Clos Networks, Beneš Networks, and Introduction to Routing Algorithms

Lecture #5: Wednesday, April 18, 2001
Lecturer: Prof. Bill Dally
Scribe: Diana Pang, Alexandre Solomatnikov, and James Hsu
Reviewer: Kelly Shaw

1 Clos Rearrangable Non-blocking Network

For an n x m Clos network, the network is rearrangeably non-blocking when the output ports $m \geq n$. When this condition is true, all the incoming calls can be routed. The routing example we will analyze is for a (m, n, r) rearrangable network with $m = 3$, $n = 3$ and $r = 4$.

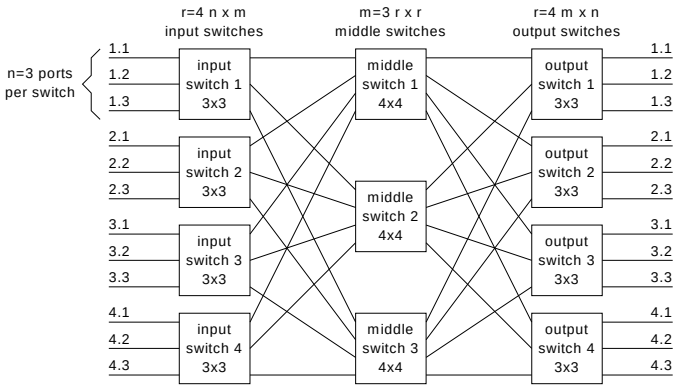

Figure 1: (3,3,4) Clos network used in our example.

For the 3-stage Clos network, the traffic pattern is as described in the table below:

Input switch	Output switches
1	2, 3, 4
2	2, 4, 1
3	3, 4, 1
4	1, 3, 2

Table 1: Traffic pattern for Clos network example.

In this problem, the only free variable for routing is which middle switch to go through, so we only worry about the middle stage switch assignments. This is a bipartite graph coloring problem. We assign different colors to each of the middle stage switches, with the color of an edge routing between a pair of input and output switches being the same as that of the middle stage switch it is going through. For this problem, we color the three middle stage switches as red, blue and green. (Note: for convenience, will use I# for input switch number, O# for output switch number, and P# for port number, all the input ports are on the first stage and all the output ports are on the third stage. We will also use Red to denote middle switch 1, Blue for middle switch 2 and Green middle switch 3.

			Middle		Input Free				Output Free			
Step #	In	Out	New	Old	1	2	3	4	1	2	3	4
1	3	1	1		111	111	011	111	011	111	111	111
2	2	1	2		111	101	011	111	001	111	111	111
3	3	3	2		111	101	001	111	001	111	101	111
4	3	4	3		111	101	000	111	001	111	101	110
5	1	4	1		011	101	000	111	001	111	101	010
6	4	2	1		011	101	000	011	001	011	101	010
7	4	1	3		011	101	000	010	000	011	101	010
8	2	4	1		011	001	000	010	000	011	101	010
9	1	4	2	1	101	001	000	010	000	011	101	000
10	1	2	3		100	001	000	010	000	010	101	000
11	4	3	2		100	001	000	000	000	010	101	000
12	3	3	1	2	100	001	010	000	000	010	001	000
13	3	1	2	1	100	001	000	000	100	010	001	000
14	2	1	1	2	100	011	000	000	000	010	001	000
15	2	4	2	1	100	001	000	000	000	010	001	100
16	1	4	1	2	010	001	000	000	000	010	001	000
17	1	3	2		000	001	000	000	000	010	001	000
18	4	3	3	2	000	001	000	010	000	010	000	000
19	4	1	2	3	000	001	000	000	001	010	000	000
20	3	1	3	2	000	001	010	000	000	010	000	000
21	3	4	2	3	000	001	000	000	000	010	000	001
22	2	4	3	2	000	010	000	000	000	010	000	000
23	2	2	2		000	000	000	000	000	000	000	000

Table 2: Routing Table for Clos network example.

Now, as an example, let's walk through Table 2 above and set up routes between switches. Please refer to this table as well as the Figures 2, 3, and 4 as we go through this example. We need to set up the connections so that all the edges connecting one vertex of the input/output switch have different colors. In other words, they will get routed through different middle stage switches.

