

# Configuring a Load-Balanced Switch in Hardware

Srikanth Arekapudi\*, Shang-Tse Chuang\*, Isaac Keslassy\*\*, Nick McKeown\*  
{sarek, stchuang, keslassy, nickm}@stanford.edu

\*Computer Systems Laboratory, Stanford University

\*\*Now with the Technion, Haifa, Israel

**Abstract**—The load-balanced switch architecture is a promising way to scale router capacity. We explained in [1] how it can be used to build a 100Tb/s router with no centralized scheduler, no memory operating faster than the line-rate, no packet mis-sequencing, a 100% throughput guarantee for all traffic patterns, and an optical switch fabric that simply spreads traffic evenly among linecards. This switch fabric uses optical MEMS switches that are reconfigured only when linecards are added and deleted, allowing the router to function when any subset of linecards is present and working.

In [2] we introduced a configuration algorithm that will find a correct configuration of the MEMS switches in polynomial time. However, we found that our algorithm takes over 50 seconds to run in software for a 100Tb/s router. Our goal is to restore the router to operation within 50ms upon failure. So we modify our algorithm for implementation in dedicated hardware. In particular, to simplify the Ford-Fulkerson algorithm in bipartite matches, we reduce memory accesses and use bit manipulation schemes. Then, we decompose permutations using the Slepian-Duguid algorithm and reduce the configuration time with a simplified memory scheme. Our experimental results show that it is possible to achieve the 50ms target.

## I. INTRODUCTION

Our goal is to identify router architectures with predictable throughput and scalable capacity. At the same time, we would like to identify architectures in which optical technology (for example optical switches and wavelength division multiplexing) can be used inside the router to increase capacity by reducing power consumption.

In a previous paper [1] we explained how to build a 100Tb/s Internet router with a single-rack switch fabric built from essentially zero-power passive optics, but without sacrificing throughput guarantees. Compared to routers available today, this is approximately 40 times more switching capacity than can be put in a single rack, with throughput guarantees that no commercial router can match today. The key to the scalability is the use of the *load-balanced switch*, first described by C-S. Chang *et al.* in [3]. In [1] we extended the basic architecture so that it has provably 100% throughput for any traffic pattern, and doesn't mis-sequence packets. It is scalable, has no central scheduler, is amenable to optics, and can simplify the switch fabric by replacing a frequently scheduled and reconfigured switch with a single, fixed, passive mesh of WDM channels.

Unfortunately, as mentioned in [2], the number of linecards present can keep on changing as more and more linecards are added as the network grows or linecards are removed as they fail. The load-balanced switch works by uniformly spreading packets over all linecards, and therefore needs to

be aware of which linecards are present and which are not. If some linecards are missing, the switch fabric must be able to schedule the traffic uniformly over the linecards present. In [1] we described a hybrid electro-optical architecture that solves this problem, and will operate with any subset of linecards. [2] describes an algorithm to configure the switch fabric, and proves that it will always find a valid configuration in polynomial time.

Upon linecard failure we require a restoration time below 50ms in order to provide a fast recovery [4], [5], [6], [7]. However, the polynomial-time algorithm we described previously took over 50 seconds to run. A simple conversion to hardware of the software algorithm would be too slow by at least an order of magnitude because the algorithm is extremely memory intensive. The goal of this paper is to show that a suitably modified hardware implementation can keep the reconfiguration time below 50ms.

The polynomial-time algorithm requires many repetitions of two graph matching algorithms. The first finds the maximum flow in a graph, which is commonly realized using the Ford-Fulkerson [8] algorithm. The second algorithm decomposes a matrix into a minimal number of permutations, which is commonly solved using a Birkhoff-von Neumann [9], [10] decomposition.

Both the Ford-Fulkerson and the Birkhoff-von Neumann algorithms require a large number of memory accesses in order to find matches. Therefore, in order to speed up the running time, we adapt the original algorithms to minimize memory accesses. First we modify the Ford-Fulkerson algorithm to work specifically for bipartite matches. Based on the binary matrix structure specific to our problem, we can then utilize bit-manipulation schemes to reduce the time required to search for new matches.

