Feasibility Study of Configuration Algorithms for the Switch Fabric of a Load-Balanced Switch with an Arbitrary Number of Linecards

Srikanth Arekapudi
sarek@stanford.edu
Computer Systems Laboratory
Stanford University
Stanford, CA 94305-9030

I. Introduction

A. Background

With increasing demand for high speed communications there is an urgent need for high capacity and low power router architectures. One such architecture using optical technology is proposed in [1].

Figure 1 shows the block diagram of the proposed architecture in [1]. As mentioned in [1] the number of linecards present can keep on changing as more and more linecards are added as network grows or linecards are removed as they fail. The load-balanced switch works by 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, traffic must be spread equally over the remaining linecards. The switch fabric must be able to schedule the traffic uniformly over the linecards present. [1] describes an algorithm to do this, and proves that it will always find a valid configuration.

We will assume throughout that 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, ..., L_G are fixed for a given linecard arrangement.

During every frame of N time-slots each sending linecard needs to be connected exactly once to each of the N receiving linecards. Similarly, each receiving linecard needs to be connected exactly once to each of the N sending linecards. Furthermore, in every time-slot, each sending linecard cannot connect to more than one receiving linecard, and vice-versa.

Put mathematically, if sending linecard i is connected to receiving linecard T_{ij} in time-slot j, then:

$$\begin{cases} T_{ij'} \neq T_{ij} & \text{for all } j' \neq j \\ T_{i'j} \neq T_{ij} & \text{for all } i' \neq i \\ T_{ij} \in \{1, ..., N\} & \text{for all } i, j \end{cases}$$

We'll call T the *linecard schedule*. T is a Latin square, i.e. the numbers from 1 to N appears exactly once in every row and every column. We will refer to a time-slot as a column.

For instance, let's assume that $L_1 = 3$, $L_2 = 2$, and $L_3 = 2$ (i.e., $G = 3, N = 7$). Then the following is a linecard schedule:

$$T = \left(\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 3 & 4 & 5 & 6 & 7 & 1 \\ 3 & 4 & 5 & 6 & 7 & 1 & 2 \\ \hline 4 & 5 & 6 & 7 & 1 & 2 & 3 \\ 5 & 6 & 7 & 1 & 2 & 3 & 4 \\ \hline 6 & 7 & 1 & 2 & 3 & 4 & 5 \\ 7 & 1 & 2 & 3 & 4 & 5 & 6 \end{array}\right)$$

B. Findings Of The Work

It can be concluded from above that the line cards need to be scheduled in an efficient way, so that it will not effect the throughput. The basic algorithm for implementing the linecard scheduling was discussed in [1] in detail. The typical requirement for the router would be to perform the scheduling with in 50ms. Software implementations are slow and it is highly impossible to achieve these speeds with them. The basic software implementation results in [1] show that it takes 50 seconds to complete a scheduling of 640 linecards with 40 groups. Custom hardware implementations can be used to achieve the 50ms target. This can be done by exploiting the parallelism in the algorithm and efficiently implementing that in hardware. The following are the highlights of this work

1) Showed that the 50ms target is achievable using a 4ns clock cycle time for the hardware.
2) Modified the Ford-Fulkerson algorithm [13] and implemented the Group-Group (GG) schedule in 2 phases, greedy and back-tracing. It is the back-tracing that uses modified Ford-Fulkerson algorithm targeted towards hardware implementation.
3) Used an algorithm based on Slepian-Duguid [14] instead of Birkhoff-von Neumann decomposition theorem [5], [6] to exploit the parallelism for the Linecard-Group (LG) and Linecard-Linecard (LL) schedules. This step is also divided into greedy and back-tracing.
4) A simple priority encoder is used for searches ,i.e, when ever a decision needs to be made in hardware.
5) Further parallelism is exploited in LG and LL schedules.

This work focuses on

- Initial feasibility study of different algorithms, implementing them in C for functional verification.