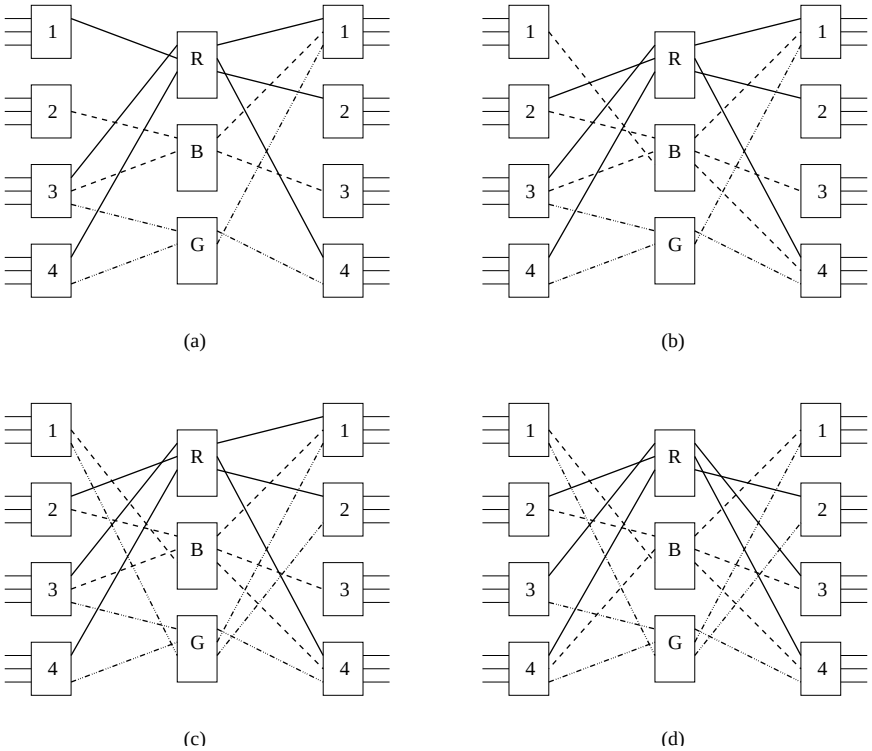

Figure 2: Edge Coloring Example; (a) link state up to step 8 of Table 1, (b) link state up to step 10 after a flip has occured, (c) link state up to step 11 of Table 1, (d) link state up to step 13 of Table 1 after another flip has occured.

Refer to Figure 2 above for the link diagrams corresponding to different points of time while going through each step in Table 2. Figure 2(a) corresponds to the links that are created in steps 1-7 in the table. Figure 2(b) shows the state after step 9 in the table where a one link flip has occured to accomodate for the I2 to O4 connection through the Red middle switch. The I1 to O4 connection had to be re-routed from the Red middle switch to the Blue middle switch. Figure 2(c) covers the link state up to step 11 in the table, where no other flips have been necessary yet. Figure 2(d) shows the state after more flips are needed in steps 12 and 13. If we were to continue working this example out, we would continue flipping edges until we have reached a point where all the links have been accomodated.

The section below shows each of the steps in more detail:
1. 3 to 1:
All the output ports from I3 are free and all the input ports to O1 are free (If Table 2 were to have a row before the first row, it would be 111 for both Input Free 3 and Output Free 1). I3 is connected to O1 via middle switch 1. The edge is colored Red.

2. 2 to 1:
All the ports from I2 are free (111). P1 is not free for O1 (011), so I2 is routed to O1 via middle switch 2. So the edge is colored Blue.
3. 3 to 3:
Similarly, they are connected through middle switch 2 and the edge is colored Blue.
4. 3 to 4:
Connected through middle switch 3 and the edge is colored Green.
5. 1 to 4:
Connected through middle switch 1 and the edge is colored Red.
6. 4 to 2:
Connected through middle switch 1 and the edge is colored Red.
7. 4 to 1:
The only free input port on O1 is P3, so the connection is done through middle switch 3 and the edge is colored Green.
8. 2 to 4:
The free ports on I2 are P1 and P3 while the only free port on O4 is P2. We can't make a route between the two directly. We need to reroute the connection of 1 to 4 and use the P1 of I2 to connect to O4.

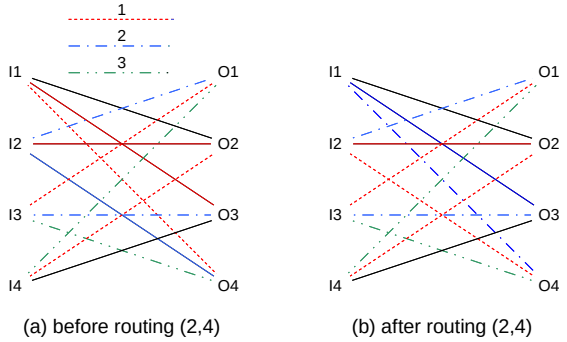

Figure 3: Stages of routing the call set in the Example. Assignment to middle stages is denoted by the *dotting* of the edges. (a) Before routing call (Input 2, Output 4), there are no middle switches free on both Input 2 and Output 4. (b) To route this call, (Input 1, Output 4) is moved to middle switch 2 allowing (Input 2, Output 4) to use middle switch 1. (From Dally's Chapter Notes).

