; Figure C.1b compares the mean delays of SB-QPS, Fair-Frame, and MWM under the 4 different traffic patterns with an offered load of 0.8. Both of them clearly show that mean delays of SW-QPS, like those of MWM, are almost independent of N, whereas those of Fair-Frame grow logarithmically with N (*i.e.,* $O(\log N)$ scaling) as shown in Figure C.1b.

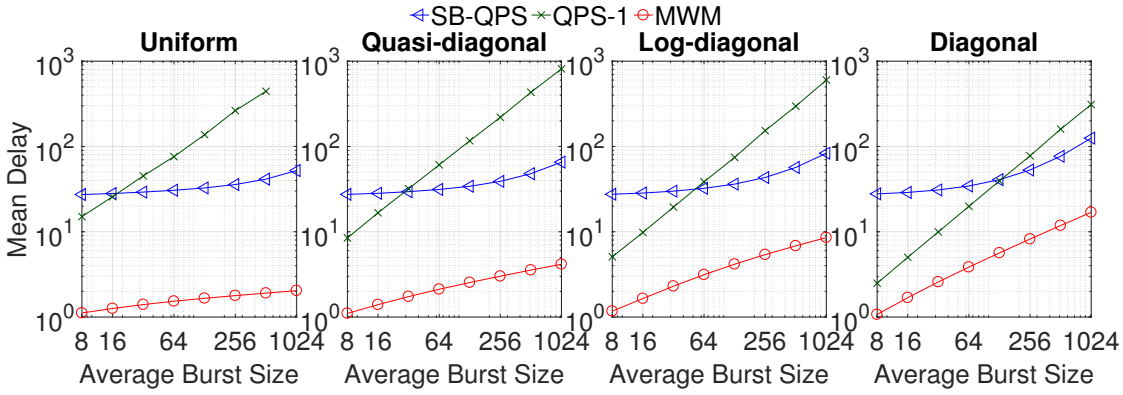
C.1.2 Bursty Arrivals

In real networks, packet arrivals are likely to be bursty. In this section, we evaluate the performances of SB-QPS, QPS-1, Fair-Frame and MWM under bursty traffic arrivals, generated by a two-state ON-OFF arrival process that we have described in §4.4.3.2.

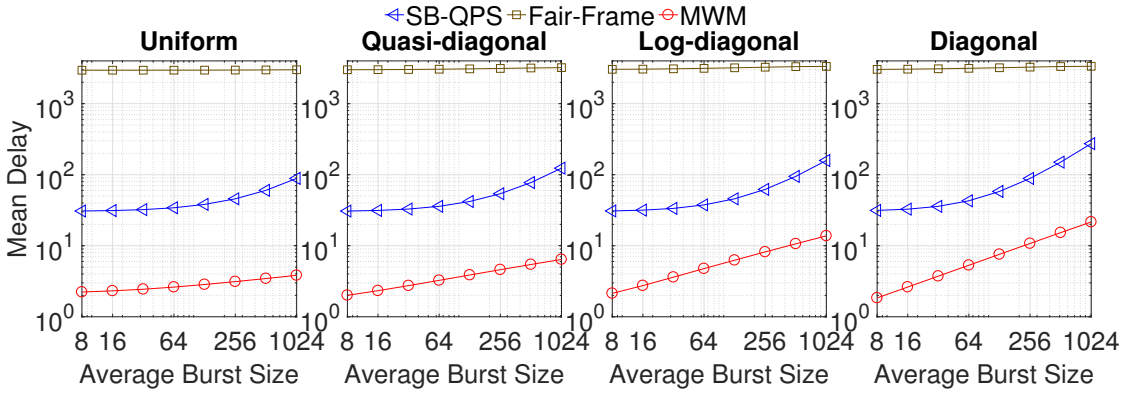
We evaluate the mean delay performances of these four algorithms, with the average burst size ranging from 8 to $1{,}024$ packets, under a moderate offered load of 0.6 and a relatively heavy offered load of 0.8, respectively. The simulation results for the former are shown in Figure C.2a; those for the later are shown in Figure C.2b. One point for QPS-1 is missing in the leftmost sub-figure in Figure C.2a, because QPS-1 is not stable when the burst size becomes $1{,}024$ under the uniform traffic pattern and an offered load of 0.6.

