EE482B: Advanced Computer Organization **Lecture #9**
Interconnections Networks Architecture and Design
Stanford University

Lecture #9:	Wednesday, 2 May 2001
Lecturer:	Prof. Dally
Scribe:	Alok Kuchlous, Mayank Gupta
Reviewer:	Kelly Shaw

Throughput vs. Latency Curve:

Flit control reservation makes more efficient use of buffers and hence increases the saturation throughput with the same amount of buffering. This throughput can be achieved with virtual channel flow control, but with more buffers.

Figure 1 shows a throughput vs. latency curve for virtual channel flow control and flit reservation flow control. It shows that with same amount of buffering, throughput of flit reservation flow control is better than the throughput of virtual channel flow control.

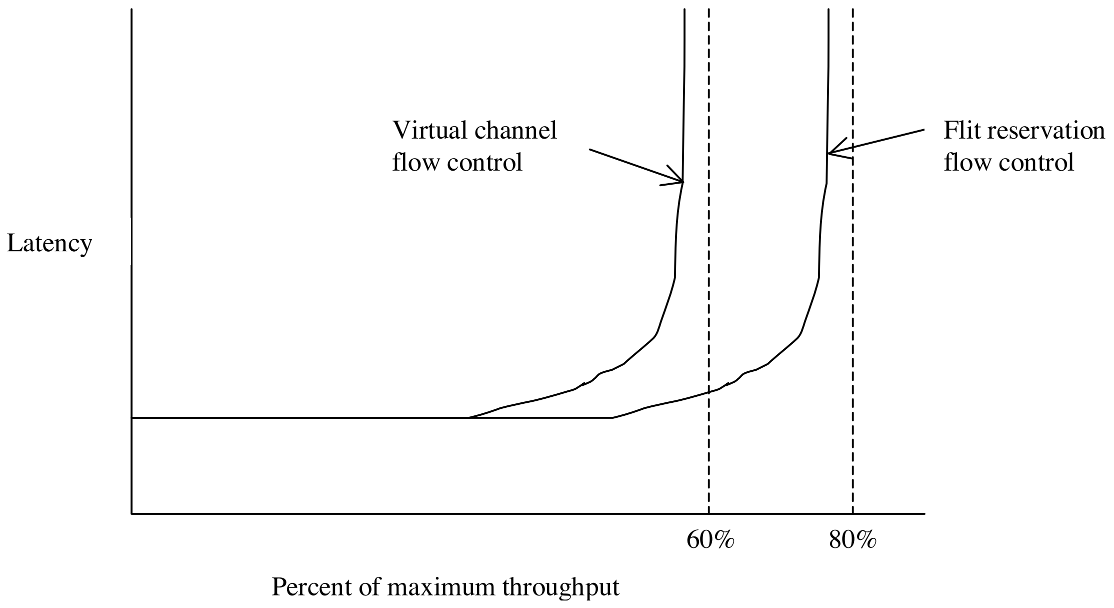

Figure 1: Throughput vs. Latency Curve

Deadlock

Definition: There is a subset of the network in which there is a set of packets and a set of resources but no progress is being made. The common cause for this situation is a cycle in list of packets that are waiting for resources and packets holding the resources. Figure 2 illustrates this situation. Deadlock can occur whenever there is a cyclic dependency for resources. These resources are generally buffers, but for some cases (flit level flow control) can be channels also.

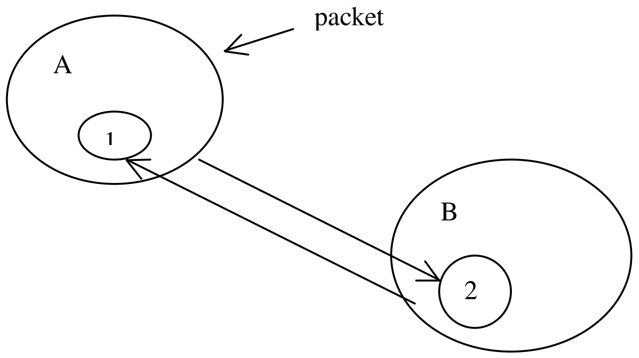

Figure 2: This network is deadlocked. A and B are packets. 1 and 2 are virtual channels and associated buffers

In figure 2, packet A is holding channel 1, packet B is holding channel 2. Packet A needs to acquire channel 2 and packet B needs to acquire channel 1. Thus there will be no progress in this network and deadlock will happen.