---

9. 1 to 4: (rerouted)
Since P2 is available for both I1 and O4, we can use these ports for this connection along edge Blue.
10. 1 to 2:

Connected through middle switch 3 and the edge is colored Green.

11. 4 to 3:

The two switches don't have common free ports, so we'll have to reroute. We will connect I4 to O3 through P2 (blue) because this is the only port free on I4. This connection breaks the I3 to O3 connection made earlier.

12. 3 to 3: (rerouted)

From the previous step, the I3 to O3 connection through P2 was severed. Thus, we must reconnect I3 to O3 using another middle stage. We see that P2 (red) is the first free port on O3 amd so we make the connection through this port. Unfortunately, this conflicts with the I3 to O1 connection we made earlier. The I3 to O1 connection will have to be rerouted in the following step to accomodate the I3 to O3 connection.

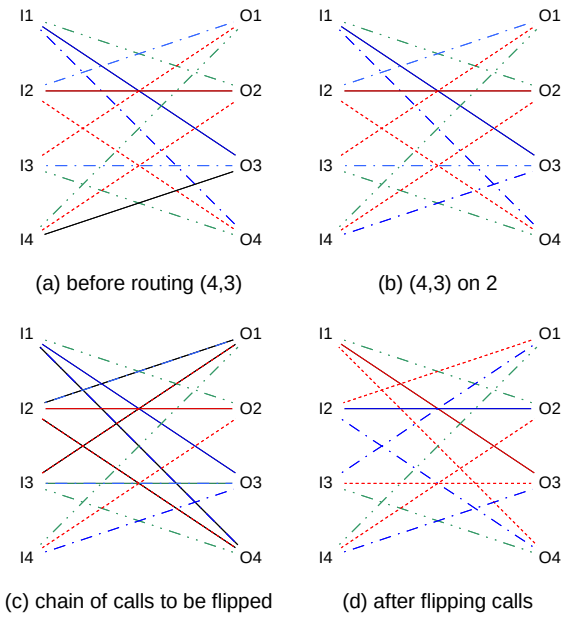

Figure 4: Stages of routing the call set in the Example. Assigment to middle stages is denoted by the *dotting* of the edges. (a) Before routing call (Input 4, Output 3), there are no middle switches free on both Input 4 and Output 3. (b) The call is routed on middle switch 2, creating a conflict with call (Input 3, Output 3). (c) Moving (Input 3, Output 3) to switch 1 starts a chain of conflicting calls (Input 3, Output 3), (Input 3, Output 1), (Input 2, Output 1), (Input 2, Output 4), and (Input 1, Output 4) that must be *flipped* between switches 1 and 2 to resolve the conflict. (d) After flipping these calls, no further conflicts exist. (From Dally's Chapter Notes).

From this point on, we flip the colors of the links as we traverse through the connec-

tions. As we go in the direction from input to output, we flip the red links to blue, and as we move from output to input, we flip the blue links to red until we find an empty port. In this case, it is the empty red port on input switch 1. At the end of the first 12 steps, the link state can be described by a bipartite graph as shown in Figure 5.

Let's look in depth at the algorithm for rearrangement that we use at step 8 in our example. The algorithm is as follows:
1. find the first free ports on both I1 and O4, which would be P1.
2. Use P1 of I2 to connect O4 using middle switch 1. Uncolor the route of I1 to O4 which was Red. Color the route from I2 to O4 as Red.
3. Reroute the connection I1 to O4 through middle switch 2 (since the first free port on O4 is P2).

The reroute in the algorithm above was an easy one because P2 of I1 happened to be free when we moved the connection through the blue link. What if the blue link on I1 had been occupied? If this had been so, we would just have to tear up the blue link that is being used by some other connection and put the I1 to O4 route onto it. Then we'd reroute the newly broken link using the red link. It is guaranteed that there will be a free port leading to the red middle switch since we ripped up the I1 to O4 link on red earlier. Therefore, the red link will be available at this point.

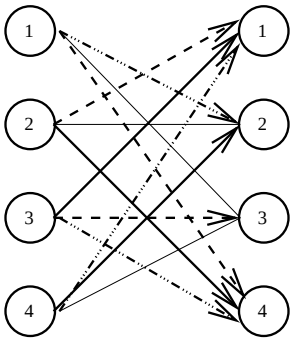

Figure 5: Set of circuits to be routed on the Clos Network after step 12 of the example.

The key to this algorithm is that we only swap between the two middle switches Red and Blue among the links to be rearranged. All the links through the rest of the middle stage switches stay the same. This is trivial for the example we are using with only 3 middle stage switches. For a Clos network with hundreds of middle stage switches, however, it gets very complicated. How do we know that the algorithm is going to terminate? You can show that once an input or output port is visited, it will never be revisited on the same colored port using this algorithm.