Figure C.2a clearly show that when the average burst size is small, SB-QPS has higher mean delays than QPS-1. However, when the average burst size is large enough (say > 16 under the uniform traffic), SB-QPS starts to outperform QPS-1 by an increasingly wider margin, in both absolute and relative terms, as the burst size becomes larger. Figure C.2b clearly shows that SB-QPS significantly outperforms Fair-Frame under all burst sizes and all different traffic patterns, though the performance gaps get smaller, as the burst size

becomes larger.



(a) Comparison against QPS-1 and MWM (offered load: 0.6).



(b) Comparison against Fair-Frame and MWM (offered load: 0.8).

Figure C.2: Mean delays under bursty traffic arrivals with the 4 traffic patterns.

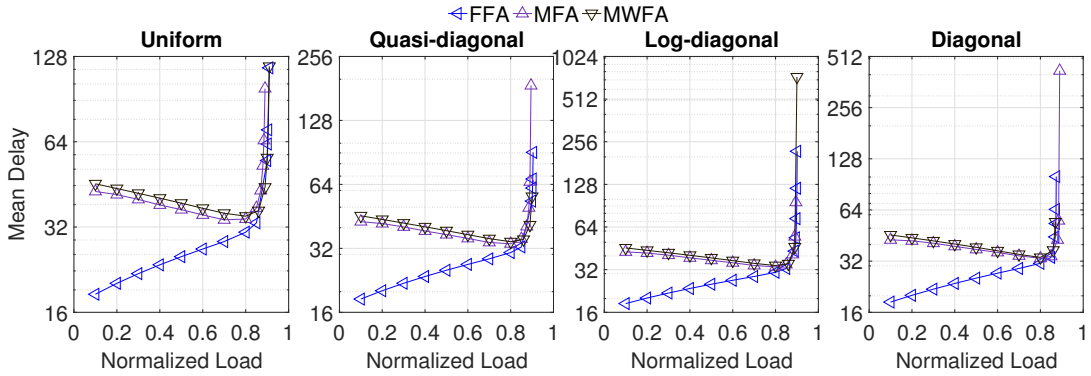


Figure C.3: Mean delays of SB-QPS with the 3 different accepting strategies under the 4 traffic patterns.

C.1.3 FFA vs. MFA vs. MWFA

In this section, we compare the performances of SB-QPS using the three different accepting strategies we described in §6.2: First Fit Accepting (FFA), Maximum Fit Accepting (MFA), and Maximum Weighted Fit Accepting (MWFA).

Figure C.3 presents the mean delays of SB-QPS with the 3 different accepting strategies under *i.i.d.* Bernoulli traffic arrivals with the 4 different traffic patterns. Figure C.3 clearly shows that MFA and MWFA result in either slightly worse or similar mean delay and maximum sustainable throughput performances than FFA in all these scenarios.

REFERENCES

[1] Y. Dai, K. Wang, G. Qu, L. Xiao, D. Dong, and X. Qi, "A scalable and resilient microarchitecture based on multiport binding for high-radix router design," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 429–438.

[2] C. Cakir, R. Ho, J. Lexau, and K. Mai, "Scalable high-radix modular crossbar switches," in *Proceedings of the IEEE Symposium on High-Performance Interconnects (HOTI)*, Aug. 2016, pp. 37–44.

[3] N. Chrysos, C. Minkenberg, M. Rudquist, C. Basso, and B. Vanderpool, "SCOC: High-radix switches made of bufferless clos networks," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 402–414.

[4] P. Giaccone, B. Prabhakar, and D. Shah, "Randomized scheduling algorithms for high-aggregate bandwidth switches," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 546–559, May 2003.

[5] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201, Apr. 1999.

[6] Y. Tamir and G. L. Frazier, "High-performance multi-queue buffers for VLSI communications switches," *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2, pp. 343–354, May 1988.

[7] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," *IEEE Transactions on Communications*, vol. 47, no. 8, pp. 1260–1267, Aug. 1999.

[8] A. S. Asratian, T. M. J. Denley, and R. Häggkvist, *Bipartite Graphs and Their Applications*. USA: Cambridge University Press, 1998, ISBN: 052159345X.

[9] A. Mekkittikul and N. McKeown, "A practical scheduling algorithm to achieve 100% throughput in input-queued switches," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 1998, 792–799 vol.2.

[10] D. Shah, P. Giaccone, and B. Prabhakar, "Efficient randomized algorithms for input-queued switch scheduling," *IEEE Micro*, vol. 22, no. 1, pp. 10–18, Jan. 2002.

