CHAPTER 6
IMPLEMENTATION OF A CONFIGURATION ALGORITHM FOR THE LOAD-BALANCED SWITCH

6.1 Introduction

The demand for network capacity continues to increase every year. In the recent past, routing capacity per unit volume has doubled every eighteen months (Fig.6.1). Internet providers are hence, required to double the number of routers in their network each year, which is difficult since it would require large capital investments and increases in the support and maintenance infrastructure. Further, additional routers are needed to interconnect other routers in the enlarged topology. So, it would take more than twice as many routers to carry twice the user traffic to achieve the same link utilization and throughput. Instead, it seems reasonable to periodically replace existing routers with newer higher capacity systems [13-16].

Fig. 6.1 shows the growth in router capacity over time [13]. Each point represents a commercial router (For e.g., Cisco routers). Each generation of routers consumes more power than the previous, and it is now difficult to package a router in one rack of equipment. Network operators can supply and dissipate about 10 kW per rack, and single-rack routers have reached this limit. There has therefore been a move towards multi rack systems. Existing multi-rack systems suffer from two main problems: unpredictable performance, and poor scalability. So router architectures with predictable throughput and scalable capacity have to be identified.

The load-balanced router is a very promising architecture introduced recently [67, 71-73]. It has been proved to provide 100% throughput. It is scalable, requires no centralized scheduler, and is amenable to optics. It simplifies the switch fabric, replacing a frequently scheduled and reconfigured switch with two identical switches that follow a fixed sequence.
This chapter is the report on the work done in configuration algorithms for line card scheduling in the load balanced switch architecture. The algorithm is carried out in three stages and includes double back-tracing for improved performance.

### 6.1.1 Conventional Router and the Load Balanced Router

Typical router architectures with centralized schedulers have high packet processing complexity since the scheduler has to configure the switch fabric every timeslot. The load balanced router architecture needs no centralized scheduler. The schedule matrix is predetermined using algorithms like the TDM algorithm [93,94]. So, the switch fabric need not be reconfigured for each time slot. A basic load-balanced switch is shown in Figure 6.2. It consists of a single stage of N buffers, sandwiched between two identical stages of switching, where each switch is built from a uniform mesh.

Fig. 6.1 The growth in router capacity over time [13]
The Load balanced switch architecture

As shown in the figure, the architecture is based on two fully interconnected meshes with N line cards interconnected by N^2 links. Each line card in the first stage is connected to each line card in the center stage by a channel at rate R/N, where R is the line rate and N is the number of line cards. Likewise, each line card in the center stage is connected to each line card in the final stage by a channel at rate R/N. The buffer stage is partitioned into N separate FIFO queues, one per output (Virtual Output Queues). There are a total of N^2 VOQs in the switch.

6.1.2 Operation of the Load Balanced Switch

The working of the two switch fabrics is different from a single stage packet switch. Instead of picking a switch configuration based on the occupancy of the queues, both switching stages walk through a fixed sequence of configurations. At time t, input ‘i’ of each switch fabric is connected to output [(i + t) mod N] + 1, i.e., each input is connected to each output exactly (1/N)th of the time, irrespective of the arriving traffic.
Each stage is a fixed, equal-rate switch. Although they are identical, the two stages perform different functions. The first stage is a load-balancer that spreads traffic over all the VOQs. The second stage is an input-queued crossbar switch in which each VOQ is served at a fixed rate. When a packet arrives at the first stage, it is immediately transferred to an intermediate input, which depends on the current configuration of the load-balancer. In the intermediate input, the packet is stored in a VOQ according to its final output. Sometime later, the second fixed equal-rate switch will serve the VOQ. The packet will then be transferred across the second switch to its output, from where it will leave the system. Thus, the two meshes work identically, but perform two different functions. The first one load-balances packets across the center stages, sending 1/Nth of the traffic to each intermediate input, and the second one switches packets to their correct destination by servicing each VOQ at fixed rate R/N.

6.1.3 The Load-Balanced Router-Line Card Architecture

Each line card has three parts: An input block, an output block, and an intermediate input block shown in Fig. 6.3. Arriving variable length packets are segmented into fixed sized packets and then transferred to the eventual output, where they are reassembled into variable length packets again. The input block performs address lookup, segments the variable length packet into one or more fixed length packets, and then forwards the packet to the switch. The intermediate input block accepts packets from the switch and stores them in the appropriate VOQ. It takes packets from the head of each VOQ at rate R/N and sends them to the switch to be transferred to the output. Finally, the output block accepts packets from the switch, collects them together, reassembles them into variable length packets, and delivers them to the external line.
It is also assumed that all line cards have the same line rate and that time is slotted, so that at the most one packet can arrive at any input port and at most one packet can depart from any output port in each time-slot.

6.1.4 Throughput Guarantee in a Load Balanced Switch

In a conventional single-stage crossbar switch, throughput guarantees are only possible if a scheduler configures the switch based on the knowledge of the states of all the VOQs or of all the traffic rates. In the load-balanced switch, the sequence of switch configurations is predetermined, regardless of the state of the VOQs or the traffic rates. For a single, fixed, equal-rate crossbar switch with uniform destinations, it is clear that each VOQ is served at rate $R/N$. With arrival rate $\lambda < R/N$ and service rate $\mu = R/N$, the system is stable (the queues will not grow without bound), and hence 100% throughput is
guaranteed. If arrivals are uniform, a fixed, equal-rate switch, with VOQs, has a guaranteed throughput of 100% [70]. Since real network traffic is not uniform, an extra load-balancing stage can spread out non-uniform traffic, making it sufficiently uniform to achieve 100% throughput. If arrivals and services of a work-conserving switch are stationary and ergodic, and at any time the expected number of arrivals is less than the expected number of services, then the switch has 100% throughput. The exhaustive mathematical proof is given in [13,94].

6.1.5 Advantages of the Load Balanced Switch Architecture

A basic load-balanced router uses no centralized scheduler. This is different from the most common current router architectures, which use a centralized arbiter that computes a match between the inputs and the outputs for each time-slot. A basic load-balanced router does not need any scheduling, since the sequence of crossbar permutations is predetermined.