The *looping* algorithm on Page 84 of the Chapter Notes summarizes the steps taken to route unicast traffic on a rearrangeable Clos network. For a connection between two switches a and b, we first find out whether there is a middle switch free to connect the two. If so, we make the connection by picking one of the free middle switches. Otherwise, we find mida and midb. Then we link a to b using mida. Finally, we identify the existing call c to b which uses mida and reroute it through midb.

The time taken to rearrange the network is bounded by the number of ports. Since the algorithm is set up so that it never revisits a particular vertex, it will complete its iterations after visiting at most $2r$ vertices. In fact, at most $2r - 2$ calls are rearranged, where r is the number of input/output ports of the middle switches. This is because the first port where the rerarranging starts and the last port where the algorithm terminates doesn't need to set up links (thus accounting for the subtraction of 2). This ensures that even if the rearranging algorithm touches all the nodes of the Clos network, any permutation will be rearranged in finite steps of execution.

The above algorithm is often used in practice. There are other algorithms for rearranging a Clos network. The Chapter notes present another algorithm that uses matrix decomposition. It partitions the network upfront without backtracking along the existing connections. In reality, calls arrive and leave incrementally, and ripping up/rerouting links has to be performed continuously. The bipartite graph coloring algorithm is better suited for this situation. Otherwise if you know ahead of time all the connections that you have to make, you can set up the links up front. Is the algorithm minimum in some sense? It is in the sense that you don't need to move those links that you don't have to rearrange.

2 Clos Networks With More Than Three Stages

How do you build 64 x 64 rearrangeably non-blocking network given only 4 x 4 switches?

First level of switches has sixteen 4 x 4 switches. Each of them has 4 outputs, which should be connected to 4 different middle switches. Thus, the middle level must have four 16 x 16 switches. The third level of switches is similar to the first level. Now the problem is reduced to a previously solved problem.

How do we build 16 x 16 switches? We already know how to create a 16 x 16, 3-stage Clos Network using 4 x 4 switches. Also, we were able to build a 64 x 64 network by plugging in a 16 x 16 network for each middle switch. If we want to build 256 x 256 network we can do a similar thing. The result is a network with an odd number of stages: 16 x 16 has 3 stages, 64 x 64 has 5 stages, 256 x 256 has 7 stages, etc.

What about scheduling? How do we set up (route) a call from a particular input to a particular output? The question in the 16 x 16 network was: what is the middle stage? In the 64 x 64 network the question is: what is the big middle stage? First, we should determine which big middle stage could be used. If the first and last switches have unused channels of the same color (i.e. free channels to the same middle stage), then the appropriate middle switch should be used. However, if there are no such channels,

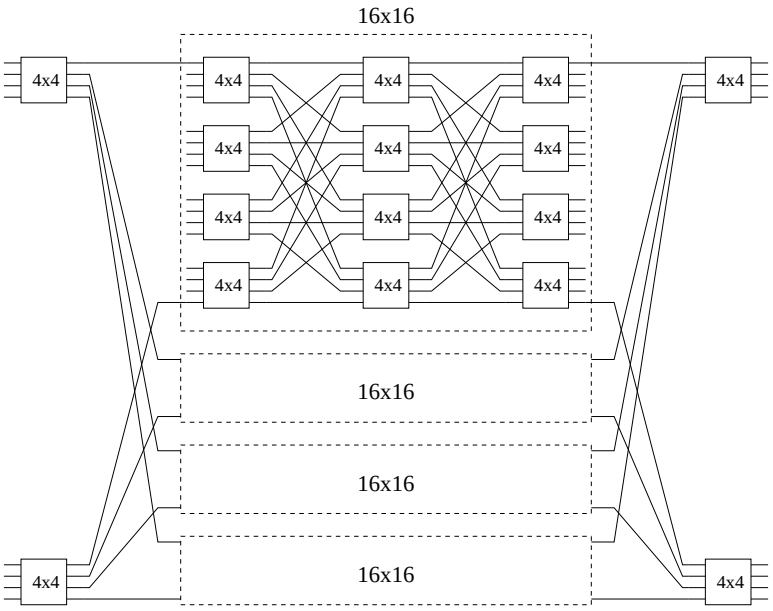

Figure 6: A 64 x 64 rearrangeably non-blocking network.

then rearrangements should be done to free channels to the same middle stage. After the outer level is defined, the next step is to determine how to route a call inside the big middle stage.

Next question: what if I insist that the network should be strictly non-blocking? How should we redimension the stages? How many stages do we need? (The same example of 64 x 64 network).