Second, in order to decompose a matrix into permutations, the Birkhoff-von Neumann decomposition repeatedly finds a permutation using either a maximum size match or a simplified Ford-Fulkerson. By using the Slepian-Duguid algorithm instead, we find all the permutations at the same time. This reduces the number of iterations to one, and therefore the number of pre-processing steps linked with each iteration. In addition, we provide a simple mechanism to search for the matrix elements not yet assigned to a permutation.

Finally, the experimental results show that it is possible to achieve the 50ms target for our 100Tb/s router consisting of up to 640 linecards.

Here is an outline of this paper. Section II provides an

overview of the algorithm used to configure the switch fabric of the 100Tb/s router. Then, sections III and IV respectively present the details of the modified Ford-Fulkerson algorithm and the Slepian-Duguid algorithm. These sections describe how these algorithms are memory-intensive, and how bit manipulation schemes can drastically reduce the number of memory accesses. Finally, in Section V, the simulation results show how the reduction of memory accesses makes the 50ms target feasible.

## II. OVERVIEW OF CONFIGURATION ALGORITHM

Although the configuration algorithm is described fully in [2] (and we assume the reader is familiar with both references [1], [2]), we give a brief reminder of the algorithm here.

As explained in [2], there are  $G$  groups; group  $i$  contains  $L_i$  linecards, and the total number of linecards is:

$$N = \sum_{i=1}^G L_i.$$

We will assume that  $L_1, L_2, \dots, L_G$  are fixed for a given linecard arrangement. Our objective is to create a schedule where linecards spread packets evenly across all other linecards. Therefore, during every frame of  $N$  time-slots each sending linecard needs to be connected exactly once to each of the  $N$  receiving linecards and vice-versa. This is the classical time-slot assignment problem, known as a Latin square when rates are equal. However, the main difference is an additional constraint which arises from the use of MEMS switches in the switch fabric architecture. Within each time-slot, the rate from each transmitting group of linecards to each receiving group of linecards is limited. Therefore, it is possible that two different linecards in a transmitting group cannot simultaneously send to two different linecards in a receiving group.

An algorithm for constructing the schedule was proposed in [2]. The algorithm constructs three consecutive schedules. First, it creates a schedule between sending groups and receiving groups by repeatedly solving the connection assignment problem defined in Section III. Second, the algorithm creates a schedule between sending linecards and receiving groups. Third, the algorithm creates the final schedule between sending linecards and receiving linecards. These last two steps repeatedly decompose matrices into a minimal number of permutations as defined in Section IV.

In the next two sections, we will formally define the connection assignment problem and the matrix decomposition problem, and then show how they can be efficiently solved in hardware.

## III. CONNECTION ASSIGNMENT PROBLEM

### A. Problem Definition

The configuration algorithm of the load-balanced switch needs to solve the following connection assignment problem. Consider  $2G$  nodes separated into  $G$  left nodes and  $G$  right nodes. The left nodes are connected to the right nodes using

TABLE I

EXAMPLE OF CONNECTION ASSIGNMENT PROBLEM

$$C = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}, RL = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}, RR = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$$

$$\Rightarrow R = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

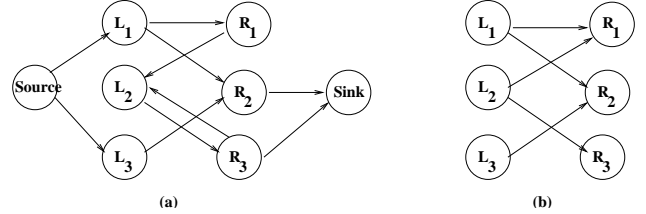


Fig. 1. Ford Fulkerson and Modified Ford Fulkerson

a 0-1 capacity matrix  $C$  of size  $G \times G$ . The rows of the connection matrix correspond to the left nodes, and the columns to the right nodes. We want to find a 0-1 connection matrix  $R$  such that it is below capacity and satisfies a target number of connections per node.  $RL_i$  represents the target number of connections needed for left node  $i$ , and  $RR_j$  similarly represents the target number of connections needed for right node  $j$ .