Since there is no centralized scheduler, a basic load-balanced router does not need any exchange of VOQ-state and scheduling-decision information between the line cards and a centralized scheduler. This simplifies the switch control and removes the loss of bandwidth due to the exchange of information.

A basic load-balanced router also simplifies the switch fabric. It uses only N states for the switch fabric out of the possible N! states. The state sequence is predetermined, and not computed at each time slot.

A load-balanced router simplifies the line cards, especially the buffering stage. It only uses one stage of buffering, while typical centralized-scheduler switches run the switch fabric faster than the line rate and require two stages of buffering (at the inputs and at the outputs).
A basic load-balanced router also requires less buffering per line card because it spreads long bursts across the intermediate inputs. All these properties suggest that it might be possible to scale a basic load-balanced router more easily than traditional architectures.

In Section 6.2, the missing line card problem is stated. Section 6.3 discusses the proposed scheduling algorithm for line card configuration, which is carried out in 3 stages. Section 6.4 presents software simulation and hardware implementation results and discussion.

6.2 The Missing Line Card Problem

6.2.1 Problem Definition

Designing a router based on the load-balanced switch is challenging since non-uniform placement of line cards have to be supported. If all the line cards were always present and working, they could be interconnected by a uniform mesh of fibres. But if some line cards are missing, or have failed, the switch fabric needs to be reconfigured so as to spread the traffic uniformly over the remaining line cards. To illustrate the problem, let there be a situation where all but two line cards are removed from a load-balanced switch based on a uniform mesh. When all line cards were present, the input line cards spread data over N center-stage line cards, at a rate of $2R/N$ to each. With only two remaining line cards, the traffic must spread over both line cards, increasing the rate to $2R/2 = R$. This means that the switch fabric must now be able to interconnect line cards over a range of rates from $2R/N$ to $R$. The need to support an arbitrary number of line cards is a problem for network operators who want the flexibility to add and remove line cards when needed. Some kind of reconfigurable switch is needed to spread the traffic uniformly over the line cards that are present.
6.2.2 Solution for the Missing Line Card Problem

The architecture is arranged as \( G \) groups of \( L \) line cards. In the center, \( M \) statically configured (GxG) MEMS switches interconnect the \( G \) groups. The MEMS switches are reconfigured only when a line card is added or removed. This is shown in Figure 6.4. Each group of line cards spreads packets over the MEMS switches using an (LxM) electronic crossbar. Each output of the electronic crossbar is connected to a different MEMS switch over a dedicated fiber at a fixed wavelength (the lasers are not tunable). Packets from the MEMS switches are spread across the \( L \) line cards in a group by an (MxL) electronic crossbar. Time Division Multiplexing algorithms [94] can be used to configure the MEMS switches and schedule the crossbar. In the next section, a
scheduling algorithm based on TDM, used for implementing the scheduling of the line cards at the sending and receiving ends is presented.

6.3 The Scheduling Algorithm

- Let ‘t’ be the number of time-slots left to schedule after every iteration.
- Let ‘N’ be the total number of line cards.
- Let the line cards be divided into G groups; group i contains $L_i$ line cards, and the total number of line cards is given by

$$N = \sum_{i=0}^{G-1} L_i \tag{6.1}$$

- It is assumed that $L_1, L_2, \ldots, L_G$ are fixed for a given line card arrangement.

6.3.1 Rules for the Scheduling Algorithm

1) In a schedule of $N$ time-slots, each transmitting line card sends exactly one packet to each receiving line card. If sending line card ‘$i$’ is connected to receiving line card $T_{ij}$ in time-slot ‘$j$’, then, $T_{ij} \neq T_{ij'}$ for all $j' \neq j$

$T_{ij} \neq T_{ij'}$ for all $i' \neq i$

$T_{ij} \in \{1...N\}$ for all $i,j$.

where, ‘$T$’ is the line card schedule. $T$ is a Latin square, i.e. the numbers 1 to N appear exactly once in every row and every column.

2) In the final scheduled matrix, no line card of the sending group should be scheduled to two or more line cards of the receiving group in a particular timeslot and vice versa.

3) For full capacity utilization, all the timeslots must be utilized which means the final schedule matrix should not contain any zeros.

4) The MEMS constraint [94] must be satisfied. Let $L_i$ represent the number of line cards in group $i$. The rate needed between group $i$ and group $j$ is equal to

$$\frac{L_i \cdot 2R \cdot L_j}{N},$$

\[91\]
where $1 \leq i, j \leq G$ and $R$ is line rate. This is because the incoming traffic is spread uniformly over all $N$ receiving line cards, and group $j$ receives a portion $(L_j/N)$ of the traffic. As assumed above, two groups can only communicate at a rate up to $2R$ through any single MEMS switch. Therefore, the minimum number of MEMS switches between group $i$ and group $j$ is:

\[
\left[ \frac{L_i \cdot 2R \cdot L_j}{N} \right] + \left[ \frac{L_i \cdot L_j}{N} \right] = \left[ \frac{L_i \cdot L_j}{2R} \right]
\]

This is called as the MEMS constraint. The final schedule matrix must satisfy this MEMS constraint. For instance if $L_1 = 3$, $L_2 = 2$, $L_3 = 2$ then the maximum number of connections allowed between group $i$ and group $j$ at any timeslot is the remainder of $(3 \times 3/7) = 2$.

The algorithm has 3 phases: Group-Group scheduling, Line card-Group scheduling and Line card-Line card scheduling, which are discussed in the next sections.

