

Scheduling Traffic Matrices On General Switch Fabrics

Xiang Wu
AMD

Amit Prakash
Microsoft

Marghoob Mohiyuddin
UC Berkeley

Adnan Aziz
UT Austin

Abstract

A traffic matrix is an $|S| \times |T|$ matrix M , where M_{ij} is a non-negative integer encoding the number of packets to be transferred from source i to sink j . Chang et al. [2] have shown how to efficiently compute an optimum schedule for transferring packets from sources to sinks when the sources and sinks are connected via a rearrangeable fabric such as crossbar. We address the same problem when the switch fabric is not rearrangeable. Specifically, we (1.) prove that the optimum scheduling problem is NP-hard for general switch fabrics, (2.) identify a sub-class of fabrics for which the problem is polynomial-time solvable, and (3.) develop a heuristic for the general case.

1 Context

A switch fabric is an ensemble of links and programmable crosspoints that connect a set of source nodes S to set of sink nodes T [9]. A traffic matrix is an $|S| \times |T|$ matrix M , where M_{ij} is a non-negative integer encoding the number of packets to be transferred from source i to sink j . Given a fabric and matrix M , a schedule is a collection of configurations, where each configuration consists of choices for all programmable crosspoints. These choices result in a set of channels that connect a subset of S to a subset of T . We assume the fabric does not buffer packets internally; hence for a configuration to be valid, no two channels can intersect each other. For each configuration, a fixed-duration cycle is allocated to program the fabric and transfer packets. During each cycle, the transfer is implemented by passing exact one packet through each channel. A schedule Σ is said to complete the matrix M , if by following the procedure above for each configuration in Σ , we can transfer all packets encoded in M from S to T .

A fabric is rearrangeable if given any one-to-one mapping σ from S and T , there exists a valid configuration providing a channel from each node in S to its image under σ . Given n sources and n sinks connected via a rearrangeable fabric (e.g., a crossbar), and an $n \times n$ traffic matrix M , Chang et al. [2] have shown how to efficiently compute

an optimum schedule for completing M , where optimum means the least number of cycles. Their approach is built on a result due to Birkhoff and von Neumann on the decomposition of matrices, which in turn is based on the existence of perfect matches in a Δ -regular bipartite graph [10]. We will refer to the schedule computed by Chang et al. [2] as the *BvN schedule* of M . Their schedule takes exactly $\max\{\max_i \sum_j M_{ij}, \max_j \sum_i M_{ij}\}$ cycles; we will refer to this value as the *BvN bound* of M .

We address the same problem as Chang et al. [2] when the switch fabric from sources to sinks is not rearrangeable, i.e., the switch fabric cannot implement arbitrary mappings from sources to sinks. Such switch fabrics are much cheaper to build than rearrangeable switch fabrics—a crossbar takes $\Theta(n^2)$ crosspoints to connect n sources to n sinks—and next-generation on-chip switch fabrics will likely be of this kind.

To illustrate the need for general switch fabrics, multi-core designs, wherein a collection of processors are integrated onto a single die, are becoming common. For example, Azul Systems has recently announced a design containing 48 cores. These cores are arranged in a mesh-like fashion on the die, and from a VLSI-implementation perspective, it is desirable to connect the cores by having each core directly connected only to its neighbors in the mesh. Consider a 64 core design, organized as a 8×8 mesh. Each core has direct connections only to its neighbors. Viewed as a graph, the mesh is fully connected—given any two cores, there exists a path between them. However, the mesh cannot support arbitrary mappings between the 64 cores. Let κ be a mapping in which each core in the left half maps to a core in the right half. The left and right halves can be separated by removing 8 edges, but to implement κ we need 32 paths from the left to right halves. Therefore κ is infeasible for the mesh fabric, illustrating the need for studying the scheduling problem for general switch fabrics.

Our specific contributions in this paper are:

1. a proof that optimum scheduling for a general switch fabric is NP-hard (Section 3);
2. a polynomial time algorithm for an important class of switch fabrics (Section 4); and

- the development of a heuristic for the decomposition problem (Section 5).

2 Formulation

We represent the switch fabric as an undirected graph $G = (V, E)$. Sources, sinks and intermediate crosspoints all correspond to vertices, and the links are modeled as edges. Note that rows of a traffic matrix correspond to sources, and columns to sinks. We will refer to a switch fabric and its graph interchangeably. The fabric operates on fixed-size packets; segmentation and reassembly are assumed to be performed outside the fabric.