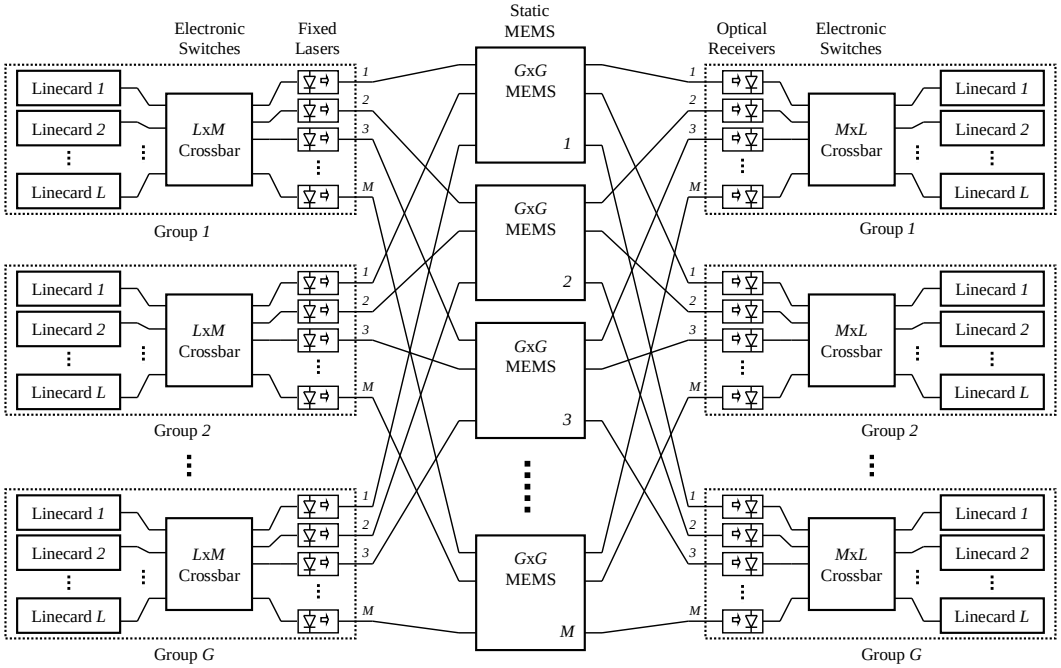

Fig. 1. Hybrid optical and electrical switch fabric.

- Implementing the recommended algorithms from above in Hardware (verilog) exploiting the parallelism.
- Synthesizing the basic modules using FPGA to understand the critical path and hence the maximum frequency (speed) that can be achieved.
- Develop the C-models for the hardware implemented to compute the clock cycles it takes for different configurations.

The algorithms used in the implementation of the final software and hardware versions are described in the next sections. These algorithms would be useful for applications that would need to implement the Ford-Fulkerson and Slepian-Duguid algorithms in hardware. You can skip sections II III, if you have already gone through [1]. They are mentioned here for reference.

II. Constructing a Valid G-G Schedule

A. Algorithm for Constructing a Valid G-G Schedule

We will now construct an algorithm that recursively builds a valid group schedule time-slot after time-slot, for the N time-slots of the frame.

At the start:

Let t be the number of time-slots left to schedule after each iteration. At the start, $t = N$, since all the time-slots are unscheduled. Also, let $M \equiv M^t = M^N$ 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", ...). At the start, for all i, j, $M_{ij} = L_i \cdot L_j$, i.e. there are $L_i \cdot L_j$ connections to schedule from sending group i to receiving group j during the whole frame.

Iteratively:

For $t = N, N-1, ..., 1$, proceed as follows.

1) For each i, j, do the decomposition of M^t_{ij} in base t: $M^t_{ij} = P^t_{ij} \cdot t + Q^t_{ij}$ (i.e. $P^t = \left\lfloor \frac{1}{t} M^t \right\rfloor$, $Q^t = M^t - P^t \cdot t$). In this iteration, we will start by scheduling P^t, and then consider the remainder Q^t and schedule a part of it such that all the constraints are satisfied.
2) Define the vectors a^t and b^t such that

$$\begin{cases} a^t_i = \frac{\sum_{j'=1}^{G} Q^t_{ij'}}{t} & \text{for all } i \\ b^t_j = \frac{\sum_{i'=1}^{G} Q^t_{i'j}}{t} & \text{for all } j \end{cases}$$

   a^t and b^t are integer vectors (cf proof).
3) Find a 0-1 matrix $R^t \leq Q^t$ such that:

$$\begin{cases} \sum_{j'=1}^{G} R^t_{ij'} = a^t_i & \text{for all } i \\ \sum_{i'=1}^{G} R^t_{i'j} = b^t_j & \text{for all } j \\ R^t_{ij} \in \{0, 1\} & \text{for all } i, j \end{cases}$$

   R^t can be build using Ford-Fulkerson max-flow algorithm.
4) Use the schedule $S^t = P^t + R^t$ for this time-slot. Update $M^{t-1} = M^t - S^t$.

B. Example

We build the matrix V given the schedules, S^t, provided in Table I. More specifically, S^t_{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 instance, the schedule

$$S^7 = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

helps us create the first column of V having two A's and one B in the first row, one B and one C in the second row, and

TABLE I

EXAMPLE OF APPLICATION OF THE ALGORITHM

$$M^7 = \begin{pmatrix} 9 & 6 & 6 \\ 6 & 4 & 4 \\ 6 & 4 & 4 \end{pmatrix}, P^7 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, Q^7 = \begin{pmatrix} 2 & 6 & 6 \\ 6 & 4 & 4 \\ 6 & 4 & 4 \end{pmatrix}, R^7 = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}, S^7 = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}.$$

$$M^6 = \begin{pmatrix} 7 & 5 & 6 \\ 6 & 3 & 3 \\ 6 & 4 & 3 \end{pmatrix}, P^6 = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, Q^6 = \begin{pmatrix} 1 & 5 & 0 \\ 0 & 3 & 3 \\ 6 & 4 & 3 \end{pmatrix}, R^6 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, S^6 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

$$M^5 = \begin{pmatrix} 6 & 4 & 5 \\ 5 & 2 & 3 \\ 4 & 4 & 2 \end{pmatrix}, P^5 = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, Q^5 = \begin{pmatrix} 1 & 4 & 0 \\ 0 & 2 & 3 \\ 4 & 4 & 2 \end{pmatrix}, R^5 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, S^5 = \begin{pmatrix} 2 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}.$$

$$M^4 = \begin{pmatrix} 4 & 4 & 4 \\ 4 & 1 & 3 \\ 4 & 3 & 1 \end{pmatrix}, P^4 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, Q^4 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 3 \\ 0 & 3 & 1 \end{pmatrix}, R^4 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, S^4 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