### 6.3.2 Group – Group Scheduling

A Group-to-Group (GG) schedule $V$ is a matrix with $G$ rows corresponding to the $G$ sending linecard groups, $N$ columns corresponding to the $N$ time-slots of the frame, and $L_i$ letters per row-column intersection in row $i$. A G-G schedule $V$ is said to be valid iff the $i^{th}$ letter appears exactly $L_i \cdot L_j$ times in each row $j$ (corresponding to sending group $j$), $L_i$ times in each column, and at most $(L_i \cdot L_j \mod N)$ times in any row-column intersection in row $i$ (MEMS constraint). GG scheduling is done using the Latin Square Algorithm. The algorithm recursively builds a valid Group - Group schedule time-slot after time-slot, for the $N$ timeslots of the frame. It gives a $(G \times N)$ matrix from which a valid Linecard – Group schedule matrix can be constructed [93].
Table 6.1 Example of application of Latin square algorithm

<table>
<thead>
<tr>
<th>Total no. of linecards</th>
<th>N: 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total no. of Groups</td>
<td>G: 3</td>
</tr>
<tr>
<td>No. of Line cards in each Group</td>
<td>L1: 3; L2: 2; L3: 2</td>
</tr>
</tbody>
</table>

\[
M^7 = \begin{pmatrix} 9 & 6 & 6 \\ 6 & 4 & 4 \\ 6 & 4 & 4 \end{pmatrix} \quad P^7 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q^7 = \begin{pmatrix} 2 & 6 & 6 \\ 6 & 4 & 4 \\ 6 & 4 & 4 \end{pmatrix} \quad R^7 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad S^7 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\]

\[
M^6 = \begin{pmatrix} 8 & 5 & 5 \\ 5 & 3 & 4 \\ 5 & 4 & 3 \end{pmatrix} \quad P^6 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q^6 = \begin{pmatrix} 2 & 5 & 5 \\ 5 & 3 & 4 \\ 5 & 4 & 3 \end{pmatrix} \quad R^6 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad S^6 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\]

\[
M^5 = \begin{pmatrix} 7 & 4 & 4 \\ 4 & 2 & 4 \\ 4 & 4 & 2 \end{pmatrix} \quad P^5 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad Q^5 = \begin{pmatrix} 2 & 4 & 4 \\ 4 & 2 & 4 \\ 4 & 4 & 2 \end{pmatrix} \quad R^5 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad S^5 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\]

\[
M^4 = \begin{pmatrix} 6 & 3 & 3 \\ 3 & 1 & 4 \\ 3 & 4 & 1 \end{pmatrix} \quad P^4 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad Q^4 = \begin{pmatrix} 2 & 3 & 3 \\ 3 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \quad R^4 = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad S^4 = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}
\]

\[
M^3 = \begin{pmatrix} 4 & 2 & 3 \\ 2 & 1 & 3 \\ 3 & 3 & 0 \end{pmatrix} \quad P^3 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \quad Q^3 = \begin{pmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad R^3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad S^3 = \begin{pmatrix} 2 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}
\]

\[
M^2 = \begin{pmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{pmatrix} \quad P^2 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \quad Q^2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad R^2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad S^2 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\]

\[
M^1 = \begin{pmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{pmatrix} \quad P^1 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \quad Q^1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad R^1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad S^1 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\]

6.3.2.1 Steps for Latin Square Algorithm

At the start, \( t = N \), since all the time-slots are unscheduled. Also, let \( M = M^t = M^N \) be the initial matrix of all the elements that need to be scheduled. Its rows represent the sending groups, its columns the receiving groups (letters "A", "B" ...). For all \( i, j \), \( M_{ij} = \)
L_j*L_j, i.e. there are (L_j*L_j) connections to schedule from sending group i to receiving group j during the whole frame. For t = N, N - 1... 1, the steps to be done iteratively are as follows:

1) For each i, j, M_{ij}^t is decomposed in base t:

\[ M_{ij}^t = P_{ij}^t \cdot t + Q_{ij}^t \] (i.e. P_i^t = M_i^t/t, Q_i^t = M_i^t - P_i^t \cdot t)

First P_i^t is scheduled, then the remainder Q_i^t is considered and a part of it is scheduled such that all the constraints are satisfied.

2) The vectors a_i^t and b_j^t (integer vectors) are defined such that

\[ a_i^t = \frac{\sum_{j=1}^{G} Q_{ij}^t}{t} \quad \text{for all } i; \quad b_j^t = \frac{\sum_{i'=1}^{G} Q_{i'j}^t}{t} \quad \text{for all } j; \]

3) A 0-1 matrix R_i^t < Q_i^t is found such that R_{ij}^t \in \{0,1\}

\[ \sum_{j'=1}^{G} R_{ij'}^t = a_i^t \quad \text{for all } i; \quad \sum_{i'=1}^{G} R_{i'j}^t = b_j^t \quad \text{for all } j \]

R_i^t can be built using Ford-Fulkerson max-flow algorithm.

4) Using the schedule S_i^t = P_i^t + R_i^t for the time-slot t, M_{i=1}^{t-1} is updated to M_{i=1}^{t-1} = M_i^t - S_i^t.

Table 6.1 gives an example for the application of the Latin square algorithm.

6.3.2.2 Flow Matrix Construction

The flow matrix R can be constructed using the modified Ford Fulkerson algorithm [94], which is a simpler version of the original Ford Fulkerson algorithm.

R_i^t is a 0-1 matrix, R_i^t \leq Q_i^t. Let C be the capacity matrix \leq Q_i^t such that \{C_{i,j}^t = 1, \text{ if } Q_{i,j}^t > 0 \text{ for all } i,j\}. The capacity matrix shows the connections that are possible between groups. Each group has a fixed sending rate and receiving rate. With reference to Table 94...
6.2, the RL matrix (rate left as the sending groups are visualized as being on the left) indicates the sending rates of the groups and the RR matrix (rate right as the receiving groups are visualized as being on the right) indicates the receiving rates of the groups. For the example given in the Table 6.2, Group 1 has a sending rate of 2 and hence can make 2 connections to any of the groups. Group 1 has a receiving rate of 2 and hence can take 2 connections from any of the groups. R is the final schedule that needs to be obtained. A 1 at (i,j) in R signifies a connection between Group i and Group j. As can be observed, the number of 1's in row 1 of R is equal to 2 and the number of 1's in column 1 of R is equal to 2 corresponding to the RL and RR of group 1. A 0-1 capacity matrix was considered as it is easier to implement in hardware. Ford-Fulkerson algorithm was used for this purpose.