[11] L. Tassiulas, "Linear complexity algorithms for maximum throughput in radio networks and input queued switches," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, San Francisco, CA, USA, Mar. 1998, pp. 533–539.

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

[13] J. v. 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.

[14] I. Olkin and A. W. Marshall, *Inequalities: Theory of Majorization and Its Applications*. Academic press, 2016.

[15] S. P. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability*. Springer Science & Business Media, 2012.

[16] 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*, vol. 11, no. 4, pp. 319–352, Nov. 1993.

[17] B. Hu, K. L. Yeung, Q. Zhou, and C. He, "On iterative scheduling for input-queued switches with a speedup of $2 - 1/N$," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3565–3577, Dec. 2016.

[18] A. Eryilmaz, R. Srikant, and J. R. Perkins, "Throughput-optimal scheduling for broadcast channels," in *Proceedings of the ITCom (Modeling and Design of Wireless Networks)*, Denver, CO, Aug. 2001.

[19] M. J. Neely, "Delay analysis for maximal scheduling in wireless networks with bursty traffic," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2008.

[20] ——, "Delay analysis for maximal scheduling with flow control in wireless networks with bursty traffic," *IEEE/ACM Transactions on Networking*, vol. 17, no. 4, pp. 1146–1159, Aug. 2009.

[21] J. Dai and B. Prabhakar, "The throughput of data switches with and without speedup," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Tel Aviv, Israel, Mar. 2000, pp. 556–564.

[22] G. Aggarwal, R. Motwani, D. Shah, and A. Zhu, "Switch scheduling via randomized edge coloring," in *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 2003, pp. 502–512.

[23] M. J. Neely, E. Modiano, and Y. S. Cheng, "Logarithmic delay for N $\times$ N packet switches under the crossbar constraint," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 657–668, Jun. 2007.

[24] L. Wang, T. Lee, and W. Hu, "A parallel complex coloring algorithm for scheduling of input-queued switches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1456–1468, 2018.

[25] L. Gong, L. Liu, S. Yang, J. J. Xu, Y. Xie, and X. Wang, "SERENADE: A parallel iterative algorithm for crossbar scheduling in input-queued switches," in *Proceedings of the IEEE International Conference on High Performance Switching and Routing (HPSR)*, May 2020.

[26] L. Gong, P. Tune, L. Liu, S. Yang, and J. J. Xu, "Queue-proportional sampling: A better approach to crossbar scheduling for input-queued switches," *Proceedings of the ACM on Measurement and Analysis of Computing Systems - SIGMETRICS*, vol. 1, no. 1, 3:1–3:33, Jun. 2017.

[27] L. Gong, J. J. Xu, L. Liu, and S. T. Maguluri, "QPS-r: A cost-effective iterative switching algorithm for input-queued switches," in *Proceedings of the EAI International Conference on Performance Evaluation Methodologies and Tools (ValueTools)*, Tsukuba, Japan: Association for Computing Machinery, 2020, 19–26, ISBN: 9781450376464.

[28] R. Duan and H.-H. Su, "A scaling algorithm for maximum weight matching in bipartite graphs," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Kyoto, Japan, 2012, pp. 1413–1424.

[29] I. Keslassy and N. McKeown, "Analysis of scheduling algorithms that provide 100% throughput in input-queued switches," in *Proceedings of the Allerton Conference on Communication, Control and Computing*, Oct. 2001.

[30] D. Shah and D. Wischik, "Optimal scheduling algorithms for input-queued switches," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Barcelona, Spain, Apr. 2006, pp. 1–11.

[31] M. Fayyazi, D. Kaeli, and W. Meleis, "Parallel maximum weight bipartite matching algorithms for scheduling in input-queued switches," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, New Mexico, USA, Apr. 2004, pp. 4–11.

[32] M. Bayati, B. Prabhakar, D. Shah, and M. Sharma, "Iterative scheduling algorithms," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Anchorage, AK, USA, May 2007, pp. 445–453.

[33] M. Bayati, D. Shah, and M. Sharma, "Max-product for maximum weight matching: Convergence, correctness, and lp duality," *IEEE Transactions on Information Theory*, vol. 54, no. 3, pp. 1241–1251, Mar. 2008.

[34] G. R. Gupta, S. Sanghavi, and N. B. Shroff, "Node weighted scheduling," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, USA, Jun. 2009, pp. 97–108.

[35] S. Atalla, D. Cuda, P. Giaccone, and M. Pretti, "Belief-propagation-assisted scheduling in input-queued switches," *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 2101–2107, Oct. 2013.