Table I shows an example of the connection assignment problem with  $G = 3$ . For instance, the first left node in this table needs to make two connections, as specified in the first element of  $RL$ . Similarly, the second right node needs to make one connection, as shown in the second element of  $RR$ . Therefore, the 0-1 solution matrix  $R$  has two elements on its first row, and one element on its second column.

Put mathematically, we want to solve the following problem. Find a 0-1 matrix  $R \leq C$  such that:

$$\begin{cases} \sum_{j'=1}^G R_{ij'} = RL_i & \text{for all } i \\ \sum_{i'=1}^G R_{i'j} = RR_j & \text{for all } j \\ R_{ij} \in \{0, 1\} & \text{for all } i, j \end{cases}$$

Note that the solution is not necessarily unique, and that for the load-balanced switch configuration the capacity matrix will always be sufficiently large to guarantee the existence of a solution [2].

### B. Earlier Work

Given a capacity matrix, it was shown in [2] that the Ford-Fulkerson algorithm can be used to find the solution in polynomial time. The Ford-Fulkerson algorithm consists of finding the augmenting paths from the source node to the sink node until there are no more augmenting paths. The resulting flow is the maximum flow. Either Breadth First Search (BFS) or Depth First Search (DFS) can be used.

Our goal is to implement the algorithm in hardware, and reduce its runtime.

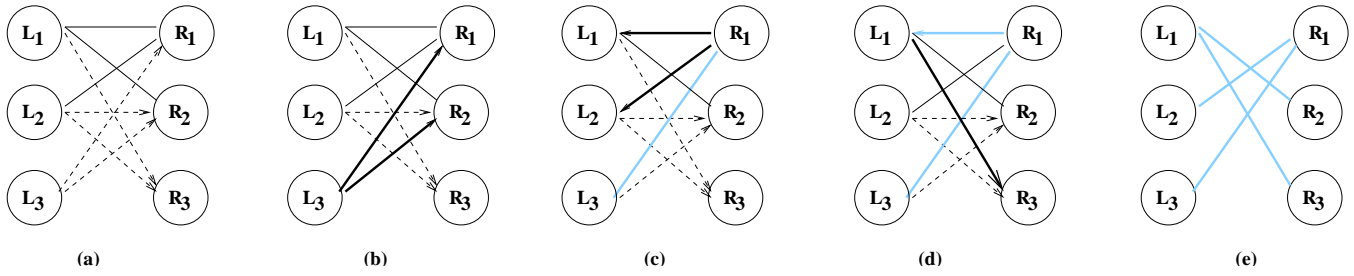


Fig. 2. Back Tracing a) After applying greedy b) Tracing Sending node  $L_3$  c) Tracing  $R_1$  backwards d) Tracing  $L_1$  e) Final Schedule

TABLE II  
RESULT OF THE GREEDY ALGORITHM

$$P = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, RL' = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, RR' = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

$$C' = C - P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

### C. Modified Ford-Fulkerson Algorithm

Each entry in the capacity matrix  $C$  is binary, so  $C$  represents a bipartite graph. Therefore, we can convert the Ford-Fulkerson graph to only consider the left and right nodes. Figure 1 shows the difference between a typical Ford-Fulkerson graph and our modified Ford-Fulkerson graph. The left nodes are named  $L_1, L_2, L_3$  and the right nodes are named  $R_1, R_2, R_3$ . Unlike the original Ford-Fulkerson algorithm, we search from the left nodes to the right nodes, not from the source to the sink. In addition, we do not allow connections from the right nodes to the left nodes.

Our modified Ford-Fulkerson algorithm can be subdivided into two separate parts. The first part uses a greedy approach to make connections between nodes. The second part uses back tracing to find the remaining connections. Let's first explain the greedy part of the algorithm.

**Greedy Algorithm** For each left node, the greedy algorithm keeps adding as many temporary connections as possible to the right nodes. A connection can be added if and only if this connection exists in the binary capacity matrix and the target numbers of left and right connections are not exceeded.