6.3.2.3 The Ford-Fulkerson and Modified Ford-Fulkerson Algorithm

Ford-Fulkerson algorithm is based on finding the augmenting paths from the source node to the sink node and this is done until there are no more augmenting paths. The resulting flow is the maximum flow. Breadth First Search (BFS) or Depth First Search (DFS) can be used to find the augmenting paths from the source ‘S’ to sink ‘T’. However, since each entry in C is either a 1 or a 0 and this is a bi-partite graph, the algorithm can be modified to make it simpler. Figure 6.5 shows the difference between the Ford-Fulkerson implementation and the modified Ford-Fulkerson implementation. In this algorithm there are multiple sources (nodes on the left side of the graph) and sinks (nodes on the right side of the graph). Each source has a specified rate that needs to be
fulfilled (RL in the example) and each sink has a specified rate that it can handle (RR in the example). Rates are the connections that the group (node) needs to have. The first part of the algorithm uses a greedy approach to schedule the groups. The later part uses backtracing (simple version of BFS) to find the remaining schedule. For this algorithm there are G nodes on the left (source nodes) and G on the right (sink nodes).

i) Greedy

Table 6.3 shows how the algorithm works for the capacity and rate matrices given in Table 6.2. In the example G = 3, and the groups are named as A, B, C for reference. Matrix P shows the partial schedule obtained after the greedy algorithm is applied. C₁ is the capacity matrix, F₁ is the flow matrix, RL₁ and RR₁ are the rates of the left and right nodes after applying the greedy algorithm. A flow matrix keeps track of the connections already made. Combining C₁ with F₁ at anytime gives the original capacity matrix C. The final flow matrix is the R matrix. The greedy algorithm works as follows

1. Step through each sending group until the rate goes to 0.

2. For each group check whether there is a capacity element that points to a group whose rate right is non-zero. If there is, then move the element to flow matrix.

3. Reduce the rate left of the sending group and rate right of the receiving group.
Table 6.2 Example for Scheduling using the modified Ford Fulkerson Algorithm

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>RL</th>
<th>RR</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.3 Example- Application of the modified Ford Fulkerson Algorithm

<table>
<thead>
<tr>
<th>RL₁</th>
<th>RR₁</th>
<th>C₁</th>
<th>F₁</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In the example, sending group A has a rate left of 2 and receiving group A has a rate right of 2. When a connection (element 1, 1 of the C matrix) is made from sending group A to receiving group A, the RL and RR of group A are reduced to 1. The element (1, 1) in the capacity matrix is made 0 and the element (1, 1) in the flow matrix is made 1. For the example shown, Group A on the left is assigned to Group A and Group B on the right. Group B on the left is scheduled with Group A on the right, making the RL of both Group A and Group B as zero. Group C on the left does not have a path to Group C on the right and Group C is the only node on the right that has a positive RR. Hence Group C cannot be scheduled. So, back-tracing is applied to schedule Group C.

ii) Back Tracing

As shown in Table 6.3, RL₁ and RR₁ are non zero and back-tracing needs to be applied to find the remaining schedule. In this example Group C has a non-zero RL, but has no connection to the non-zero RR element (receiving group C) in the capacity matrix. Hence, back-tracing needs to be done.
Fig. 6.6 Back tracing to find GG Schedule  

a) After applying greedy  
b) Tracing sending group C  
c) Tracing receiving group A backwards  
d) Selecting sending group A  
e) Final schedule  

Fig. 6.6 shows how the back tracing is done using the BFS algorithm. The greedy algorithm finds paths $A \rightarrow A$, $A \rightarrow B$, $B \rightarrow A$ (thin solid lines in Fig. 6.6 (a)). These paths form the flow matrix. The dashed lines show the remaining part of the capacity matrix. In this case C needs to be scheduled, the two groups on the right it has a path to are A and B whose rates are 0. Either A or B can be traced back as shown in Fig. 6.6 (b) by thick solid lines. $C \rightarrow A$ is traced back, since A has a path to C whose rate is non-zero. The final paths are shown in Fig. 6.6 (e). The predecessors of the nodes traced are kept track of, so that the capacity and flow matrices can be updated, once the BFS finds an augmenting path.

6.3.2.4 Constructing the GG Schedule Matrix

The GG schedule matrix denoted by $V$ is constructed from the schedules, $S_i$, given in Table 6.1. $V$ is a $(G \times N)$ matrix with each row having $L_i$ elements in each column where $L_i$ represents the number of line cards in $i^{th}$ group. Here 'i' represents the row.
index. \( S_{ij} \) represents the number of occurrences of the \( j \)th letter in the \( i \)th row in column \((N - t + 1)\) of matrix \( V \).

For example, considering the \( S^7 \) matrix,

\[
S^7 = \begin{pmatrix}
1 & 1 & 1 \\
1 & 1 & 0 \\
1 & 0 & 1
\end{pmatrix}
\]

From this matrix the first column of \( V \) can be created having one A, one B and one C in the first row; one A and one B in the second row; and one A and one C in the third row, i.e., column 1 of \( S^7 \) represents group A, column 2 represents group B and column 3 represents group C. Moving on to \( S^6 \) and up to \( S^1 \), columns 2 to \( N \) of \( V \) matrix can be constructed. The final GG schedule matrix obtained is:

\[
V = \begin{pmatrix}
ABC & ABC & ABC & AAB & AAC & ABC & ABC \\
AB & AB & AB & AC & BC & AC & AC \\
AC & AC & AC & BC & AB & AB & AB
\end{pmatrix}
\]

From this matrix it is inferred that in the first timeslot, group A denoted by row 1 is connected to 3 line cards of groups A, B, C. Group B denoted by row 2 is connected to 2 line cards of groups A, B and Group C denoted by row 3 to 2 line cards of groups A, C. Likewise connections can be inferred for all remaining \((N - 1)\) timeslots. From this Group -Group schedule matrix, Line card-Group schedule matrix can be obtained which identifies the sender line card and the receiver group. In the proposed algorithm, the \( V \) matrix is skipped and the Line card – Group schedule is found directly from the \( S^i \) matrices.

6.3.3 Line Card – Group Scheduling

A line card-to-group (L-G) schedule \( U \) is a matrix with \( N \) rows corresponding to the \( N \) sending line cards, \( N \) columns corresponding to the \( N \) timeslots of the frame, and one letter per row-column intersection corresponding to the receiving group. An L-G schedule \( U \) is said to be valid iff the \( i \)th letter appears exactly \( L_i \) times in each row and
each column, and at most \((L_i \cdot L_j) \mod N\) times in the linecards of group \(i\) in any column of \(U\) (MEMS constraint).