[36] N. McKeown, "Scheduling algorithms for input-queued cell switches," Ph.D. dissertation, University of California at Berkeley, May 1995.

[37] S. Ye, T. Shen, and S. Panwar, "An $O(1)$ scheduling algorithm for variable-size packet switching systems," in *Proceedings of the Allerton Conference on Communication, Control and Computing*, Illinois, USA, Sep. 2010, pp. 1683–1690.

[38] S. Rajagopalan, D. Shah, and J. Shin, "Network adiabatic theorem: An efficient randomized protocol for contention resolution," in *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, USA: ACM, 2009, pp. 133–144, ISBN: 978-1-60558-511-6.

[39] J. Ghaderi and R. Srikant, "On the design of efficient csma algorithms for wireless networks," in *Proceedings of the IEEE Conference on Decision and Control (CDC)*, Dec. 2010, pp. 954–959.

[40] D. Shah and J. Shin, "Randomized scheduling algorithm for queueing networks," *The Annals of Applied Probability*, vol. 22, no. 1, pp. 128–171, 2012.

[41] J. Ni, B. Tan, and R. Srikant, "Q-CSMA: Queue-length-based CSMA/CA algorithms for achieving maximum throughput and low delay in wireless networks," *IEEE/ACM Transactions on Networking*, vol. 20, no. 3, pp. 825–836, Jun. 2012.

[42] E. Vigoda, "A note on the glauber dynamics for sampling independent sets," *Electronic Journal of Combinatorics*, vol. 8, no. 1, pp. 1–8, 2001.

[43] B. Hu, F. Fan, K. L. Yeung, and S. Jamin, "Highest rank first: A new class of single-iteration scheduling algorithms for input-queued switches," *IEEE Access*, vol. 6, pp. 11 046–11 062, 2018.

[44] Y. Li, S. Panwar, and H. J. Chao, "On the performance of a dual round-robin switch," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Anchorage, AK, USA, Apr. 2001, 1688–1697 vol. 3.

[45] A. Scicchitano, A. Bianco, P. Giaccone, E. Leonardi, and E. Schiattarella, "Distributed scheduling in input queued switches," in *Proceedings of the IEEE International Conference on Communications (ICC)*, Jun. 2007, pp. 6330–6335.

[46] D. Lin, Y. Jiang, and M. Hamdi, "Selective-request round-robin scheduling for VOQ packet switch architecture," in *Proceedings of the IEEE International Conference on Communications (ICC)*, Jun. 2011, pp. 1–5.

[47] X. Li and I. Elhanany, "Stability of a frame-based oldest-cell-first maximal weight matching algorithm," *IEEE Transactions on Communications*, vol. 56, no. 1, pp. 21–26, Jan. 2008.

[48] R. Rojas-Cessa and C. Lin, "Captured-frame matching schemes for scalable input-queued packet switches," *Computer communications*, vol. 30, no. 10, pp. 2149–2161, 2007.

[49] T. Lee, Y. Wan, and H. Guan, "Randomized Δ-edge colouring via exchanges of complex colours," *International Journal of Computer Mathematics*, vol. 90, pp. 228–245, Feb. 2013.

[50] K. Seong, R. Narasimhan, and J. M. Cioffi, "Queue proportional scheduling in gaussian broadcast channels," in *Proceedings of the IEEE International Conference on Communications (ICC)*, vol. 4, Jun. 2006, pp. 1647–1652.

[51] K. Seong, R. Narasimhan, and J. M. Cioffi, "Queue proportional scheduling via geometric programming in fading broadcast channels," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 8, pp. 1593–1602, Aug. 2006.

[52] B. Li and R. Srikant, "Queue-proportional rate allocation with per-link information in multihop networks," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Portland, OR, USA, Jun. 2015, pp. 97–108, ISBN: 978-1-4503-3486-0.

[53] N. Walton, "Concave switching in single and multihop networks," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Austin, Texas, USA, Jun. 2014, pp. 139–151, ISBN: 978-1-4503-2789-3.

[54] E. Modiano, D. Shah, and G. Zussman, "Maximizing throughput in wireless networks via gossiping," in *Proceedings of the ACM SIGMETRICS/PERFORMANCE*

Joint International Conference on Measurement and Modeling of Computer Systems, Saint Malo, France: ACM, 2006, pp. 27–38, ISBN: 1-59593-319-0.