Dependencies: In figure 2, packet A holds channel 1, wants channel 2, i.e. there is a dependency between channel 1 and channel 2 due to packet A. The network in figure 2 is in deadlock because channel 1 depends on channel 2 and channel 2 depends on channel 1. (Dependency between channel 1 and channel 2 means that channel 1 can potentially wait on channel 2).

Dependency occurs when an agent is holding a resource and that agent tries to obtain another resource. In our example, agents are packets and resources are channels.

Another Deadlock Example

$$\begin{tikzcd} 1 \arrow[r, "12"] & 2 \arrow[d, "23"] \\ 4 \arrow[u, "41"] & 3 \arrow[l, "34"] \end{tikzcd}$$

Figure 3: 4 Node deadlock example

In figure 3, the circles 1, 2, 3, 4 are nodes, and the edges 12, 23, 34, 41 are channels. Channels here represent not only the physical wires connecting nodes, but also the buffer and control state at the node. For example Channel 12 is the physical wire connecting node 1 and node 2 and also the buffer and control state at the node 2 (assuming single flit buffer at each node).

Dependency Graph for 4 node example:

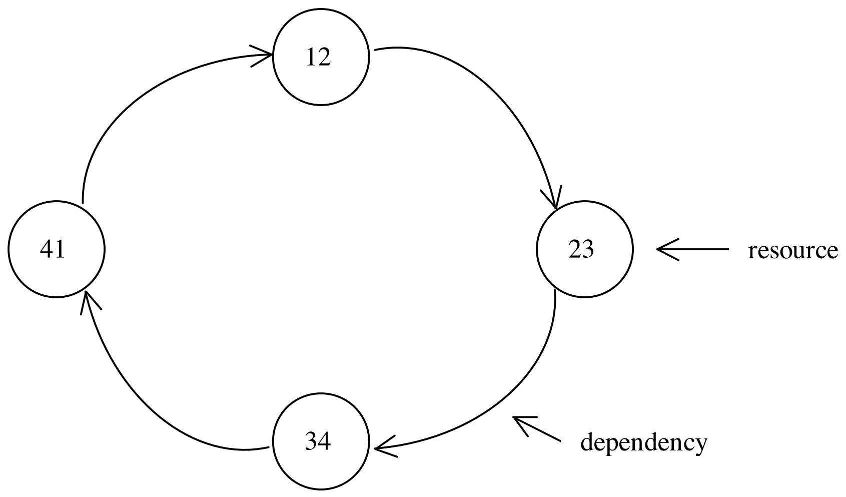

Figure 4: Dependency graph

Figure 4 is the dependency graph for the network in figure 3. In a dependency graph, nodes are resources and the edges indicate the dependencies. In the above figure, nodes 12, 23 etc. are channels.

If we want to send a packet from node 1 to node 3 (of figure 3), we will first acquire channel 12. Next, we need to acquire channel 23, without giving up channel 12. Thus there is a dependency between channel 12 and channel 23. By symmetry we have cyclic dependencies and we can have a deadlock. (If each node sends a packet, all the nodes will grab the first resource and will wait for next resource.)

Breaking a Deadlock

We can break a deadlock by eliminating the cycle, or by making the graph acyclic. In figure 4, we can eliminate the cycle by deleting an edge between node 34 and node 41. This means that if you are holding channel 34, you cannot wait for channel 41, i.e. if you reach node 4, you must be destined for it. Deletion of this edge results in no path from node 3 to node 1 via node 4. This leads to an unconnected graph. To keep the graph connected and acyclic, we can add another virtual channel between node 4 and node 1, and use this channel to go from node 3 to node 1 via node 4. Adding this one virtual channel between node 4 and node 1 keeps the graph completely connected and acyclic, as shown in figure 5a.

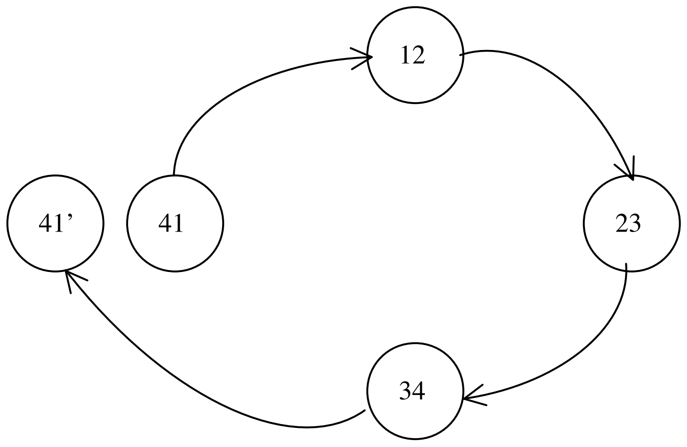

Figure 5a: Breaking a deadlock

One extra channel breaks the deadlock in this network if routes are not more than 2 hops, that is, we do not use long way routes for this network. One extra virtual channel is not enough for long way routes because we can not go from node 3 to node 2. We have to use channel 41' to get to node 1, but we can not go anywhere else because channel 41' can not be dependent on any other existing channel. Therefore, for long way routes, we will have to add second virtual channel between node 1 and node 2 (channel 12') as shown in figure 5b.