Table II shows the matrix  $P$  of temporary connections after the greedy algorithm is applied.  $RL'$  and  $RR'$  represent the remaining target number of connections to be made for the left and right nodes. Notice that after the greedy algorithm is applied,  $L_1$  is connected to  $R_1$  and  $R_2$ , and  $L_2$  is connected to  $R_1$ . Therefore, the target number of connections for  $L_1, L_2, R_1$ , and  $R_2$  are met. The only connections not yet met are for  $L_3$  and  $R_3$  as seen in  $RL'$  and  $RR'$ . The greedy algorithm cannot connect  $L_3$  to  $R_3$  since the only connections available in the capacity matrix from  $L_3$  are to  $R_1$  and  $R_2$ . After the greedy algorithm is applied, the remaining target connections

specified by  $RL'$  and  $RR'$  are made through the back tracing algorithm. The  $C'$  matrix specifies the connections not used by the greedy algorithm ( $C' = C - P$ ).

### Back Tracing Algorithm

Figure 2 illustrates in our example how the back tracing is done using a simplified version of the BFS algorithm.

Initially the greedy algorithm finds the connections made in the  $P$  matrix, shown by thin solid lines in Figure 2a. These edges are the temporary connections currently made. The connections in the  $C'$  matrix are shown by the dashed lines. In our example  $L_3$  has no connection to  $R_3$ , but has connections to  $R_1$  and  $R_2$ .

This is where the back tracing algorithm starts. Either  $R_1$  or  $R_2$  can be traced back as shown in Figure 2b by thick solid lines. Assume  $R_1$  is traced back. From  $R_1$  we can only trace back to  $L_1$  and  $L_2$ , as shown in Figure 2c. When back tracing from the right nodes, only temporary connections are considered. Assume that  $L_1$  is traced back. Notice in Figure 2d that  $L_1$  has a connection to  $R_3$ , for which the target number of connections is not yet achieved. This ends the trace. Temporary connections are updated and the final connections are shown in Figure 2e.

This back tracing algorithm is repeated for all other nodes that do not achieve their targets.

### Implementation

Memories are used to keep track of the following elements. Throughout both the greedy and back tracing algorithms, we store the current capacity matrix  $C'$ , which keeps track of the temporary connections made, and the remaining number of connections needed to be made to each node. In addition, in the back tracing algorithm, a predecessor memory is needed to remember the trace.

Let's see why the greedy algorithm is memory-intensive. In the greedy algorithm, when connections are added, the algorithm must search for the next available connection to a right node. If there are  $G$  right nodes, this could require up to  $G$  memory accesses per left node, and therefore a total of up to  $G^2$  memory accesses.

We use bit manipulation schemes to reduce the number of memory accesses in the greedy algorithm. We first arrange the current capacity matrix  $C'$  associated with a left node as a bitmap of size  $G$ . We similarly represent the  $RR'$  array as

a bitmap of size  $G$ , where the bit is set if the corresponding  $RR'_j$  is positive. Then, a logical AND between these two bitmaps gives a bitmap representation of the available connections. We can then find the next available connection by finding the first set bit in the resulting bitmap. This can be done in a single clock cycle by using a priority encoder. Therefore, by reusing the resulting bitmap, we can reduce the total number of memory accesses by a factor of up to  $G$ .

Now let's consider why the back tracing algorithm uses many memory accesses. In back tracing we need to keep track of the trace. Since we are using a BFS-based back tracing, each step of the search might require adding up to  $G$  nodes to the predecessor memory. For instance, in one search step of a left node, up to  $G$  right nodes can be considered.

In our implementation we arrange the predecessor memory as a binary matrix of size  $G \times G$ . We implement this matrix by using a memory structure that allows a memory write to an entire row, and a memory read of an entire column. In a search step of the BFS algorithm, instead of writing each node individually, we write the entire set of available nodes in parallel to the entire row. Then, after a trace is done, in order to find the predecessor of a node in the trace, we use an encoder on the entire column of the memory read to find the position of the single bit set in the column. This position corresponds to the index of the predecessor. Using this bit manipulation scheme, we can reduce each search step to a single memory access. Therefore, we reduce the total number of memory accesses by a factor of up to  $G$ .