### 6.3.3.1 Manipulation of the \(V\) Matrix

For each \(1 < j < G\), row \(j\) in \(V\) may be considered. In the example cited, the first row of \(V\) is: \((ABCABC \ ABC \ AAB \ AAC \ ABC \ ABC)\)

Each row \(j\) must be subdivided into \(L_j\) sub-rows, corresponding to the subdivision of each sending group \(j\), into \(L_j\) sending linecards, thus forming a valid \(L\)-G schedule. First, each letter has \(L_i \cdot L_j\) occurrences in any given row of \(V\). Arbitrarily they are divided into \(L_i\) subscripted letters ("sub-letters") of \(L_j\) elements. In the given example, the letters representing the three groups \(A\), \(B\), \(C\) of \(V\) are transformed into \(N\) arbitrarily assigned sub-letters \((A_1, A_2, A_3, B_1, B_2, C_1, C_2)\). For instance, since \(A\) appears 9 times in the first row, the \(A\)'s are arbitrarily replaced with 3 \(A_1\)'s, 3 \(A_2\)'s and 3 \(A_3\)'s:

\((A_1B_1C_1 \ A_1B_1C_1 \ A_1B_1C_1 \ A_2A_2B_2 \ A_2A_3C_2 \ A_3B_2C_2 \ A_3B_2C_2)\)

In row \(j\) of matrix \(V\), each of the \(N\) sub-letters has \(L_j\) occurrences, and each of the \(N\) columns has \(L_j\) elements. A new matrix, say \(T\), that has sub-letters as inputs and columns as outputs is formed. There will be \(G\), \(T\) matrices. In this \(T\) matrix, all columns and all rows have a sum of \(L_j\), where \(j\) varies from 1 to \(G\). A \(T\) matrix is constructed for each row of the \(V\) matrix. The \(T\) matrix for the first row of \(V\) matrix is:

\[
T = \begin{pmatrix}
A_1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
A_2 & 0 & 0 & 0 & 2 & 1 & 0 & 0 \\
A_3 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
B_1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
B_2 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
C_1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
C_2 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\end{pmatrix}
\]
Similarly, a T matrix can be constructed for the 2\textsuperscript{nd} and 3\textsuperscript{rd} rows of the V matrix.

6.3.3.2 T Matrix Decomposition

LG schedule matrix U, can be built from the permuted T matrices. From \( i \)\textsuperscript{th} T matrix \( L_i \), permutation matrices can be obtained. One permutation matrix gives one row of matrix, U. A permutation matrix must have exactly one 1 in each row and column. For T matrix decomposition, two algorithms can be used.

1) Birkhoff Von Neumann decomposition

2) Slepian – Duguid decomposition

\textit{a) Birkhoff Von Neumann Decomposition:}

The permutation matrices are constructed sequentially. Considering another example,

\[
X = \begin{pmatrix}
1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 1
\end{pmatrix}
\]

This \( X \) matrix can be decomposed into 3 (the sum of elements of each row) matrices. The matrices are filled with the first occurrence of 1 in the \( X \) matrix satisfying the condition that each row and each column contains only one 1. After constructing each permutation matrix the corresponding element is subtracted from the main matrix \( X \).

\textit{Iteration 1:}

\[
\text{per1= } \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix}
\]

\[
\text{X= } \begin{pmatrix}
0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1
\end{pmatrix}
\]

after subtracting per1 from \( X \)

\textit{Iteration 2:}

\[
\text{per2= } \begin{pmatrix}
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
\text{X= } \begin{pmatrix}
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix}
\]

after subtracting per2 from \( X \)
Iteration 3:

\[
\begin{pmatrix}
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 
\end{pmatrix}, \quad \begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 
\end{pmatrix}
\]

After subtracting \(\text{per}_3\) from \(X\)

Since \(X\) matrix is not an all zero matrix even after 3 iterations (denoted by the sum of elements) the permutation matrices should be rearranged till \(X\) becomes an all zero matrix.

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 
\end{pmatrix}, \quad \begin{pmatrix}
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 
\end{pmatrix}, \quad \begin{pmatrix}
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 
\end{pmatrix}
\]

The main disadvantages of Birkhoff Von Neumann decomposition are: (1) If the \(X\) matrix does not become an all zero matrix after \(K\) iterations then rearranging the permutation matrices is difficult. (2) Since all the permutation matrices cannot be constructed simultaneously.

\textit{b) Slepi\-an – Duguid Decomposition:}

The disadvantages of Birkhoff Von Neumann Decomposition are overcome in this algorithm. So it is used instead of Birkhoff Von Neumann Decomposition. There are two phases (1) Greedy (2) Back-tracing Initially greedy is applied and if the main matrix is non zero then back-tracing is applied to complete the schedule.

\textit{i) Greedy}

Here, rows are assumed as inputs and columns as outputs. Considering the example matrix \(X\), shown above, the algorithm starts with scheduling \((1, 1)\) (input 1 and output 1) which is the first non-zero element of matrix \(X\).
Since all the permutations are empty initially, (1, 1) is scheduled in permutation \( P_1 \). This is illustrated in Table 6.4. The \( X_i \) matrices shown have the permutations as the rows, inputs as the columns and the elements of the matrix indicate the outputs to which the inputs are connected. As the same input or the same output cannot be scheduled twice in any of the permutations, conflicts arise and some of the input output pairs cannot be scheduled using greedy. One such example is shown in Table 6.4. When the schedule constructed is \( X_8 \), the next non-zero element in \( X \) that needs to be scheduled is (3,5). The only permutation free for input 3 is \( P_3 \) and output 5 is already scheduled in it. \( X_15 \) in Table 6.4 shows the final output of the greedy algorithm. (3,5) and (4,3) are not scheduled by greedy and need to be scheduled using back-tracing. This is indicated by the presence of zeros in \( X_{15} \).