Finally, for t=3,2,1: $M^t = \begin{pmatrix} t & t & t \\ t & 0 & t \\ t & t & 0 \end{pmatrix} = t \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}, R^t = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ and $S^t = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$.

one A and one C in the last row. S^6 will determine the second column, S^5 will determine the third column, and so on. The resulting matrix is

$$V = \left(\begin{array}{ccccccc} AAB & ABC & AAC & ABC & ABC & ABC & ABC \\ \hline BC & AB & AB & AB & AC & AC & AC \\ \hline AC & AC & BC & AC & AB & AB & AB \end{array}\right)$$

III. Valid L-L Schedule

A. From a Valid G-G Schedule to a Valid L-G Schedule

We will now construct an algorithm that successively builds a valid L-G schedule given a valid G-G schedule, and then a valid L-L schedule given a valid L-G schedule.

For each $1 \leq j \leq G$, consider row j in V. In our example, the first row is:

$$(\ AAB \quad ABC \quad AAC \quad ABC \quad ABC \quad ABC \quad ABC\ )$$

We want to subdivide each row j 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 divide them into L_i subscripted letters ("sub-letters") of L_j elements. In our example, we transform the letters of V 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, we replace the $A's$ arbitrarily with 3 A_1's, 3 A_2's and 3 A_3's:

$(A_1A_1B_1;\ A_1B_1C_1;\ A_2A_2C_1;\ A_2B_1C_1;\ A_3B_2C_2;\ A_3B_2C_2;\ A_3B_2C_2)$

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. Let's form a new matrix that has sub-letters as inputs and columns as outputs. In this new matrix, all columns and all rows have L_j elements. In our example, the new matrix for the first row of V is :

$$\left(\begin{array}{c||ccccccc} & col.1 & col.2 & col.3 & col.4 & col.5 & col.6 & col.7 \\ \hline\hline A_1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ A_2 & 0 & 0 & 2 & 1 & 0 & 0 & 0 \\ A_3 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ B_1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ B_2 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ C_1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ C_2 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}\right)$$

We can now apply the Birkhoff-von Neumann decomposition theorem to this matrix, by decomposing it into a sum of L_j permutations [5], [6].[1] We obtain L_j permutations. By reading column after column, each of these permutations gives a sequence of sub-letters that corresponds to a row of the desired L-G schedule. Therefore, the L_j permutations yield the L_j rows of the L-G schedule corresponding to group j. In our example, the first permutation could be:

$$\left(\begin{array}{c||ccccccc} & col.1 & col.2 & col.3 & col.4 & col.5 & col.6 & col.7 \\ \hline\hline A_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ A_3 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ B_1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ B_2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ C_1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ C_2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right),$$

yielding the first row of:

$$\begin{pmatrix} A_1 & B_1 & A_2 & C_1 & A_3 & B_2 & C_2 \\ A_1 & C_1 & A_2 & B_1 & C_2 & A_3 & B_2 \\ B_1 & A_1 & C_1 & A_2 & B_2 & C_2 & A_3 \end{pmatrix}$$

We finally replace each sub-letter by the corresponding letter, and get the valid L-G schedule. Upon examination of the algorithm, it is clear that we only permute letters within the same

[1] Because all elements are integers we could use graph-coloring instead [7][8][9][10].

column of the same sending group, thus yielding a valid L-G schedule. In our example, the resulting L-G schedule is:

$$U = \left(\begin{array}{ccccccc} A & B & A & C & A & B & C \\ A & C & A & B & C & A & B \\ B & A & C & A & B & C & A \\ \hline B & A & A & B & A & C & C \\ C & B & B & A & C & A & A \\ \hline A & A & C & C & A & B & B \\ C & C & B & A & B & A & A \end{array}\right)$$

B. From a Valid L-G Schedule to a Valid L-L Schedule

In the previous section, we constructed a valid L-G schedule given a valid G-G schedule. In this section, we will transform the valid L-G schedule into a valid L-L schedule.

We apply the Birkhoff-von Neumann theorem (or graph-coloring) for each letter. First, we replace each *A* with a "1", and every other letter with a "0". For our example, we get:

$$\left(\begin{array}{ccccccc} A & 0 & A & 0 & A & 0 & 0 \\ A & 0 & A & 0 & 0 & A & 0 \\ 0 & A & 0 & A & 0 & 0 & A \\ \hline 0 & A & A & 0 & A & 0 & 0 \\ 0 & 0 & 0 & A & 0 & A & A \\ \hline A & A & 0 & 0 & A & 0 & 0 \\ 0 & 0 & 0 & A & 0 & A & A \end{array}\right)$$

$$\rightarrow \left(\begin{array}{ccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right)$$