Therefore, in both the greedy and back tracing algorithms, we can reduce the total number of memory accesses by a factor of up to  $G$  by using bit manipulation schemes and encoders.

#### IV. MATRIX DECOMPOSITION PROBLEM

The configuration algorithm of the load-balanced switch needs to repeatedly decompose matrices into a minimal number of permutations. In this section we'll describe the Birkhoff-von Neumann solution, explain why it is memory-intensive, and then explain why the Slepian-Duguid algorithm leads to a more efficient implementation.

##### A. Problem Definition

Assume that we are given a 0-1 square matrix  $S$  and a positive integer  $n$  satisfying:

$$\begin{cases} \sum_{j'} S_{ij'} = n & \text{for all } i \\ \sum_{i'} S_{i'j} = n & \text{for all } j \\ S_{ij} \in \{0, 1\} & \text{for all } i, j \end{cases}$$

We want to decompose  $S$  into  $n$  permutation matrices, i.e. find  $n$  permutation matrices  $\{P^k\}_{1 \leq k \leq n}$  such that:

$$\begin{cases} \sum_{j'} P_{ij'}^k = 1 & \text{for all } i, k \\ \sum_{i'} P_{i'j}^k = 1 & \text{for all } j, k \\ \sum_{k'} P_{ij}^{k'} = n & \text{for all } i, j \end{cases}$$

Note that although the decomposition is not necessarily unique, it always exists because the chromatic number of a bipartite graph is equal to its maximum degree.

For example, consider the following matrix  $S$ :

$$S = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

A permutation matrix has exactly one 1 in each row and column. Given a matrix  $S$ , the algorithm can generate the following permutation matrices.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Note that in the load-balanced switch example, the binary matrices  $S$  could be of a size up to  $640 \times 640$ , and they are typically sparse, having a maximum of 16 ones in each row and each column.

##### B. Earlier Work

In [2] the polynomial-time Birkhoff-von Neumann decomposition algorithm [9], [10] was proposed. In this decomposition, each permutation can be found by applying a bipartite graph coloring algorithm on the  $S$  matrix (e.g. a maximum size matching algorithm, or the Ford-Fulkerson algorithm). Then, the permutation is removed from the  $S$  matrix. The procedure is repeated until all the  $n$  permutations are found.

In our 100Tb/s router the matrix size can be up to  $640 \times 640$ , requiring large memory structures for the coloring algorithms. Other graph coloring algorithms use smaller data structures and use parallelism [11], [12], [13], [14], [15]. However, none are fast enough for us because they all require at least one occurrence of the maximum size matching algorithm.

##### C. The Slepian-Duguid Algorithm

Instead we use the Slepian-Duguid [16] algorithm designed for scheduling calls in a circuit switch.

First, to reduce the size of the memory, we use the sparsity of the matrices in the load-balanced switch example. In particular, the ones of the binary matrix  $S$  are represented as a list of (row,column) pairs.

Then, to reduce the number of memory accesses, we apply an algorithm based on Slepian-Duguid. This algorithm attempts to produce  $n$  permutation matrices at once, and uses the (row,column) pair list structure. The initial part of our algorithm uses a greedy scheme to assign the easily-matched elements, and the second part uses the Slepian-Duguid algorithm to reassign these elements and provide a solution.