**ii) Back Tracing**

The back-tracing step is based on Slepian-Duguid algorithm. Table 6.5 extends the above example to explain how (3,5) and (4,3) are scheduled. Considering (3,5) first, the permutations that have input 3 and output 5 free are identified. \( X_{15} \) of Table 6.4 shows that \( P_3 \) (row 3) has input 3 free and \( P_1 \) has output 5 free. So (3,5) is scheduled in \( P_3 \) as shown in \( X_{16} \) of Table 6.5. Now \( X_{16} \) has two 5's in \( P_3 \). Hence, 5 of \( P_3 \) is swapped with 3 of \( P_1 \) for the same input. \( X_{17} \) shows the resulting matrix. The procedure is repeated until no permutation has multiple inputs or outputs scheduled in it. \( X_{18} \) of Table 6.5 shows the final resulting matrix.
Table 6.5 Back-tracing for the LL and LG Schedules

\[
X_{15} = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 0 & 1 & 2 \end{pmatrix}, \quad X_{16} = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 5 & 1 & 2 \end{pmatrix}, \quad X_{17} = \begin{pmatrix} 1 & 5 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 3 & 5 & 1 & 2 \end{pmatrix}, \quad X_{18} = \begin{pmatrix} 1 & 5 & 2 & 3 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 3 & 5 & 1 & 2 \end{pmatrix}
\]

\text{inserting 5} \quad \text{swapping 3,5} \quad \text{inserting 3}

Each row of this matrix represents one permutation matrix with its elements denoting the column having an entry 1. For example, the first row of this \(X_{18}\) matrix (1 5 2 3 4) is expanded to \(Y_1\) which is given by

\[
Y_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}
\]

For the previous example of 7 linecards, the resultant matrices with Slepian–Duguid decomposition are obtained as

\[
P_1 = \begin{pmatrix} 1 & 4 & 5 & 2 & 6 & 3 & 7 \\ 3 & 4 & 6 & 1 & 7 & 2 & 5 \\ 2 & 5 & 7 & 3 & 4 & 1 & 6 \end{pmatrix}; \quad P_2 = \begin{pmatrix} 1 & 3 & 6 & 2 & 5 & 4 & 7 \\ 2 & 4 & 7 & 1 & 3 & 5 & 6 \end{pmatrix}; \quad P_3 = \begin{pmatrix} 1 & 5 & 6 & 4 & 7 & 2 & 3 \\ 2 & 3 & 7 & 5 & 6 & 1 & 4 \end{pmatrix}
\]

The \(U\) matrix can be constructed from these \(N\) permutation matrices.

**6.3.3 Construction of the LG Schedule Matrix \(U\)**

LG schedule matrix \(U\) is a \(N \times N\) matrix with \(\text{per}_i\) yielding the \(i^{th}\) row. In a permutation matrix, the elements represent the column index of \(U\) matrix. For example, considering \(\text{per}_1\) (row 1 of \(P_1\)), the first row of \(U\) matrix is constructed as follows. This matrix yields the first row of \(U\) as \([A_1 \ B_1 \ C_1 \ A_2 \ A_3 \ B_2 \ C_2]\). The complete \(U\) matrix can be constructed from the remaining \(\text{per}_i\) matrices.
6.3.4 Line Card-Line Card Scheduling

A line card-to-line card (L-L) schedule FT is a matrix with N rows corresponding to the N sending line cards, N columns corresponding to the N time-slots of the frame, and one receiving line card index per row-column intersection.

An L-L schedule FT is said to be valid iff a receiving line card appears exactly once in every row and column of FT, and at most \((L_i \times L_j) \mod N\) receiving line cards from group j are connected to sending line cards from group i in any column of FT (MEMS constraint).

6.3.4.1 Manipulation of the U Matrix

First, G temporary matrices are created iteratively each obtained by replacing the \(i^{th}\) group's alphabet by 1 and the remaining groups' alphabets by zero.
After getting this temporary matrix, say TEMP$_i$ (i=1,2,...G), it is then decomposed into L$_i$ permutation matrices where L$_i$ specifies the number of line cards in the $i^{th}$ group. $i=1,2,...,G$. For obtaining the permutation matrices, Slepian – Duguid algorithm is applied as for the LG schedule. The resultant matrices after decomposing TEMP$_i$ are given by finalper$_i$.

\[
\text{finalper}_1 = \begin{pmatrix}
1 & 3 & 2 & 6 & 4 & 5 & 7 \\
5 & 4 & 7 & 1 & 2 & 6 & 3 \\
4 & 6 & 5 & 3 & 7 & 1 & 2
\end{pmatrix}; \quad \text{finalper}_2 = \begin{pmatrix}
2 & 1 & 4 & 5 & 3 & 7 & 6 \\
6 & 7 & 3 & 2 & 1 & 4 & 5 \\
7 & 2 & 6 & 4 & 5 & 3 & 1
\end{pmatrix}; \quad \text{finalper}_3 = \begin{pmatrix}
7 & 2 & 6 & 5 & 3 & 1 \\
3 & 5 & 1 & 7 & 6 & 2 & 4
\end{pmatrix}
\]

All the finalper matrices are then concatenated to a 'fullper' matrix that is given below.

\[
\text{fullper} = \begin{pmatrix}
1 & 3 & 2 & 6 & 4 & 5 & 7 \\
5 & 4 & 7 & 1 & 2 & 6 & 3 \\
4 & 6 & 5 & 3 & 7 & 1 & 2 \\
2 & 1 & 4 & 5 & 3 & 7 & 6 \\
6 & 7 & 3 & 2 & 1 & 4 & 5 \\
7 & 2 & 6 & 4 & 5 & 3 & 1 \\
3 & 5 & 1 & 7 & 6 & 2 & 4
\end{pmatrix}
\]
6.3.4.2 Constructing the Line Card – Line Card Schedule Matrix

The LL schedule matrix FT can be constructed by considering every column index of the ‘fullper’ matrix as the row index of FT, the elements of fullper are the column indices of FT. The elements of FT are the row indices of fullper. The FT matrix thus constructed is given below:

<table>
<thead>
<tr>
<th>Time slots</th>
<th>Receivers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 4 7 3 2 5 6</td>
</tr>
<tr>
<td></td>
<td>4 6 1 2 7 3 5</td>
</tr>
<tr>
<td></td>
<td>7 1 5 4 3 6 2</td>
</tr>
<tr>
<td></td>
<td>2 5 3 6 4 1 7</td>
</tr>
<tr>
<td></td>
<td>5 2 4 1 6 7 3</td>
</tr>
<tr>
<td></td>
<td>3 7 6 5 1 2 4</td>
</tr>
<tr>
<td></td>
<td>6 3 2 7 5 4 1</td>
</tr>
</tbody>
</table>

From the ‘FT’ matrix, it can be inferred that in timeslot 1 the sender line cards [1 2 3 4 5 6 7] are connected to receiver line cards [1 4 7 2 5 3 6] respectively.