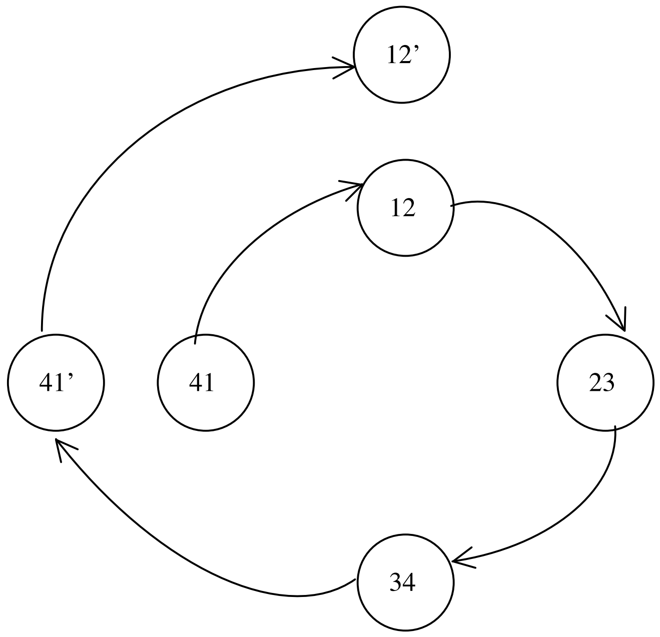

Figure 5b: Breaking a deadlock for long way routes

There is no need for extra virtual channels at the remaining nodes, but they may be added for better load balance. Any performance gain due to extra virtual channels is because of load balance.

Implementation

We can use one of the following relations to decode which output channel to use at node 4:

R : C X N |→ C -------- (r1)
R : N X N |→ C -------- (r2)

Here r1 takes the incoming channel and the destination node and produces an output channel while r2 takes the current node and the destination node and produces an output channel. To avoid unwanted dependencies between channels, we need to use r1, as we need to know the input channel to determine the output virtual channel. In figure 5b, at node 1 we need to know which input channel the packet arrived in order to route either on channel 12 or channel 12'.

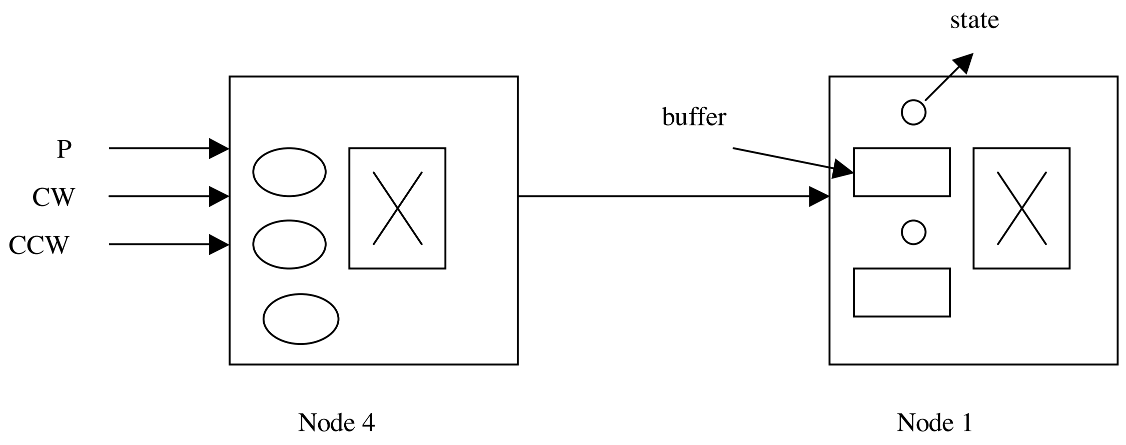

Figure 6: Implementation of an extra channel at node 4

There are three inputs. P: from the processor (or line card), CW: clock wise from node 3, CCW: counter clock wise from node 1. For flits on P, going to node 1, virtual channel 0 will be used. This corresponds to node 41 in the dependency graph (figure 5). For flits on CW, virtual channel 1 will be used, i.e. node 41' in the dependency graph. For flits going to node 3, virtual channels on the node 3 will be used.