##### Greedy Algorithm

The  $n$  permutations are also organized in a sparse manner, i.e. by using a (row,column) pair list structure. For clarity, we will refer to rows as inputs and to columns as outputs. Each

TABLE III  
GREEDY ALGORITHM FOR SLEPIAN-DUGUID

$$A_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, A_4 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 \end{pmatrix}, A_8 = \begin{pmatrix} 1 & 3 & 2 & 0 & 0 \\ 3 & 1 & 4 & 0 & 0 \\ 4 & 5 & 0 & 0 & 0 \end{pmatrix}, A_{15} = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 0 & 1 & 2 \end{pmatrix}$$

TABLE IV  
BACK TRACING FOR SLEPIAN-DUGUID

$$B_0 = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 0 & 1 & 2 \end{pmatrix}, B_1 = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 5 & 1 & 2 \end{pmatrix}, B_2 = \begin{pmatrix} 1 & 5 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 3 & 5 & 1 & 2 \end{pmatrix}, B_3 = \begin{pmatrix} 1 & 5 & 2 & 3 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 3 & 5 & 1 & 2 \end{pmatrix}$$

permutation is arranged as an array of outputs. For instance, the  $i$ -th element in the array refers to the output that is matched with input  $i$ . Note that a valid permutation will not match more than one output to the same input. Since we want to find  $n$  permutations, we maintain  $n$  such arrays. We arrange these arrays into a matrix  $A$  where each row corresponds to a different permutation and therefore to a different array.

In the greedy algorithm, the matrix  $A$  is initially empty. Then, the algorithm goes through the list of (input,output) pairs, denoted  $(i, o)$ , and tries to assign each such pair to a permutation for which input  $i$  and output  $o$  are both unassigned. This continues until no more  $(i, o)$  pairs can be assigned.

Table III illustrates how the greedy algorithm works. The matrix  $A_k$  shows the state of the permutations after the  $k$ -th  $(i, o)$  pair is assigned. For instance, if  $o$  is the  $(p, i)$ -th element of  $A_k$ , then the  $p$ -th permutation matches input  $i$  to output  $o$  in the  $k$ -th step. Since matrix  $A_0$  is initially empty, each element in the matrix is set to 0. Let's now look at how the  $(2, 1)$  pair is assigned in the 4-th step. In  $A_4$ , the  $(2, 1)$  pair can only be assigned to the second or third permutation since output 1 is already scheduled in the first permutation. Let's assume that it is assigned to the second permutation. It is possible that an  $(i, o)$  pair can not be assigned. For instance, in  $A_8$ , the  $(3, 5)$  pair cannot be assigned since the only permutation free for input 3 is the third permutation, and output 5 is already assigned in the third permutation.  $A_{15}$  in Table III shows the final state of the permutations after the greedy algorithm. Notice that the  $(3, 5)$  and  $(4, 3)$  pairs are not yet assigned and need to be assigned in the Slepian-Duguid algorithm.

### Slepian-Duguid Algorithm

For each (input,output) pair  $(i^1, o^1)$  that is left unassigned, the algorithm works as follows:

- 1) Identify the permutations  $P_{i^1}$  and  $P_{o^1}$  such that input  $i^1$  is not assigned in  $P_{i^1}$  and output  $o^1$  is not assigned in  $P_{o^1}$ .
- 2) Swap the input  $i^1$  with  $i^2$ , where  $i^2$  is an input such that  $(i^2, o^1)$  was already assigned in  $P_{i^1}$ . Now we need to track  $(i^2, o^1)$ .
- 3) Swap the output  $o^1$  with  $o^2$ , where  $o^2$  is an output such

that  $(i^2, o^2)$  was already assigned in  $P_{o^1}$ . Now we need to track  $(i^2, o^2)$ .

Repeat steps 2 and 3 until we have unassigned slot for  $(i^n, o^n)$  in either of the permutations  $P_{i^1}$  or  $P_{o^1}$ .

The  $B_i$  matrices are similar to the  $A_i$  matrices. Table IV extends the above example to explain how the  $(3, 5)$  pair is assigned. Let's consider the  $(3, 5)$  pair first. Initially we start with identifying the permutations which have input 3 and output 5 free. As can be seen in  $B_0$ ,  $P_3$  (row 3) has input 3 free and  $P_1$  (row 1) has output 5 free. So we assign  $(3, 5)$  in  $P_3$  as shown in  $B_1$ . Now as can be observed in  $B_1$  there are two inputs assigned to output 5 in  $P_3$ . We swap output 5 of  $P_3$  with output 3 of  $P_1$  for the same input.  $B_2$  shows the resulting matrix. Since output 5 is only assigned once in  $P_1$ , the algorithm stops here. Otherwise, it repeats the procedure of swapping until no permutation has multiple outputs assigned.  $B_3$  shows the final resulting matrix.