Recall from Section 1 that a valid configuration is a set of non-intersecting channels, which corresponds to a collection of paths in G , and no two paths share a common vertex; we refer to such paths as being *vertex disjoint*.

Given a switch fabric G , a matrix m is defined to be G -feasible if there exists a single configuration that completes m . It follows that a matrix m is feasible iff all entries in m are either 0 or 1, and all source-sink pairs corresponding to 1s in m can be connected by a collection of vertex disjoint paths in G . Note that each configuration in the schedule can be mapped onto a feasible matrix, or equivalently a *vertex-disjoint-path-set* (VDPS). Consequently, we will interchangeably refer to a schedule as a collection of feasible matrices or a collection of VDPSs.

Given a traffic matrix M , and a general switch fabric represented by G , we want to answer the same question as in BvN scheduling: what is the minimum number of feasible matrices, not necessarily distinct, that sum up to M ? We refer to this problem as the *generalized scheduling* of M on G . For the special case where G is rearrangeable, such as a crossbar, the problem reduces to computing the BvN schedule.

We present a small but surprisingly interesting instance of the generalized scheduling problem in Figures 1 and 2. Specifically, it illustrates that building the schedule greedily—that is by always picking the largest possible VDPS—is suboptimum. For this example, the largest VDPS corresponds to the set of packets $\{a, c, f\}$. After a simple enumeration, it is clear that no two packets from $\{b, d, e\}$ can be transferred in one cycle—if a matrix has two 1s at positions $\{(1, 4), (3, 6)\}$, $\{(3, 6), (5, 2)\}$ or $\{(5, 2), (1, 4)\}$, it will become infeasible because of the limited connectivity offered by G . We show the results of the greedily computed schedule in Figure 3 as well as an optimum schedule. Note that the BvN bound for M is 2, corresponding to the schedule $\{(1, 2), (3, 4), (5, 6)\}$, $\{(1, 4), (3, 6), (5, 2)\}$. The limited connectivity of G means it takes an additional cycle to schedule M on G , compared to the case where $\{1, \dots, 6\}$ are interconnected by a crossbar.

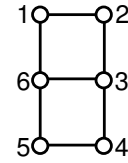


Figure 1. A mesh-structured switch fabric G . Each vertex can be either a source or a sink, but not both, in a cycle.

$$\begin{pmatrix} 0 & 1^a & 0 & 1^b & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1^c & 0 & 1^d \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1^e & 0 & 0 & 0 & 1^f \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2. Traffic matrix M for the fabric in Figure 1. The superscripts are packet identifiers, e.g., we will refer to the packet from source 1 to sink 2 as a .

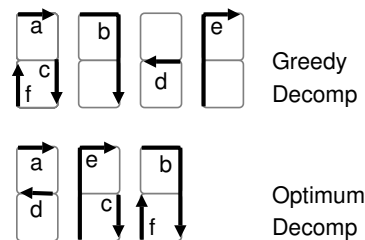


Figure 3. Greedily constructed and optimum schedules for G and M as presented in Figure 1 and 2, respectively.

3 Complexity of Optimum Schedule

From the previous example, it should become apparent that for even a very simple switch fabric and traffic matrix, greedy approaches for computing the schedule can be sub-optimum. This suggests that the scheduling problem may be intrinsically hard, a fact we now prove.

Theorem 3.1 *Given a graph G and a traffic matrix M , determining whether there exist no more than L matrices m_1, \dots, m_L which are G -feasible and whose sum equals M is NP-hard.*

We will use the following lemma:

Lemma 3.1 *Given a graph G and a matrix m , determining whether m is G -feasible is NP-hard.*

Since Lemma 3.1 is the special case of Theorem 3.1 with $L = 1$, it immediately implies the theorem.

We now prove Lemma 3.1.

We will prove this lemma by transforming the 3-CNF-SAT problem to checking if a matrix m is G -feasible. We use the *component design technique* from [6]. For each variable in the clause database we design a *choice component*. For example, suppose the database is $\{(\overline{x_1} + x_2), (x_1 + x_3), (\overline{x_1} + \overline{x_2} + \overline{x_3})\}$; then the choice component for variable x_1 is constructed as in Figure 4.