How many middle switches do I need inside the big middle switches? $2n$-1 = 7. First level switches should be 4 x 7 and the last level switches should be 7 x 4 switches. The number of big middle switches should also be $2n$-1 = 7. The total number of 4 x 4 switches in the middle of the network is now 49 instead of 16 as before.

In a 5-level network the number of small middle switches will be $(2n\text{-}1)^2$ for a strictly non-blocking network instead of n^2 for the rearrangably non-blocking network. In a 7-level strictly non-blocking network there will be $(2n\text{-}1)^3$ small middle switches instead of n^3 for the rearrangably non-blocking network.

In general, the number of small middle switches will be $(2n\text{-}1)^i$ in the strictly non-blocking network versus n^i in the rearrangably non-blocking network.

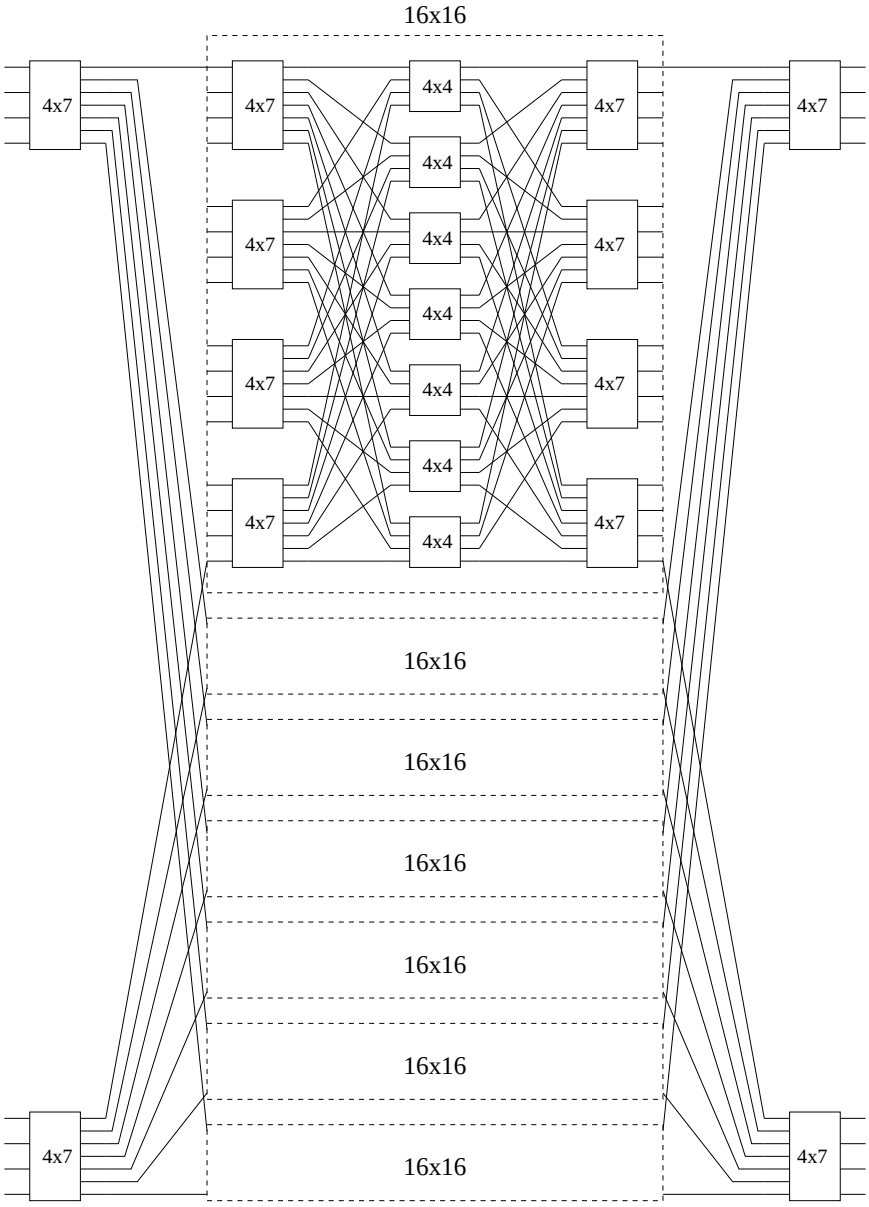

Figure 7: A 64 x 64 strictly non-blocking network.

3 Beneš Networks

A special case of Clos networks is one made of 2 x 2 switches, called a Beneš network. It is a 2-flies back-to-back network with two middle stages fused into one stage (in fact, all Clos networks are back-to-back butterfly networks).

Why is it an interesting network? Why is it interesting to build with 2 x 2 switches?