6.3.5 Modifications to the Algorithm

6.3.5.1 Double Back-tracing

For certain combinations of input, it was found that single back tracing was not sufficient for scheduling of all the line cards. As a result of this in the final schedule matrix some elements are zero i.e. the corresponding time slots are not utilized. Double back tracing was applied to resolve this problem. An example of this is illustrated in Table 6.6.

For constructing the flow matrix from the capacity matrix given in Table 6.6, single back tracing was not sufficient (Fig. 6.7). From the figure, it can be inferred that single back tracing does not find a flow matrix which completely utilizes the capacity matrix.
Table 6.6 Example for application of double back-tracing

Total no. of linecards $N$: 20
Total no. of Groups $G$: 5
No. of Line cards in each Group $L$: 4

<table>
<thead>
<tr>
<th>C</th>
<th>RL</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0 0 1 1 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1 1 0 1 0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1 1 1 0 0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1 1 0 0 1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

—► denotes a connection  —► denotes no connection

Fig. 6.7 Single Back Tracing
Double back tracing finds a suitable flow matrix so that all the elements are scheduled. This method traverses the graph one more time to complete the scheduling. This is shown in Fig. 6.8. Double back tracing has time and memory constraints but full capacity utilization was found to be possible.

6.5.2 Construction of the LG Schedule without the GG Schedule Matrix V

The T matrix constructed from the V matrix can also be constructed directly from the S^i matrices, without constructing the V matrix. The i^{th} T matrix is formed from the i^{th} rows of all the S^i matrices (i=1,2,...,G). This modification also resulted in time and memory savings.

6.4 Results and Discussion

6.4.1 Software Simulation Results

The algorithm was implemented using MATLAB 6.1 for functional verification before implementation using VHDL. The results obtained for some configurations with double back-tracing are given in Table 6.7.
Table 6.7 Scheduling results obtained with double backtracing

**Ex. 1**
No. of line cards N: 5
No. of Groups G: 2
No. of Line cards in each group L1: 3, L2: 2.

<table>
<thead>
<tr>
<th>LG Schedule Matrix</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LL Schedule Matrix</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Time taken for scheduling: 0.07 sec

**Ex. 2:**
No. of line cards N: 8
No. of Groups G: 4
No. of Line cards in each group L1: 3, L2: 2, L3= 1, L4= 2.

<table>
<thead>
<tr>
<th>LG Schedule Matrix</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
### LL Schedule Matrix

\[
\begin{bmatrix}
1 & 4 & 3 & 5 & 8 & 2 & 6 & 7 \\
4 & 3 & 6 & 8 & 2 & 5 & 7 & 1 \\
7 & 2 & 5 & 1 & 4 & 8 & 3 & 6 \\
2 & 5 & 4 & 3 & 6 & 7 & 1 & 8 \\
5 & 1 & 7 & 4 & 3 & 6 & 8 & 2 \\
6 & 8 & 1 & 2 & 7 & 3 & 5 & 4 \\
3 & 7 & 8 & 6 & 1 & 4 & 2 & 5 \\
8 & 6 & 2 & 7 & 5 & 1 & 4 & 3
\end{bmatrix}
\]

Time taken for scheduling: 0.1 sec

---

### Ex. 3

No. of line cards N: 12  
No. of Groups G: 6  
No. of Line cards in each group L: 2

### LG Schedule Matrix

\[
\begin{bmatrix}
1 & 2 & 1 & 2 & 4 & 6 & 4 & 6 & 3 & 5 & 3 & 5 \\
2 & 1 & 2 & 1 & 6 & 4 & 6 & 4 & 5 & 3 & 5 & 3 \\
1 & 2 & 1 & 2 & 3 & 4 & 4 & 5 & 6 & 5 & 6 & 6 \\
2 & 1 & 2 & 1 & 4 & 4 & 3 & 6 & 5 & 6 & 5 & 6 \\
3 & 4 & 3 & 4 & 5 & 6 & 5 & 6 & 1 & 2 & 1 & 2 \\
4 & 3 & 4 & 3 & 6 & 5 & 6 & 5 & 2 & 1 & 2 & 1 \\
3 & 4 & 3 & 4 & 2 & 5 & 2 & 5 & 1 & 6 & 1 & 6 \\
4 & 3 & 4 & 3 & 5 & 2 & 5 & 2 & 6 & 1 & 6 & 1 \\
5 & 6 & 5 & 6 & 1 & 3 & 1 & 3 & 2 & 4 & 2 & 4 \\
6 & 5 & 6 & 5 & 3 & 1 & 1 & 4 & 2 & 4 & 2 & 4 \\
5 & 6 & 5 & 6 & 1 & 2 & 1 & 2 & 3 & 4 & 3 & 4 \\
6 & 5 & 6 & 5 & 2 & 1 & 2 & 1 & 4 & 3 & 4 & 3 
\end{bmatrix}
\]