Remember that this component is part of a graph in which we want to determine whether there is a VDPS connecting each source-sink pair specified. Each vertex with a nonstarred label in the component will become a designated source, and its partner sink will bear the same label starred. So in Figure 4, as we can see from vertex $V1$ to $V1^*$ there are only two possible paths: left one is the $\overline{x_1}$ path, right one is the x_1 path. Since the negated literal appears in both clause 1 and 3. $C1$ and $C3$ and their partners are bridged by vertices on the $\overline{x_1}$ path. Now comes the key observation: in order to connect $V1$ to $V1^*$, we must choose exact one path: $\overline{x_1}$ or x_1 ; and if any clause pair C_i and C_i^* can be connected through the component, the clause's literal of variable x_1 is true (not taken) by the choice made for $V1$ and $V1^*$.

Following exactly the same procedure, we can build components for variable x_2 and x_3 . The final graph G is constructed as in Figure 5. Dotted lines show clause vertices are connected to components' negated literal paths, solid lines show connections to non-negated literals. So to get C_i and C_i^* connected, we need at least one of literals in clause i not taken (set to true) by choice. Thus if we construct a matrix M that has element 1 at positions of (l, l^*) for all label l and it is feasible, the original SAT problem must be satisfiable as all C_i and C_i^* are connected across at least one true literal path. On the other hand, if satisfiable, we can take the path from V_i to V_i^* such that the true

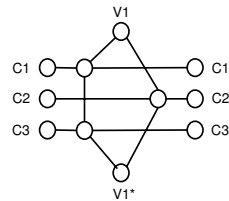


Figure 4. Choice component for x_1 .

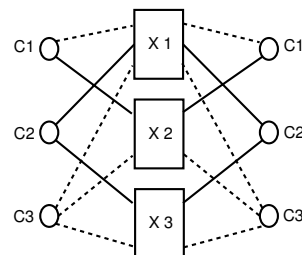


Figure 5. Constructed matrix is G -feasible iff the 3-CNF-SAT problem $\{(\overline{x_1} + x_2), (x_1 + x_3), (\overline{x_1} + \overline{x_2} + \overline{x_3})\}$ is satisfiable.

literal of x_i in the SAT assignment is reserved for bridging clause vertices for all i . By the definition of satisfiability, we can connect all C_i to their partners through at least one component, which means the matrix M is feasible. Finally, the graph size is obviously bounded by a polynomial of the SAT problem size. This completes the proof.

4 A Polynomial Scheduling Algorithm

In this section we present a pseudo-polynomial algorithm that solves the scheduling problem exactly for switch fabrics for which the graph G is a tree; we will refer to such fabrics as *tree fabrics*.

Before presenting the algorithm, we define a few useful functions. The *incidence function* $\chi(p, q, v)$ is 1 if the path from source p to sink q includes v and 0 otherwise. The function χ is well defined because p and q uniquely decide the path connecting them in G . We define the *load function* L_v as follows: $L_v = \sum_{p,q} M_{pq} \chi(p, q, v)$. We will also use the *level* of a vertex extensively in the algorithm. Given an arbitrarily chosen root vertex, the level is defined as the distance obtained by a breadth-first search from the root. Given a path P , if v appears in P and $level(v)$ is the minimum among all vertices in P , we say P roots at v . Evidently each path has one and only one root.

Theorem 4.1 *Given a tree fabric G and traffic matrix M , Algorithm 1 returns an optimum schedule in time*

Algorithm 1 Scheduling on Tree Fabrics

Input: tree G and matrix M **Output:** $Sched$ — a collection of vertex-disjoint-path-sets

```
1: Build a table  $T_l$  mapping a load value  $l$  to the ordered set  $S = \{v | L_v = l\}$ ;  
    $T_l$  is sorted using key  $l$  descending;  $S$  is sorted using key  $level(v)$  ascending;  
2: Build a table  $T_p$  mapping a vertex  $v$  to the set of paths that root at  $v$  ( $T_p$   
   includes all paths needed to discharge the traffic);  
3:  $Sched \leftarrow \emptyset$ ;  
4: while  $T_l$  is not empty do  
5:   Pop the first (maximum load) set  $S$  from  $T_l$ ;  
6:    $VDPS \leftarrow \emptyset$ ;  
7:   while  $S$  is not empty do  
8:     Pop the first (minimum level) vertex  $v$  from  $S$ ;  
9:     Pop one path  $P$  rooting at  $v$  from  $T_p$ ;  
10:    Add  $P$  to  $VDPS$ ;  
11:    for all  $p$  in  $P$  do  
12:      if  $p \in S$  then  
13:        Remove  $p$  from  $S$ ;  
14:      else  
15:        Remove  $p$  from  $T_l[L_p]$ ;  
16:      end if  
17:       $L_p \leftarrow L_p - 1$ ;  
18:      if  $L_p > 0$  then  
19:        Add  $p$  back into  $T_l[L_p]$ ;  
20:      end if  
21:    end for  
22:  end while  
23:  Add  $VDPS$  to  $Sched$ ;  
24: end while
```