[55] X. Lin and N. B. Shroff, "The impact of imperfect scheduling on cross-layer rate control in wireless networks," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, vol. 3, Mar. 2005, 1804–1814 vol. 3.

[56] L. Chen, S. H. Low, M. Chiang, and J. C. Doyle, "Cross-layer congestion control, routing and scheduling design in ad hoc wireless networks," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2006, pp. 1–13.

[57] P. Chaporkar, K. Kar, X. Luo, and S. Sarkar, "Throughput and fairness guarantees through maximal scheduling in wireless networks," *IEEE Transactions on Information Theory*, vol. 54, no. 2, pp. 572–594, Feb. 2008.

[58] A. Gupta, X. Lin, and R. Srikant, "Low-complexity distributed scheduling algorithms for wireless networks," *IEEE/ACM Transactions on Networking*, vol. 17, no. 6, pp. 1846–1859, Dec. 2009.

[59] B. Ji, C. Joo, and N. B. Shroff, "Delay-based back-pressure scheduling in multihop wireless networks," *IEEE/ACM Transactions on Networking*, vol. 21, no. 5, pp. 1539–1552, Oct. 2013.

[60] A. Israel and A. Itai, "A fast and simple randomized parallel algorithm for maximal matching," *Information Processing Letters*, vol. 22, no. 2, pp. 77–80, Feb. 1986.

[61] J.-H. Hoepman, "Simple distributed weighted matchings," *ArXiv e-prints*, Oct. 2004. eprint: `cs/0410047`.

[62] R. Preis, "Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs," in *Proceedings of the International Symposium on Theoretical Aspects of Computer Science (STACS)*, Trier, Germany, 1999, pp. 259–269, ISBN: 3-540-65691-X.

[63] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Gossip algorithms: Design, analysis and applications," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, vol. 3, Mar. 2005, 1653–1664 vol. 3.

[64] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[65] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, ISBN: 1558603484.

[66] R. Perlman, *Interconnections (2nd Ed.): Bridges, Routers, Switches, and Internetworking Protocols*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000, ISBN: 0-201-63448-1.

[67] M. W. Goudreau, S. G. Kolliopoulos, and S. B. Rao, "Scheduling algorithms for input-queued switches: Randomized techniques and experimental evaluation," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 2000, 1634–1643 vol.3.

[68] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE Transactions on Automatic Control*, vol. 37, no. 12, pp. 1936–1948, Dec. 1992.

[69] D. Shah and M. Kopikare, "Delay bounds for approximate maximum weight matching algorithms for input queued switches," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, vol. 2, 2002, 1024–1031 vol.2.

[70] *HdrHistogram: A high dynamic range (HDR) histogram*, `https://github.com/HdrHistogram/HdrHistogram`.

[71] B. Hajek, *Notes for ECE 467 communication network analysis*, `https://bit.ly/1JtPGu0`, 2006.

[72] J. M. Flegal, G. L. Jones, *et al.*, "Batch means and spectral variance estimators in markov chain monte carlo," *The Annals of Statistics*, vol. 38, no. 2, pp. 1034–1070, 2010.

[73] P. W. Glynn, W. Whitt, *et al.*, "The asymptotic validity of sequential stopping rules for stochastic simulations," *Annals of Applied Probability*, vol. 2, no. 1, pp. 180–198, 1992.

[74] A. Edelman, *Parallel prefix*, `http://courses.csail.mit.edu/18.337/2004/book/Lecture_03-Parallel_Prefix.pdf`, 2004.

[75] W. Stein, *Elementary Number Theory: Primes, Congruences, and Secrets: A Computational Approach*. Springer Science & Business Media, 2008.

[76] R. Tweedie, "The existence of moments for stationary markov chains," *Journal of Applied Probability*, pp. 191–196, 1983.

[77] D. Shah, N. Walton, and Y. Zhong, "Optimal queue-size scaling in switched networks," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, London, England, UK: ACM, 2012, pp. 17–28, ISBN: 978-1-4503-1097-0.

VITA

Long Gong received his B.Eng. degree in Electronic Information Engineering and M.Eng. in Communication and Information Systems from the University of Science and Technology of China (USTC) in 2012 and 2015, respectively. Long joined the School of Computer Science, Georgia Institute of Technology, as a Ph.D. student in August 2015. Hi finished his Ph.D. thesis work on crossbar scheduling algorithms under the guidance of Dr. Jun (Jim) Xu.