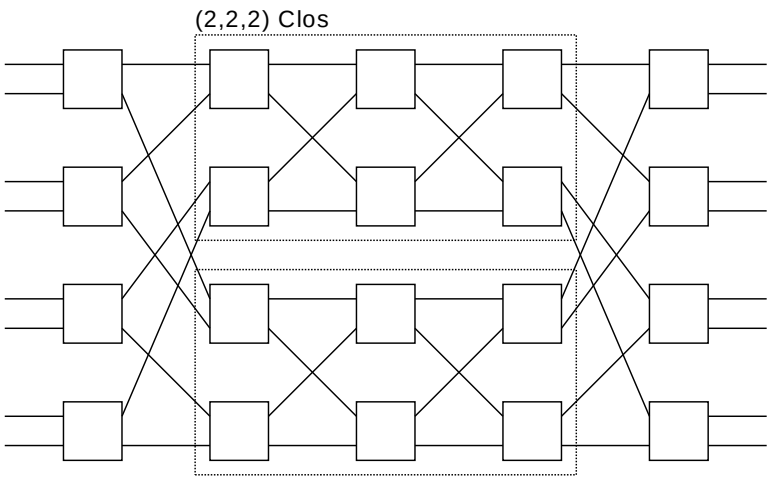

Figure 8: A rearrangeable Clos network is constructed using two (2,2,2) Clos networks as 4 x 4 middle switches. A Clos network such as this is also referred to as a Beneš network. (From Dally's Chapter Notes)

Beneš networks have the minimal number of cross points and this property was considered very important some time ago when the cost of the system was determined primarily by the number of switches used. For N x N switches the cost was N^2. If the size of switches was increased, then the cost increased as square. If instead the number of stages was increased, then the cost increased logarithmically in some sense.

This is not true anymore because transistors are cheap while pins or ports are expensive. So the result is now exactly the opposite - the fewer stages - the fewer pins that are necessary for each input - the lower the cost. This is because you will need 2 pins for input and output at each stage, and thus, the number of pins grows proportionally with the number of stages. That's why today designers build Clos networks with maximum size switches. They want a larger number of crosspoints which reduces the number of stages.

The final thing about Clos and Beneš networks is they have an axis of symmetry. That suggests a particular way of packaging such networks if input and output points are the same. They can be folded along the axis of symmetry and the first and the fifth stages can share the same package because the links are exactly symmetrical. The second and the fourth stage (from Fig. 6.17) can share the package. The third stage is packaged by itself. A drawback to such packaging (unless there is simultaneous bidirectional signaling) is that now the network should be built from smaller switches because the switches have to share the number of pins on a package.

However, this packaging has a nice property if, for example, the first stage is 4 x 7

switch and the last stage is 7 x 4 network. The total number of inputs and outputs is the same: $4 + 7 = 11$. It can be implemented as one 11 x 11 square switch.

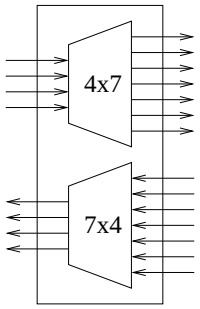

Figure 9: An 11 x 11 square switch.

When there are rearrangably non-blocking networks the routing decisions are global in a sense that they impact all calls in the network. The speed of the decision is very important when there are disruptions on some of these channels and they should be rearranged within 50 ms. There are two approaches to this problem. One approach is to do everything in one centralized scheduler. It gets requests for all new calls and requests for call disconnections and runs the algorithm we did by hand, computes call setups and programs all switches.

The other approach is to partially distribute the algorithm because the algorithm works from the outside in (for more than 3 stage network). There are global resources but they are global pair-wise in a sense that two ports can negotiate. The computing can be associated with the individual ports: they can send messages back and forth to rearrange the global level. Similarly, subswitches can be rearranged globally or rearrangement can be distributed. In general, designers prefer a centralized approach because it's simpler, and then they move to a distributed approach if they cannot meet specifications for rearrangement time because in the distributed case many rearrangements can be performed simultaneously.

If there are more stages, the algorithm does not get more complicated but the order of algorithm increases (as n^3 in the worst case chain of things to be rearranged). The algorithm run time goes up in the worst case. Most designers design for the worst-case scenario: the worst-case arrangement and when a new call comes in the maximum number of links should be rearranged. The average case is much better.

Is routing more complicated on VLSI chips? Today's chip can have, for example, 512 inputs and 512 outputs but the number of wires in one direction on one layer is about 20,000. You can route a much bigger switch than you can get connections off the chip. That's why designers don't worry about the size of the switches and number of cross points, they worry about pin number.

What happens to the critical path if the switches get larger? On one hand the delay of the switch increases but on the other hand the number of switches (hops) on the critical path decreases. It's not clear how delay changes. Big Clos networks are used in circuit switches of backbone networks in which the delay is not important (within reasonable limits) as long as it is constant because the delay of the switch is small in comparison with delay of transcontinental fiber.