$Poly(n, A)$, where n is the size of G and A is the entry in M with the largest value.

First, we make the claim, proved in Lemma 4.1, that we can always find such a P in Line 9 of Algorithm 1. For now simply note that when a VDPS is generated, the set S with the largest load is removed from T_l . Even though some vertices in S will be inserted back, their loads will be decreased by 1 as in Line 17, which means the largest load in table T_l will be decreased by 1 in each iteration of the outermost while loop. This guarantees a finite termination of the algorithm. To be more precise, the main while loop will iterate exactly $\max\{L_v\}$ times, which is bounded by $O(n^2A)$. All operations inside the loop are clearly polynomial time, hence the time bound in Theorem 4.1 is confirmed. One byproduct is that the size of the schedule $Sched$ is proved to be $\max\{L_v\}$, which is the least number of cycles possible because all paths passing that maximum loaded vertex must be used in distinct cycles. Technically, because of the dependence of run-time on the entries of M , the algorithm is pseudo-polynomial. In practice, the entries in M are small, thus the algorithm is truly polynomial time.

To complete the proof of Theorem 4.1, we need to prove the following lemma:

Lemma 4.1 *When Line 9 of Algorithm 1 is encountered during execution, there always exists a path P rooting at v , which is the vertex of the minimum level in S at that moment. Furthermore, such P is vertex disjoint from any other path already added to VDPS.*

We will induce a contradiction by assuming no such path exists. Based on the assumption, we know that all paths in

T_p that include v do not root at v , which means they all include the unique parent of v . Denote this parent vertex as u . This tells us that $L_u \geq L_v$ and $level(u) = level(v) - 1 < level(v)$. Note that v is in the set of maximum load with the minimum level. And now we have another vertex in the same set with a strictly smaller level. This is a contradiction.

Now that we have existence secured, we need to show that any P rooting at v will not overlap with paths already added into VDPS. If there is a path Q in VDPS and a vertex x appears in both P and Q , another contradiction will be shown right next. Now look at the root of Q , i.e., vertex r . First, $level(r) \leq level(v)$ because otherwise v would be popped out before r . Second, v does not show up in Q because otherwise it would have been removed at Line 13 and cannot be popped out from S . Now clearly $v \neq r$ and we shall build the path from r to x , which is part of Q . The first segment is from r to v and vertices in this part all have levels $\leq level(v)$ obviously. The second segment is from v to x . Since x is below v as P roots at v , all vertices except v in this part will have levels $> level(v)$. Looking at the levels, we can instantly see these two segments do not overlap and combined together they form the unique path from r to x in the tree. This shows that v is actually on the path from r to x . A conflict emerges because that is to say v appears in Q .

We remark that the scheduling problem for traffic matrices with *precedence constraints*, which specify certain packets must be transferred before others, is NP-hard even for the tree topology. This result follows from a direct reduction from the multi-machine scheduling problem under partial order constraints [6].

5 Heuristic Scheduling

We proved in Section 3 that the scheduling problem in a general graph is NP-hard. Although we derived a fast algorithm for tree topologies, a tree fabric is inadequate because of its limited connectivity. We will see topologies such as mesh greatly improve performance without changing the asymptotic density of the graph.

Our heuristic is built upon the metric of *congestion* on edges. Consider an instance of the scheduling problem, with G the fabric, and M the traffic matrix, with sources S and sinks T . Define the distance $d_{e,w}$ between an edge $e = \{u, v\}$ and a vertex w by $\max\{d(u, w), d(v, w)\}$, where $d(x, y)$ is length of the shortest path in G between x and y , where edges are of unit length. We define the congestion on edge e by the following equation:

$$C_e = \sum_{s \in S} \frac{W_s}{d_{e,s}} + \sum_{t \in T} \frac{W_t}{d_{e,t}}$$

where W_s and W_t are the corresponding row and column sums for M .