### Implementation

Memories are used to keep track of which inputs and outputs are unassigned after each permutation. For each input  $i$ , we store the  $n$  permutations in a bitmap of size  $n$ . If input  $i$  is not assigned in a given permutation, we set the bit corresponding to this permutation in the bitmap. The same is done for each output. Then, in the greedy algorithm, an (input,output) pair can easily find a free permutation by taking a logical AND between the input and output bitmaps. By finding the first set bit in the resulting bitmap, the (input,output) pair can be matched to a free permutation in a single clock cycle. Therefore, by reusing the resulting bitmap, we can reduce the total number of memory accesses by a factor of up to  $n$ .

## V. RESULTS

The algorithms have been implemented in hardware and the results are presented here.

### A. Synthesis

The hardware implementations mentioned in the previous section are implemented using Verilog and synthesized using a 0.13um process. In this implementation for 40 groups, up to 640 linecards and a maximum of 16 linecards per group, the

modified Ford-Fulkerson algorithm uses 10K gates, 24Kbits of memory, whereas the core for the Slepian-Duguid algorithm uses 25K gates, 230Kbits of memory. Based on the synthesis results, both the cores ran within a 4ns clock cycle time.

### B. Simulations

Because of the number of experiments we wanted to run, and the complexity of the algorithm, running our Verilog implementation was too slow. Instead, we developed a cycle-accurate C-model and verified its accuracy by comparing it with the Verilog implementation. We then used the C-model to run a large number of tests. The C-model reports the number of clock cycles the simulation would run for in Verilog. Because of the constraints in the load-balanced switch [1], we generate different sets of results over 1000 iterations with different ranges of linecards between 0 and 640, spread over  $G = 40$  groups. Linecards are randomly spread across the 40 groups where the maximum number of linecards in a group is 16. For each range, the worst case running time of the simulations is obtained.

A processor is assumed to be connected to the cores uploading and downloading the necessary information into the memories. The times to transfer the initial matrices and to obtain the final results to/from the processor are not considered in the results. We compare the simple conversion to hardware of existing algorithms with our implementation using the total number of memory accesses. We assume that memories can be accessed in a pipelined manner and that each memory access requires one clock cycle. We measure the number of hardware clock cycles needed. Figure 3 shows the time spent for the simple conversion and our implementation assuming a 4 ns clock cycle time. The graph plots the largest number of clock cycles needed, in any of the tests we ran. We use a logarithmic scale so as to represent both plots on the same graph. Since the algorithms are polynomial, the plots appear logarithmic. Note that we did not attempt to pipeline the Verilog implementation and believe that we can reduce the time by an additional factor of at least two. Even without complete pipelining, our results show that our implementation meets the 50ms target over the range of linecards needed in [1].

## VI. CONCLUSIONS

This paper implements the configuration algorithms of the load-balanced switch introduced in [2]. The implementation meets the 50ms recovery time imposed by network operators.

Our hardware implementation relies on bitmap manipulation schemes with priority encoders to drastically reduce the memory intensive operations. Further improvements can be achieved by pipelining, using multiport memories and by exploiting some of the parallelism in the greedy parts of the algorithms. We believe that these schemes can be generalized to accelerate hardware implementations of other graph coloring algorithms.