4 Sorting Networks

Now let's consider a completely different way of building non-blocking networks. Assume there is a box which can sort and it has 8 input and 8 outputs. Suppose we have a permutation to route. How can we view it as a sorting problem? Let's consider particular permutation: 5, 2, 3, 1, 4, 0, 6, 7. Now we put this permutation into the sorting network.

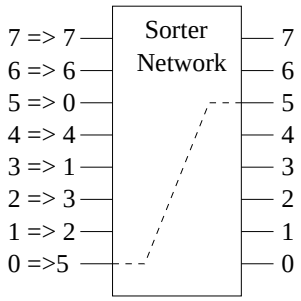

Figure 10: Simple sorting network with connections chosen in class.

Are sorting networks strictly non-blocking or rearrangably non-blocking? There are different types of sorting. Most sorting networks are rearrangably non-blocking

First, we need to start with some definitions. A Monotonic sequence is one that's either only increasing or decreasing. A Bitonic sequence has part only increasing or decreasing and part that's going the other way. It's very easy to merge a bitonic sequence into a monotonic sequence. Slice it in half and compare each point on one side to the corresponding point on the other half. What that yields is two bitonic sequences. Keep doing this until you get down to length 2 and do it once more to get a monotonic sequence.

The way to think about a bitonic sorting network is to assume bitonic sequences of length 2 and make it monotonic in a particular direction. What is really needed is a sorting box like the one in Figure 7(a). What we want is to make a bitonic sequence of length 4, so we use two sorting boxes back-to-back as in Figure 7(b). We then want to merge those into a 2-long bitonic sequence so that we can ultimately merge them into a monotonic sequence.

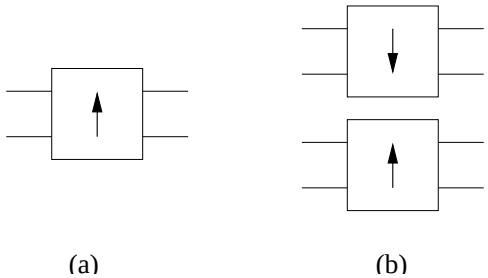

Figure 11: Sorting network building blocks; (a) sorting box puts larger number on top, (b) combination produces a bitonic sequence of length 4.

It takes one stage to merge unit sequences into a 2-long sequence. It takes 2 stages to merge the two 2-long sequences into a 4-long sequence. The first stage gives a sequence of length 2, the second stage gives sequence of length 4, and the third stage gives sequence of length 8... to get a sequence of length N, you need:

$$\sum_{i=1}^{lg(N)} i = \frac{(lg(N))^2 + lg(N)}{2}, \text{ (where lg is base 2)}$$

So a sequence of length 8 takes a 6-stage network, and a sequence of length 16 takes a 10-stage network.

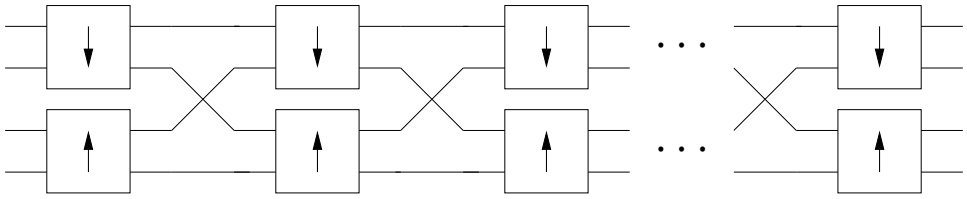

Figure 12: Number of stages required in sorting network relates to length of sequence: $[(lg(N))^2 + lg(N)]/2$.

Nobody uses sorting networks for building circuit switches because the cost is roughly proportional to the number of stages. If you want to connect 8 nodes together and you have 2 x 2 switches we can do it with a 5-stage Clos network, while it takes a 6-stage Batcher Sorting network. With a sorting network, you are growing with $[(lg(N))^2]/2$ and with a Clos, you are growing with $2lg(N)$. One nice property of sorting networks is that they are self-routing, but you pay quadraticly for that in the limit.

5 Routing

Going back to discussion on the *roadmap* we look at a 2 x 4 torus. The topology tells us the throughput you can get through the network. From topology we compute γ_c, and from γ_c:

$$\Theta = \frac{b}{\gamma_c}$$

This is throughput you will never exceed. A routing algorithm takes a particular traffic matrix and assigns the traffic to edges of topology in such a way that the best it can do is load channels to γ_c. A bad routing algorithm can do a lot worse.

For example, let's pretend we have a large mesh network and our traffic pattern is transpose, which means every node needs to send traffic with respect to an axis of symmetry. If the axis of symmetry is a diagonal (as in Figure 9), everybody in the top row goes to their respective mirrored node in the last column.