We then decompose the above matrix into the sum of L_i different permutations, such that the l^{th} permutation will indicate at which times linecard l is scheduled. Since there are exactly L_1 ones (corresponding to the L_1 *A*'s) in each row and column, this is possible by Birkhoff-von Neumann. In our example, we can decompose the above matrix into the sum of three permutations:

$$\left(\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right) + \left(\begin{array}{ccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right)$$

$$+ \left(\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

Applying this method to each letter, we then create a valid L-L schedule, with exactly one occurrence of each receiving linecard index in each row and column. In our example, we get the following valid L-L schedule, hence concluding the construction process:

$$T = \left(\begin{array}{ccccccc} 1 & 4 & 2 & 6 & 3 & 5 & 7 \\ 2 & 6 & 1 & 5 & 7 & 3 & 4 \\ 4 & 3 & 7 & 1 & 5 & 6 & 2 \\ \hline 5 & 1 & 3 & 4 & 2 & 7 & 6 \\ 7 & 5 & 4 & 3 & 6 & 2 & 1 \\ \hline 3 & 2 & 6 & 7 & 1 & 4 & 5 \\ 6 & 7 & 5 & 2 & 4 & 1 & 3 \end{array}\right)$$

IV. Implemented G-G Schedule

As mentioned in the example of Table I [1], we are trying to find a 0-1 matrix $R^t \leq Q^t$. Table II shows an example of what we are trying to achieve. C is a capacity matrix $\leq Q^t$ such that:

$$\{\ C^t_{ij} = 1 \quad if \quad Q^t_{ij} > 0 \quad \text{for all } i, j\ \}$$

The capacity matrix shows the connections that can be possible between groups. Each group has a fixed sending rate and receiving rate. RL (rate left as the sending groups are visualised as being on the left) is the sending rates of the groups and RR (rate right as the receiving groups are visualised as being on the right) is the receiving rates of the groups. For the example given in Table II 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. We considered a 0-1 capacity matrix as it is easier to implement it as a memory in hardware.

As mentioned in [1], Ford-Fulkerson algorithm [13] can be used for this purpose. 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. However since each entry in C is going to be either a 1 or a 0 and also since this is a bi-partite graph the algorithm can be modified to make it simpler. Figure 2 shows the difference between the Ford-Fulkerson implementation and the modified Ford-Fulkerson implementation. In our 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 nothing but the connections that the group (node) needs to have as explained before. The first part of the algorithm uses a greedy approach to schedule the groups. The later part uses back-tracing (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).

A. Greedy

Table III shows how the algorithm works for the capacity and rate matrices given in Table II. In our 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. C_1 is the capacity matrix, F_1 is the flow matrix, RL_1 and RR_1 are the rates of the the left and right nodes after applying the greedy algorithm. A flow matrix keeps track of the connections already made. Combining C_1 with F_1 at anytime gives us 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 a capacity element that points to a group whose rate right is non-zero, If there is then move the element to flow matrix.

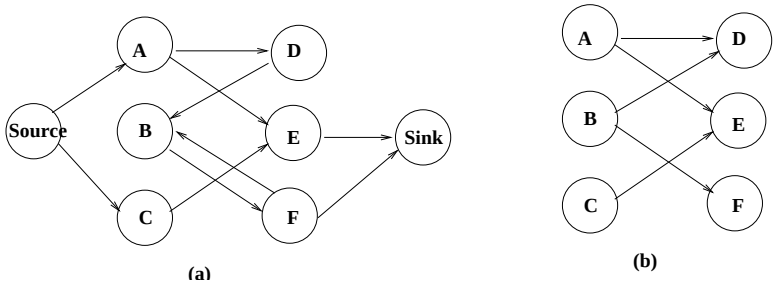

Fig. 2. Ford Fulkerson and Modified Ford Fulkerson

TABLE II
EXAMPLE FOR SCHEDULING

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

TABLE III
EXAMPLE APPLICATION OF THE ALGORITHM

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

$$RL_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, RR_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, C_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}, F_1 = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, R = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

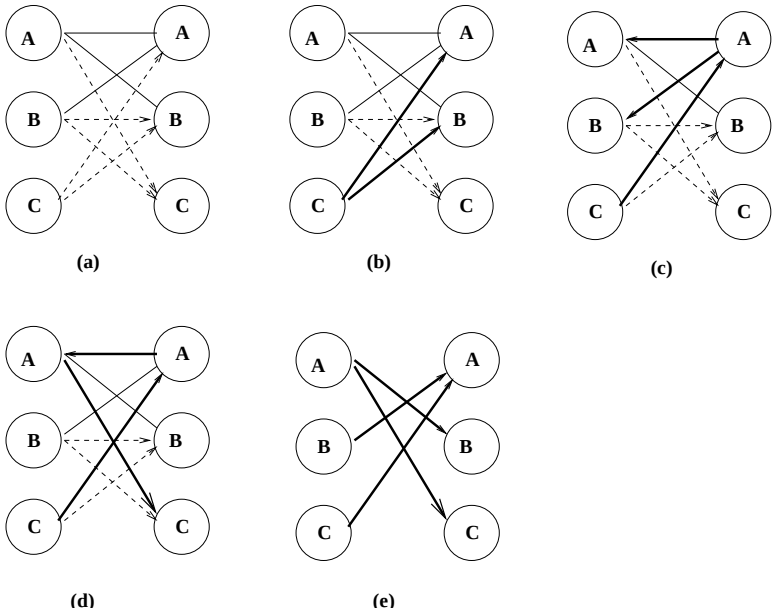

Fig. 3. Back Tracing in GG Schedule a) After applying greedy b) Tracing Sending Group C c) Tracing Receiving Group A backwards d) Tracing Sending Group A e) Final Schedule

3) Reduce the rate left of the sending group and rate right of the receiving group.

In our 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. Considering the example in detail, 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 left, making the RL of both Group A, Group B. 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.