---

### LL Schedule Matrix

\[
\begin{bmatrix}
1 & 3 & 2 & 4 & 7 & 11 & 8 & 12 & 5 & 9 & 6 & 10 \\
3 & 1 & 4 & 2 & 11 & 7 & 12 & 8 & 9 & 5 & 10 & 6 \\
2 & 4 & 1 & 3 & 5 & 8 & 6 & 7 & 10 & 11 & 9 & 12 \\
4 & 2 & 3 & 1 & 8 & 5 & 7 & 6 & 11 & 10 & 2 & 9 \\
5 & 7 & 6 & 8 & 9 & 12 & 10 & 11 & 1 & 3 & 2 & 4 \\
7 & 5 & 8 & 6 & 12 & 9 & 11 & 10 & 3 & 1 & 4 & 2 \\
6 & 8 & 5 & 7 & 3 & 10 & 4 & 9 & 2 & 12 & 1 & 11 \\
8 & 6 & 7 & 5 & 10 & 3 & 9 & 4 & 12 & 2 & 11 & 1 \\
9 & 1 & 10 & 12 & 1 & 6 & 2 & 5 & 4 & 7 & 3 & 8 \\
11 & 9 & 12 & 10 & 6 & 1 & 5 & 2 & 7 & 4 & 8 & 3 \\
10 & 12 & 9 & 11 & 2 & 4 & 1 & 3 & 6 & 8 & 5 & 7 \\
12 & 10 & 11 & 9 & 4 & 2 & 3 & 1 & 8 & 6 & 7 & 5
\end{bmatrix}
\]

Time taken for scheduling = 1.4 sec

---

111
Table 6.8 Comparison of scheduling time with and without double back-tracing

<table>
<thead>
<tr>
<th>Without double backtracing</th>
<th>With double backtracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Delay (sec)</td>
</tr>
<tr>
<td>----</td>
<td>------------</td>
</tr>
<tr>
<td>5</td>
<td>0.06</td>
</tr>
<tr>
<td>8</td>
<td>0.09</td>
</tr>
<tr>
<td>12</td>
<td>1.36</td>
</tr>
</tbody>
</table>

Fig. 6.9 Simulation time for different no. of line cards

A comparison of time taken with and without double back-tracing is given in Table 6.8. This indicates a maximum of around 20% increase in delay with double back-tracing, which can be tolerated by most applications demanding full capacity utilization. The time taken for the algorithm to find a valid schedule as the number of line cards increases is shown in Fig. 6.9. The time taken for the algorithm was found to increase slightly when the scheduling involved double back tracing. This can be inferred from the pie-charts (Figure 6.10 (a) and (b)) that show the split up of time for the three scheduling phases (GG, LG and LL) in percentages. Without double back tracing, the GG scheduling takes 40% of the total time consumed, but with double back tracing introduced for the flow matrix construction, this time was found to increase marginally to 44%.
6.4.2 Hardware Implementation

The scheduling algorithm based on Time Division Multiplexing was verified for feasibility of hardware implementation. After simulation, synthesis was carried out using FPGAs to get the critical path and achieve maximum speed. The steps followed for hardware implementation included Design Entry, Simulation, Synthesis, Implementation and Device Programming. The scheduling algorithm was first coded using VHDL. Xilinx Project Navigator (Version 6.2i) was used to implement the scheduling algorithm. Simulation was done using ModelSim XE II/Starter 5.7g in ISE software. Fig. 6.11 shows the simulation results.

In the synthesis phase the design was translated into gates and optimized for the target architecture. The design was first successfully analyzed, before it was translated into gates and optimized. The Spartan family of devices was selected for synthesis for ease of re-programmability and low cost. The implementation stage consisted of taking the synthesized netlist through translation, mapping, placing and routing. After synthesis, the design was downloaded in a FPGA kit. The logical design file created during design entry was converted into a physical file format for a specific Xilinx architecture.

![Fig. 6.10 Time taken by different phases of the algorithm](image)
Figure 6.11 Simulation results for Example in Table 6.1

Table 6.9 Scheduling time with different devices

<table>
<thead>
<tr>
<th>No. of linecards</th>
<th>Device used</th>
<th>Time taken (ns)</th>
<th>Total estimated power consumption (mw)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Spartan 2 xc2s100tq144</td>
<td>18.649</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Spartan 2 xc2s150 fg256</td>
<td>20.143</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Spartan 2 xc2s150fg256</td>
<td>23.736</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>Spartan 2 xc2s200fg456</td>
<td>24.924</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>Spartan 3 xc3s5000fg900</td>
<td>10.280</td>
<td>335</td>
</tr>
<tr>
<td>8</td>
<td>Spartan 3 xc3s5000fg900</td>
<td>10.723</td>
<td>335</td>
</tr>
<tr>
<td>10</td>
<td>Spartan 3 xc3s5000fg900</td>
<td>10.981</td>
<td>335</td>
</tr>
</tbody>
</table>

Table 6.9 shows the scheduling time taken by different devices as well as an estimate of the power consumption. From the table, it can be inferred that the total time for the scheduling algorithm increases as the number of line cards increases for a
particular device. This is due to the increased utilization of the device resources. For more number of line cards, if a device with a higher number of Input Output blocks was chosen, the scheduling time was found to considerably reduce. Among the tested devices, the device Spartan 3xc3s5000fg900 was found to provide the least delay, but involved more power consumption. For scheduling 8 line cards, this device took a time delay of 10.723 ns. The program for 2 line cards was downloaded on the FPGA kit-xc2s100tql44 shown in Fig.6.12. The results obtained, shown in Fig.6.13, indicate the scheduling of 2 line cards. The limitation on the number of input-output LEDs available in the kit restricted the number of line cards tested to 2.

In summary, providing throughput guarantees would normally require a complicated centralized scheduler to configure the switch fabric. In the hybrid electro-optic switch fabric studied in this work, only the lower-capacity local switches in each group need to be reconfigured frequently. MEMS switches change only when line cards are added or removed. All the central switching can be packaged in a single rack. A scheduling algorithm based on Time Division Multiplexing can be used to implement the load balancing stage of the router. In this work, the TDM algorithm was enhanced, thus achieving the target of full capacity utilization. The functional verification of the algorithm was done with MATLAB 6.1. The same algorithm was also checked for hardware feasibility using VHDL. After synthesis of the VHDL code, the time taken for scheduling, the critical path and the power requirements were determined. It was found that the device Spartan 3 xc3s5000fg900 provided a delay of 10.723 ns, for scheduling 8 line cards. By providing guarantees on throughput, fault-tolerance, security and traffic predictability, this novel backbone architecture is promising for the future scalability and reliability of the Internet.
Fig. 6.12 FPGA Kit xc2s100 tq144 – Top view

**FPGA Final Output**

Input LEDs Output LEDs

**Input Gin_bit=“10”**

Fig. 6.13 FPGA output for 2 line cards