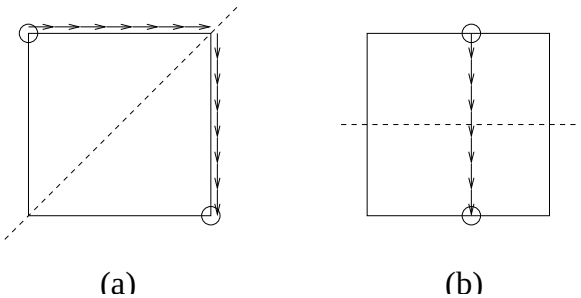

Figure 13: Transpose routing algorithms; (a) axis of symmetry along diagonal, $\gamma = 15$, (b) axis of symmetry along horizontal bisection, $\gamma = 4$.

Also suppose that the routing algorithm is dimension ordered. Everybody is routed first in x and then in y. Let's also say that k=16 in this network. So all 15 nodes (left of the rightmost top node) at the top go along the single channel connecting node 15 to node 16. Similarly, all 15 rightmost nodes above the bottom corner node go over a single channel at the bottom and $\gamma_c = 15$.

Transpose is not a bad traffic pattern in terms of bisection bandwidth. If you slice the network in half in any other way, only half the traffic crosses the bisection. So with a k=16 network, the expected channel load would be equal to 4. This is what topology tells us it should be. But if we use a particularly bad routing algorithm on it, then it is nearly four times this value. Routing algorithms are all about load balance.

One taxonomy of routing has to do with the number of paths involved. With deterministic routing, there is exactly one route to use for every source and destination pair. This is bad since there is no path diversity and a link can be overloaded very easily because there are shared links.

When there are multiple paths between a source and destination, you can choose one of those paths in a number of ways. One method is to choose amongst them randomly. This is called oblivious routing. This is actually not a bad approach since you could be attempting to use information that's negatively correlated with what you want. You can also use information like which channels are the busiest, which have the biggest queues waiting for them - this is called adaptive routing.

The problem with adaptive routing is that you only have a limited amount of knowledge as to which path is the best from node to node. You only have information about the nearest nodes. This is analogous to driving in traffic on the freeway. You only know about the cars right around you at that moment. Because you don't have global information, you may be routed into a *traffic jam* two hops down. In fact, it's impractical in these networks to propogate global information around because if you look at the time dynamics of channel loads in typical networks, they change extremely rapidly. When a channel is overloaded, it's overloaded instantaneously, and then a couple packet-times later, that overload is gone. That's why trying to do adaptive routing can get you into a lot of trouble, because you may see a momentary hotspot which will force traffic in the other direction, which will cause a hotspot in the other path, and you will end up with oscillatory behavior. Oblivious routing seems not to have that problem.

Another taxonomy of routing has to do with where you do it. This is analogous to planning your trip across country. Picking a particular route that will never be deviated from - called *All-at-Once* routing. The most common of which is called source routing where the entire route is determined at the source. The advantage of source routing is that the rest of network just needs to look at simple encoding to select an output port. This is very important in networks where latency is a critical parameter. It's also is more flexible in routing as compared to hop-by-hop routing.

Hop-by-hop routing (or incremental routing), which tends to be memoryless, makes the decision at each node. You have the ability to incrementally adapt however, assuming you are able to use information without getting yourself into trouble as described earlier.

A third taxonomy of dividing up different routing algorithms is to say whether they are minimal or not. When a routing algorithm is minimal, there is a minimum distance between nodes. It's very easy to break minimal routing algorithms as described in a previous lecture. It's easy to do with a tornado pattern in a torus, for example, because if you insist on taking minimal routes , you'll only use half the channels. But you could essentially double your throughput if you allowed a fraction of the traffic to be sent the long way around.

The alternative is non-minimal. Some people have even proposed Brownian motion. With non-minimal routing, it's essential to prove that you will eventually get to your destination and not livelock.

The final taxonomy is whether to do the algorithms in a table driven or algorithmic routing algorithm. In table driven algorithm, you have complete flexibilty. You can either choose to do source routing or hop to hop. Which means you can use any topology you want. You really don't need to know anything about the topology when you design the

router. The problem with this is if you want to make the router simple and fast, you will end up needing a lot of RAM storage and the access time to deal with them.

The alternative to this is algorithmic routing and a great example of that is dimension ordered routing, where you just use a 1 comparator with the number of bits equal to lg(number of bits in that particular direction) to simply tell you if you have moved far enough in a direction. At this point, you turn the corner and go the other direction. This is an awful algorithm in terms of load balance. Algorithmic routing reduces complexity but locks you into a topology.