B. Back Tracing

As shown in Table III RL_1 and RR_1 are non zero and back-tracing needs to be applied to find the remaining schedule. In our example Group C has a non-zero RL, however it has no connection to the non-zero RR element (Receiving group C) in the capacity matrix. Therefore, we need to do back-tracing Figure 3 shows how the back-tracking is done using a simplified version of BFS algorithm. Initially the greedy algorithm finds the paths $A-A$, $A-B$, $B-A$ (shown by thin solid lines in Figure 3a)). These arcs form the flow matrix. The remaining part of the capacity matrix is shown by the dashed lines. 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 Figure 3b) by thick solid lines. Assuming $C-A$ is traced back. From node A on the right we can either go to A or B as shown Figure 3c). Assuming that we pick A as in Figure 3d). A has a path to C whose rate is non-zero and the final paths are shown in Figure 3e). As in Ford-Fulkerson algorithm we keep track of the predecessors of the nodes traced, so that we can update the capacity and flow matrices once the BFS is done finding an augmenting path, making them ready to find the next augmenting path. As in original BFS we color the nodes to keep track of the nodes already visited so that we do not revisit a node accidentally.

The entire GG schedule construction as in Table I which uses the algorithm explained above is implemented in hardware.

V. Implementation of LG and LL schedules

The main part of LL and LG schedules is to obtain P permutation matrices (schedules), given a matrix which has P 1's in each row and each column. For example, consider a matrix S as shown below

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

A permutation matrix has exactly one 1 in each row and column. Initially the permutation matrices are empty, and the algorithm starts with filling each of the permutation matrices with the non-zero elements of the matrix S. Given a matrix S, the algorithm should generate the following permutation matrices.

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

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

[1] proposes an algorithm based on Birkhoff-von Neumann decomposition theorem [5], [6] which applies Ford-Fulkerson algorithm [13] multiple times and generates each permutation serially. Here we propose an algorithm based on Slepian-Duguid [14] which when finishes produces P permutation matrices at once. The initial part of the algorithm is done using greedy and the later part does the back-tracing to complete the schedule.

A. Greedy

Here we refer rows as inputs and columns as outputs for convenience. Consider the example matrix S as shown above. The algorithm starts with scheduling (1,1) (input 1 and output 1) which is the first non-zero element of matrix S. Since all the permutations are empty initially, we can schedule this in permutation P_1. This is illustrated in Table IV. The S_i matrices shown have the permutations as the rows, inputs as the columns and the elements are the outputs to which the inputs are connected to. i refers to the i^{th} step in scheduling. So we start with filling row 1 (P_1) and column 1 (input 1) with 1 (output 1). Similarly the rest of them are filled on first come first serve basis. As we cannot schedule same input or the same output 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 IV. At some point in time after we start scheduling, the schedule looks like S_8. The next non-zero element is S 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. S_{15} in Table IV 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.

B. Back Tracing

The back-tracing step is based on Slepian-Duguid [14]. A 0 in S_{15} means that the permutation and the input combination is not scheduled. Table V extends the above example to explain how (3,5) and (4,3) are scheduled. Lets consider (3,5) first. Initially we start with identifying the permutations which have input 3 and output 5 free. As can be seen from S_{15} of Table V P_3 (row 3) has input 3 free and P_1 has output 5 free. So we schedule (3,5) in P_3 as shown in S_{16} of Table V. Now as can be observed in S_{16} we have two 5's in P_3, we swap 5 of P_3 with 3 of P_1 for the same input. S_{17} shows the resulting matrix. As we do not have multiple 5's scheduled in P_1, we stop here. Otherwise we repeat the procedure of swapping until no permutation has multiple inputs or outputs scheduled in it. S_{18} of Table V shows the final resulting matrix.

VI. Hardware Implementation

The core algorithms explained in the previous sections are implemented in hardware. Optimizations have been done to the hardware implementation to make it efficient and faster. Parallelism has been exploited wherever possible. A priority encoder is used whenever we needed to make a decision, for example choosing between A and B as shown Figure 3c) can be done using a simple priority encoder.

TABLE IV
GREEDY ALGORITHM FOR LL AND LG SCHEDULES

$$S_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, S_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, S_8 = \begin{pmatrix} 1 & 3 & 2 & 0 & 0 \\ 3 & 1 & 4 & 0 & 0 \\ 4 & 5 & 0 & 0 & 0 \end{pmatrix}, S_{15} = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 0 & 1 & 2 \end{pmatrix},$$

TABLE V
BACK TRACING FOR THE LL AND LG SCHEDULES

$$S_{15} = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 0 & 1 & 2 \end{pmatrix}, S_{16} = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 5 & 1 & 2 \end{pmatrix}, S_{17} = \begin{pmatrix} 1 & 5 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 3 & 5 & 1 & 2 \end{pmatrix}, S_{18} = \begin{pmatrix} 1 & 5 & 2 & 3 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 3 & 5 & 1 & 2 \end{pmatrix},$$

A. Core for GG schedule

Efficient bitmap techniques have been used to process some of the stuff. For example an and operation can be performed for the rate of the rightnodes bitmap (has a 1 when rate of group on the right is > 0, has a 0 otherwise) and the capacity row (belongs to one of the groups on the left side) to see whether any path exists between this group on the left side to any of the groups on the right side. These bitmaps are updated automatically whenever the memories are updated. Similar techniques are used extensively to reduce the number of cycles it takes for constructing the schedule. As mentioned in the algorithm we need a left and right queue to keep track of the possible nodes that can be part of the augmenting path. These queues are also implemented as bitmaps and the corresponding node bit is made 0 when it is dequeued and made 1 when enqueued. Priority encoder is used to dequeue. Different memories used are described below.