The congestion metric is designed to reflect the attenuating trend when the distances to sources or sinks increase. Also importantly, this metric is fast to calculate using breadth-first search from each source and sink: simply accumulate all source or sink quotients without caring about the order in which vertices are visited. Furthermore, the distance part can be cached since when computing the congestion, the topology is always the original graph G .

Naturally, when the matrix and graph are both specified, we can conduct the calculation on edges to determine C_e throughout the graph. And we will use a combined strategy to generate a VDPS as the best choice of current cycle. After choosing the VDPS, we will transfer the maximum allowed amount of packets through each path. In the end, we update the matrix to reflect the change and go back to the beginning, i.e., recalculation of congestion, choosing VDPS, and updating the matrix.

Algorithm 2 Heuristic to Generate One VDPS

Input: graph G and a copy of matrix M
Output: $VDPS$ — a set of vertex disjoint paths

- 1: $VDPS \leftarrow \emptyset$;
- 2: Calculate C_e for all edges by doing breadth-first search from each source and sink (rows and columns in M);
- 3: **repeat**
- 4: Pick the row or column in M with the largest sum, record the corresponding vertex as v^* ;
- 5: **if** v^* is a source (row chosen) **then**
- 6: Put all sinks into the target set $Target$;
- 7: **else**
- 8: Put all sources into $Target$;
- 9: **end if**
- 10: Compute shortest paths $P = \{p_{v^*w}\}$ from v^* to vertices w in $Target$;
- 11: **for all** $p_{v^*w} \in P$ **do**
- 12: Determine the blockage of the path, the largest C_e among edges in p_{v^*w} ;
- 13: Keep track of the path with the least blockage in p^* connecting v^* to w^* ;
- 14: **end for**
- 15: **if** the entry in M for v^* and $w^* > 0$ **then**
- 16: Add the path p^* to $VDPS$
- 17: **for all** v in p^* **do**
- 18: Remove all edges incident at v ;
- 19: **end for**
- 20: Set the row or column for w^* to zeros;
- 21: **end if**
- 22: Set the row or column for v^* to zeros;
- 23: **until** all entries in M are zeros

Several key points in the heuristics are:

1. Line 4 is a direct extension of the same idea in Algorithm 1 for trees, i.e., we always deal with vertices with the maximum load first. Here we look at the vertex with the most input or output first; this is in the spirit of BvN scheduling.
2. Loop from Line 11 to 14 chooses the path with the least blockage, thereby avoiding congested regions.
3. All vertices in path p^* are isolated as in Line 18. This guarantees the vertex disjoint property of all paths

added.

4. At Line 22, we always zero out a row or column with the largest sum in M and repetition of this operation will eventually turn M into an all-zero matrix, guaranteeing finite termination.
5. The computation of shortest paths in Line 10 with Dijkstra's algorithm is very fast in theory and practice.

5.1 Experiments

Rather than showing results on random traffic matrices, we have focused on traffic matrices which correspond to real applications. Specifically, we used our heuristic to schedule the communication required on a mesh fabric interconnecting processing elements computing the Fast Fourier Transform (FFT) [4], and Low-Density Parity Check Decoding (LDPC) [5]. These computations will be an integral part of next-generation communication standards such as DVB-S2 for satellite digital video broadcasting. Both these computations have high complexity, and need to be implemented using parallel hardware to achieve the desired performance requirements. The results of the parallel hardware units need to be communicated with other units, which yields the traffic matrices; communication is known to be a bottleneck for both FFT [7] and LDPC [1].

The FFT An N -point FFT can be implemented with parallel hardware using $1 + \log_2 N$ stages, where each stage consists of $N/2$ processing elements (PEs) that implement “butterfly” operations in parallel; the results of these operations are passed on to specific processing elements in the next stage [4].

Specifically, between stage l and $l + 1$, $PE_i(l)$ passes its results to two PEs: (1.) $PE_i(l + 1)$, and (2.) depending on i , either $PE_{i+2^{l-1}}(l + 1)$ or $PE_{i-2^{l-1}}(l + 1)$. It is straightforward to encode the results that need to be communicated from one stage to the next as a traffic matrix. Since each PE has two inputs and produces two outputs, there are $N/2$ PEs per stage.