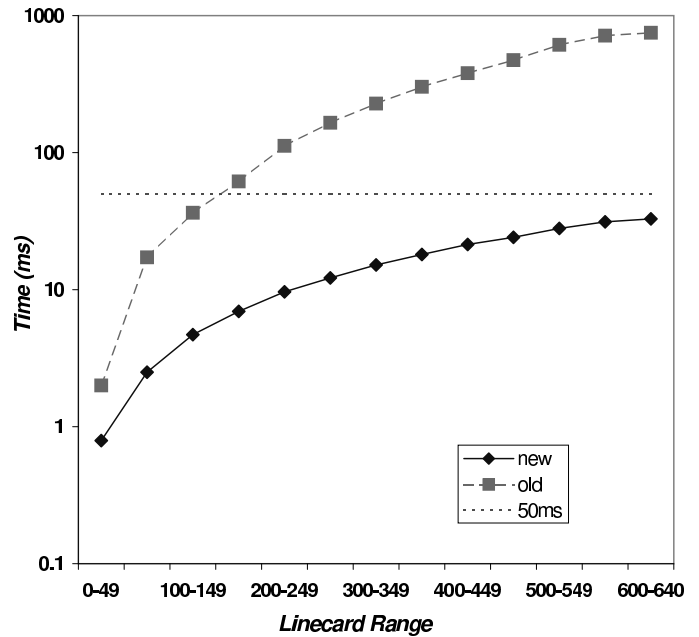


Fig. 3. Worst-case Running Time of Simple Hardware Conversion and Hardware-Specific Implementations

## REFERENCES

- [1] I. Keslassy, S.-T. Chuang, K. Yu, D. Miller, M. Horowitz, O. Solgaard, N. McKeown, "Scaling Internet routers using optics," *ACM SIGCOMM 2003*, Karlsruhe, Germany, Sept. 2003.
- [2] I. Keslassy, S.-T. Chuang, N. McKeown, "A load-balanced switch with an arbitrary number of linecards," *Proceedings of IEEE Infocom '04*, Hong Kong, March 2004.
- [3] C.-S. Chang, D.-S. Lee and Y.-S. Jou, "Load balanced Birkhoff-von Neumann switches, part I: one-stage buffering," *IEEE HPSR '01*, Dallas, May 2001.
- [4] Telcordia, GR-499 CORE, "Transport systems generic requirements (TSGR): common requirements criteria," Issue 2, Dec. 1998.
- [5] Telcordia, GR-253 CORE, "Synchronous optical network (SONET) transport systems: common generic criteria," Issue 3, Sept. 2000.
- [6] ANSI (American National Standards Institute), T1.TR.68-2001, "Enhanced network survivability performance," February 2001.
- [7] ITU-T (International Telecommunication Union Standardization Sector), Recommendation G.841, "Types and characteristics of SDH network protection architectures," July 1995.
- [8] L.R. Ford and D.R. Fulkerson, "Flows in Networks", Princeton University Press, 1962.
- [9] C.S. Chang, J.W. Chen, and H.Y. Huang, "On service guarantees for input-buffered crossbar switches: a capacity decomposition approach by Birkhoff and Von Neumann," *IEEE IWQoS, London*, 1999.
- [10] G. D. Birkhoff, "Tres observaciones sobre el algebra lineal," *Universidad Nacional de Tucuman Revista, Serie A*, vol. 5, pp. 147-151, 1946.
- [11] R. Cole, K. Ost and S. Schirra, "Edge-coloring bipartite multigraphs in  $O(E \log D)$  time," *Combinatorica*, vol. 21, pp. 5-12, 2001.
- [12] R. Cole, K. Ost and S. Schirra, "Edge-coloring bipartite multigraphs in  $O(E \log D)$  time," *New York University Technical Report NYU-TR1999-792*, New York, Sep. 1999.
- [13] N. Alon, "A simple algorithm for edge-coloring bipartite multigraphs," *Information Processing Letters*, vol. 85, issue 6, pp. 301-302, March 2003.
- [14] A. Schrijver, "Bipartite edge-coloring in  $O(\Delta m)$  time," *SIAM J. Comput.*, vol. 28, pp. 841-846, 1999.
- [15] A. Schrijver, "A course in combinatorial optimization," available at <http://www.cwi.nl/~lex/files/dict.ps>, Feb. 2003.
- [16] J. Hui, *Switching and Traffic Theory for Integrated Broadband Networks*, Kluwer Academic Publishers, Boston, 1990.