- Qmatrix Memory - This memory is same as the Q^t used in describing the algorithm. A processor is assumed to do minor preprocessing on the M^N (generated from the group information) to update Q^t (Q^N). This memory is organized as a vertical matrix which stores the (i,j) of the Q^t matrix in one row. Since the Q^t matrix is NxN we have N^2 rows each log(N) bits wide.
- Capacity Matrix - This is used to store the capacity matrix for a particular time slot. The number of schedules required is equal to the number of time slots. At the beginning of each time slot the values of the Q^t matrix are compared to see whether they are equal to the time slot being scheduled or not, if it is equal then the a 0 is placed in the capacity memory and flow1 memory, but a one is placed in the flow2 memory (predetermined path). rateright and rateleft memories are updated accordingly. If this is not equal and non-zero a 1 is placed in the capacity matrix.The capacity and flow1 memories are used to obtain the schedule. The flow2 memory and the Q^t matrix memories are updated when ever flow1 is updated. Size of this memory is N x N.
- Flow1 Memory - This is used to store the values of the reverse paths while scheduling. Size of this memory is N x N.
- Flow2 Memory - This is used to store the values of the predetermined paths and also the reverse paths while scheduling. At the beginning of scheduling each time slot all these memories are reset. Size of this memory is N x N.
- Rateleft - This memory is used to store the rate information of the groups on the left side. Size of this memory is N x log(N).
- Rateright - This memory is used to store the rate information of the groups on the right side. Size of this memory is N x log(N).
- Pred left - This memory is used to store the predecessor information of the groups on the left. Size of this memory is N x N. This memory is written through the row and accessed through the column and a priority encoder is used to choose the predecessor from the column. This speeds up the process as the way the algorithm works, we trace backwards and the current node can have multiple paths to choose from like the one in Figure 3c). The current node can be a predecessor of more than 1 node, like A is the predecessor of A and B. But when we update the memories in the end (after the augmenting path is found), we update forward and hence need to lookup in the other direction and this memory helps us in doing that in less time.
- Pred right - This memory is used to store the predecessor information of the groups on the right. This memory is similar to pred left. Size of this memory is N x N.

For a schedule with $G = 40$ and $N = 640$ we need 24K bits of memory for implementing this algorithm.

B. Core for LG and LL schedules

For a given matrix to be scheduled there are exactly P 1's in each row and column, A total of P*P 1's in the whole matrix. Often P is much less than the total number of linecards(N). So instead of giving a memory which has bunch of 0's and few 1's as an input, the input to this hardware is assumed to be only the row and column which has a 1. A processor is assumed to fill in the InOut memory which has P*P entries and each entry stores the row and column that need to be scheduled. Efficient bitmap techniques are used here as in the GG schedule implementation. The following memories are used in the design

- Permutation Memory - This memory is used to store the information regarding the connections between the inputs

and outputs and the permutation they are connected in. Lookup is done based on the input and the permutation ,i.e, we can get the output to which the given input is connected to in a given permutation. This memory is P*N (permutations*inputs) long and log(N) wide.
- Permutation Transpose Memory - This is similar to the permutation memory, but the lookup is based on output and the permutation ,i.e, we can get the input to which the given output is connected to in a given permutation. This is necessary as often we need to know which output is connected to which input in a given permutation and the Permutation Memory only gives us information regarding the output given the input and the permutation and not the other way around.
- InOut Memory - This memory is updated by the processor and used to store the entries that need to be scheduled (which have a 1 in the main matrix). Size of this memory is P*P x 2*log(N).
- Track Memory - This memory is similar to the InOut memory, but used to store the entries of InOut memory which cannot be scheduled by the greedy approach ,i.e, this serves as a InOut memory for the back tracing stage. Size of this memory is P*P x 2*log(N).
- Input Memory - This memory is used to store the information regarding the availability of the inputs for a given permutation. This is organized as P x N. It is used extensively is scheduling.
- Output Memory - This memory is used to store the information regarding the availability of the outputs for a given permutation. This is organized as P x N. It is used extensively is scheduling.

For a schedule with $G = 40$, $P = 16$ and $N = 640$ we need 230K bits of memory for implementing this algorithm.

VII. Results

Lot of time has been spent initially to identify the algorithms that can give best possible results in hardware. All these ideas have been tested with C programs for knowing their feasibility. The algorithms explained in the previous sections are identified to be the ones providing best possible results.

A. Synthesis

The hardware implementations mentioned in the previous section are implemented using verilog and synthesized using Xilinx tool set. The critical path has been identified as the priority encoder in both the implementations. Both the algorithms are synthesized using the XC2VP2 device of the XILINX VITEX2P device family. The critical path in ggschedule was taking 7.363ns whereas the critical path in the core for LG schedule was taking 6.706ns. Since this is implemented on a basic version of XILINX tools and devices, using a better process (0.13u technology) should get us below 4ns cycle time.