We placed 256 PEs on a 63×63 mesh for a 256 point FFT. Half of them implement stage l , and the other half implement stage $l + 1$. The remaining $63^2 - 256 = 3713$ crosspoints on the mesh are used as routing resources. We give the results of our heuristic on the 256 point FFT in Table 1.

LDPC Decoding An LDPC code is a block code, where there are C bits per block, which include D parity checks. It is most naturally represented as a bipartite graph on a set of C code nodes and D check nodes. The decoding algorithm [5] involves iterations of message passing back

Stage	1	2	3	4	5	6	7	8
Num. Cycles	3	4	6	9	3	4	6	9

Table 1. Scheduling for a 256-point FFT. Num. Cycles is the size of the schedule required for implementing the transfer to the next stage.

Code	C1	C3	C3	C4	C5	C6	C7	C8
Num. Cycles	18	16	16	14	14	15	18	17
BvN Bound	13	15	12	12	13	11	12	13

Table 2. Schedule size for LDPC codes C1–C8, and the corresponding BvN bound.

and forth between connected code and check nodes, and it is this communication that defines the traffic matrix.

We created 8 LDPCs C1–C8 with 96 code and 48 check nodes using randomized code construction techniques [1]. The 144 code and check nodes were embedded on a 23×23 mesh. Results for these 8 different LDPC codes are presented in Table 2. Each entry in the connection matrix corresponds to exact one transfer of a result from a code node to a check node or vice versa.

We also calculated the BvN bounds for these matrices (which, in general, can only be implemented with a rearrangeable network). Our schedules are fairly close to the bounds, reinforcing our confidence in the heuristic.

For both FFT and LDPC, our heuristic computed the schedule in seconds. Our implementation of the heuristic is very straightforward, and it could likely be sped-up greatly, but there is little incentive to do so since the computation is off-line.

6 Discussion

The problem of scheduling a traffic matrix for general switch fabrics has not, to the best of our knowledge, been addressed previously. The most closely related work, which was the inspiration for our own research, is that of Chang *et al.* [2], who restricted their studies to rearrangeable fabrics. (Their followup paper [3] deals with buffering and load balancing.)

There is vast literature on routing for general switch fabrics [7]. Much of the work is on online routing, and assumes buffering in the fabric; there are limited results on offline routing. In all cases, the focus has been on routing individual permutations, rather than traffic matrices. Certainly, a traffic matrix can be decomposed into a sum of permuta-

tion and sub-permutation matrices, which can then be individually scheduled using existing techniques. However, by operating directly on traffic matrices, we have much more flexibility in terms of which packets to transfer.

There are a number of ways in which our research can be extended. The most direct extensions are to fabrics which do allow internal buffering, and to traffic matrices with priorities on packets. Another interesting problem is to apply our results in scenarios where the traffic matrix is not known a priori. In the past, dynamic scheduling has been in this scenario, e.g., the iSLIP algorithm [8] for rearrangeable fabrics. However, by buffering packets we can effectively transform the problem to the offline case; the challenge would be to compute a schedule fast enough that the amount of buffering is not excessive.

References

- [1] A. J. Blanksby and C. J. Howland. A 690-mW 1-Gb/s 1024-b, Rate-1/2 Low-density Parity-check Decoder. *IEEE Journal of Solid-State Circuits*, 37:404–412, March 2002.
- [2] C.-S. Chang, D.-S. Lee, and Y.-S. Jou. Load balanced Birkhoff-von Neumann switches, part I: one-stage buffering. *Computer Communications*, 2001.
- [3] C.-S. Chang, D.-S. Lee, and C.-M. Lien. Load balanced Birkhoff-von Neumann switches, part II: multi-stage buffering. *Computer Communications*, 2001.
- [4] T. H. Cormen, C. E. Leiserson, and R. H. Rivest. *Introduction to Algorithms*. MIT Press, 1989.
- [5] R. G. Gallager. *Low-density parity-check codes*. PhD thesis, MIT, Cambridge, MA, 1962.
- [6] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., 1979.
- [7] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan-Kaufmann, 1991.
- [8] N. McKeown. iSLIP: A Scheduling Algorithm for Input-Queued Switches. *IEEE Transactions on Networking*, 7(2), Apr. 1999.
- [9] J. Turner and N. Yamanaka. Architectural Choices in Large Scale ATM Switches. *IEICE Transactions*, 1998.
- [10] J. van Lint and R. Wilson. *A Course in Combinatorics*. Cambridge University Press, 1992.