B. Simulations

Since it takes a long time to simulate verilog and get different sets of results for different number of linecards and groups over multiple iterations, C-models for the verilog implementations are developed in the end to produce the results. The C-models are made as close as possible to verilog in terms of the clock cycles it reports. The C-models are tweaked to provide worse results than the verilog. Different sets of results over 1000 iterations with different ranges of linecards spread over 40 groups are generated. The worst case and best case times for different stages are obtained.

The total time taken by the algorithm can be divided into

1) GG Greedy Time - Time spent in the greedy part of the GG schedule algorithm.
2) GG Back Tracing Time - Time spent in the back-tracing part of the GG schedule algorithm.
3) GG Update Time - The entire GG schedule algorithm was implemented in hardware. There is a significant amount of time spent in updating the memories before beginning to schedule for the next time slot.
4) LG Greedy Time - Time spent in the greedy part of the LG schedule algorithm.
5) LG Back Tracing Time - Time spent in the back-tracing part of the LG schedule algorithm.
6) LL Greedy Time - Time spent in the greedy part of the LL schedule algorithm.
7) LL Back Tracing Time - Time spent in the back-tracing part of the LL schedule algorithm.

A processor is assumed to be connected to the cores for some preprocessing and uploading the necessary information into the memories and also for retrieving the information from the cores and post processing it. The uploading and downloading time of the processor are not considered in the results. Figure 4 shows the breakdown of the time spent. This result was generated from the worst case results of the 600-640 linecards range. The result shows that most of the time is spent in backtracing of each phase and update phase of GG. LG schedule can be can be computed in parallel (Assuming multiple hardware blocks implementing the algorithm). For example each of the rows of GG schedule can be converted into the rows of LG schedule at the same time. The parallelism here is equal to the number of groups. Same is true with LL schedule. Figure 5 shows the best, worst and average times for different line card ranges when LG and LL schedules are done serially. Figure 6 shows the best, worst and average times for different line card ranges when LG and LL schedules are done in parallel. Both the figures assume a 4ns cycle time. It is clear from Figure 6 that the 50ms target is achieved. A simple linear relationship rather than an drastic exponential relationship between the number of linecards and the time taken can be observed from the Figures.

VIII. Conclusions

In this paper, we have presented and tested the algorithms that can be used to configure the switch fabric used in [1]. Feasibility study has been done initially to survey different modifications to the algorithms presented in [1]. The modified algorithms and some new algorithms oriented towards hardware implementation are briefly discussed in the previous sections. Also the core algorithms are implemented using FPGA's. C-models similar to the verilog are developed to simulate the results over different number of linecards and groups. Finally the

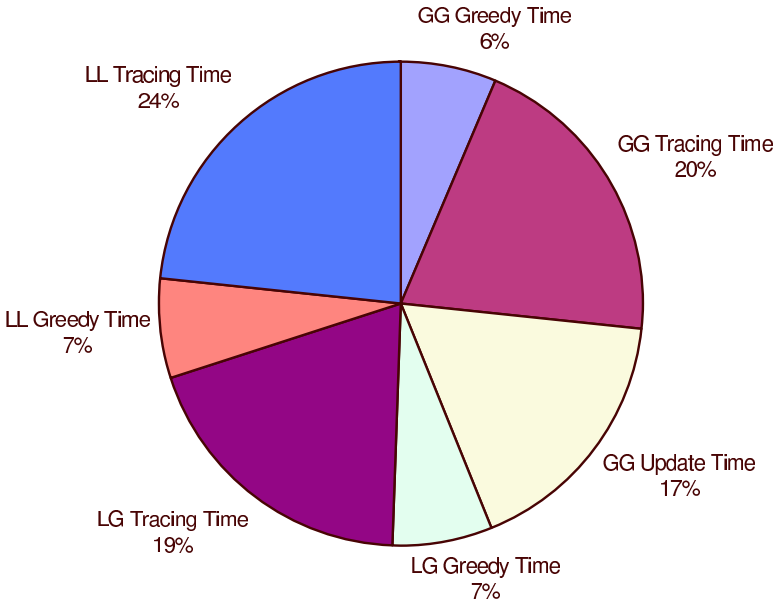

Fig. 4. Breakdown of the time consumed in different phases

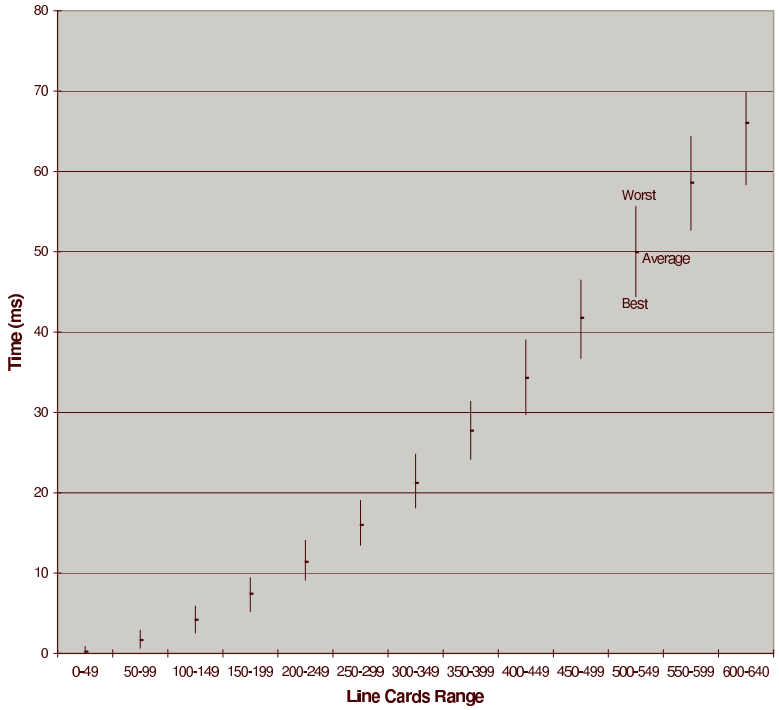

Fig. 5. Total time (serial LG and LL) Vs Linecards assuming 4ns clock cycle

results show that it is possible to achieve the 50ms target for scheduling the linecards.

IX. Future Work

In addition to the improvements and techniques presented in this paper, further improvements can be made. Time spent in memory accesses can be reduced by greater than two fold using multiport memories. Greedy approaches can be improved by exploiting some of the parallelism in implementing them. More pipelining techniques can be used in implementing the hardware which can approximately double the speed of the hardware. Time for uploading the information into the core needs to be estimated.

X. Acknowledgments

I would like to thank Professor McKeown for giving me the opportunity to work on this project. I would like to thank Da for his valuable discussions. I would also like to thank Isaac for his suggestions.

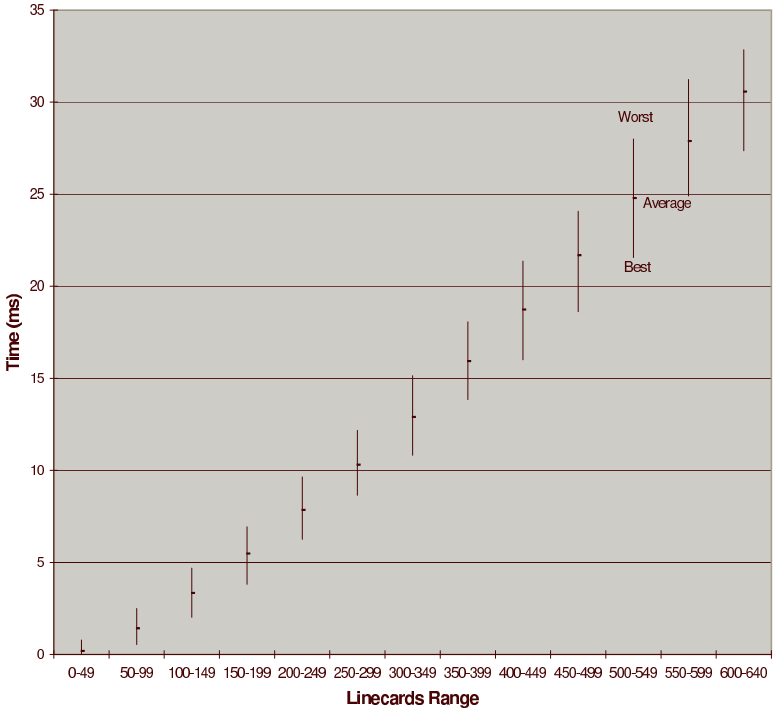

Fig. 6. Total time (parallel LG and LL) Vs Linecards assuming 4ns clock cycle

REFERENCES

[1] Isaac Keslassy, Shang-Tse Chuang, Nick McKeown, "A Load-Balanced Switch with an Arbitrary Number of Linecards," *Proceedings of IEEE Infocom '04*, Hong Kong, March 2004.

[2] Isaac Keslassy, Shang-Tse Chuang, Kyoungsik Yu, David Miller, Mark Horowitz, Olav Solgaard, Nick McKeown, "Scaling Internet routers using optics," *ACM SIGCOMM 2003*, Karlsruhe, Germany, Sep. 2003.

[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] P. Bernasconi, C. Doerr, C. Dragone, M. Capuzzo, E. Laskowski and A. Paunescu, "Large N x N waveguide grating routers," *Journal of Lightwave Technology*, Vol. 18, No. 7, pp. 985-991, July 2000.

[5] 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.

[6] G. D. Birkhoff, "Tres observaciones sobre el algebra lineal," *Universidad Nacional de Tucuman Revista*, Serie A, vol. 5, pp. 147-151, 1946.

[7] R. Cole, K. Ost and S. Schirra, "Edge-coloring bipartite multigraphs in O(E log D) time," *Combinatorica*, vol. 21, pp. 5-12, 2001.

[8] 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.

[9] A. Schrijver, "Bipartite edge-coloring in O(Δm) time," *SIAM J. Comput.*, vol. 28, pp. 841-846, 1999.

[10] N. Alon, "A simple algorithm for edge-coloring bipartite multigraphs," *Information Processing Letters*, vol. 85, issue 6, pp. 301-302, March 2003.

[11] Switch configuration algorithm, available at http://yuba.stanford.edu/or/SwitchConfig.c

[12] A. Schrijver, "A course in combinatorial optimization," available at http://www.cwi.nl/~lex/files/dict.ps, Feb. 2003.

[13] L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

[14] J. Hui, *Switching and Traffic Theory for Integrated Broadband Networks*, Boston: Kluwer Academic Publishers, 1990.