Deadlock in store and forward flow control

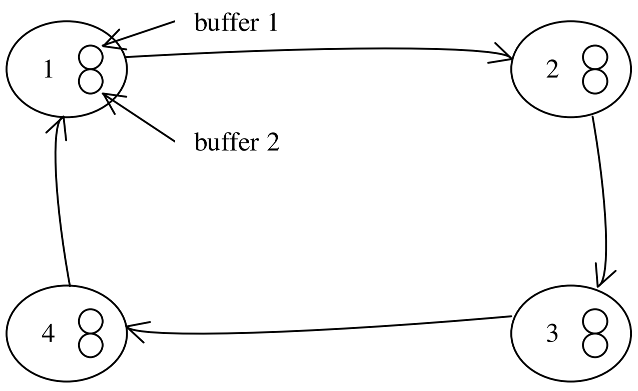

Figure 7: Deadlock in store and forward control

In store and forward flow control the resources are buffers. In the figure above, packet at node 1 needs a buffer at node 2 to release the buffer at node 1. Thus there is a dependency between buffer 1 and buffer 2.

To break the deadlock, we add more buffers as shown in figure 7 and specify rules about which buffers can forward to which buffers. The dependency graph for network in figure 7 is shown in figure 8. Nodes of this graphs are buffers and egdes indicate which buffer can forward to which buffer. For example Buffer 1' can only forward to buffer 2'.

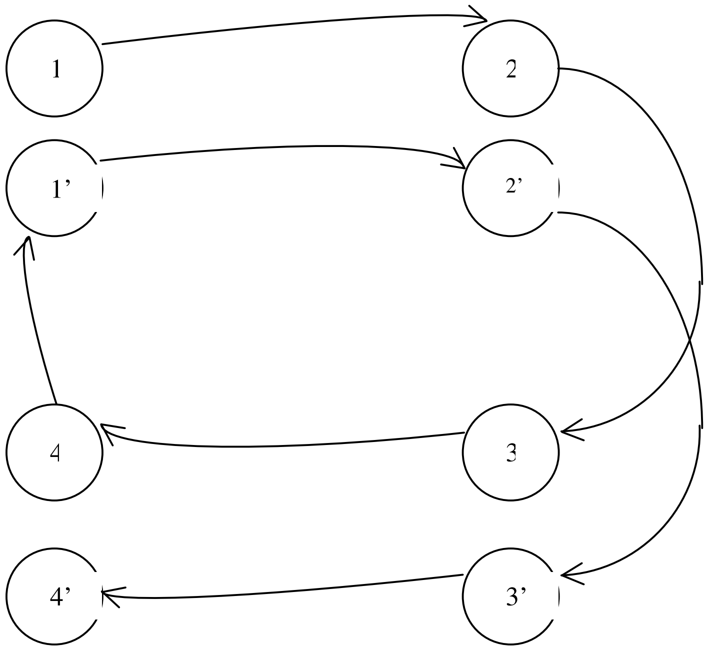

Figure 8: Dependency graph for store and forward flow control

Partial Order

Partial order on the resources is a way of avoiding cycle between resources. A partial order is relation like 'less than'. When a partial order applies, it gives a consistent ordering. It produces an acyclic directed graph, i.e. it is transitive.

Figure 9 shows a partial order between A, B and C and between D and C. Here A is less than B, B is less than C, therefore A is less than C. D is also less than C, but relationship between A, B, and D is undefined.

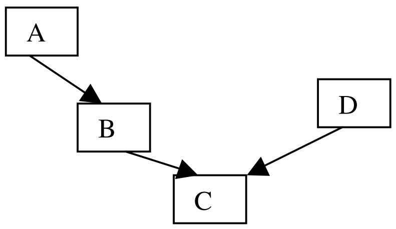

Figure 9: Partial order

Total Order

In a total order all the items are ordered. We can always take a partial order and produce a total order out of it. Figure 10 shows a total order of the partial order shown in figure 9. Here node D has been arbitrarily assigned a position, i.e. B is less than D and D is less than C. One way of avoiding deadlock given a total ordering of resources is to always traverse in one direction. For example if in figure 10, A, B, C and D are resources, we will always traverse from A to D, that is, we will grab resources in this order. If we decide to change the direction, we can get into a cycle, i.e. we may request a resource that somebody is using. In general, we try to number resources and then try to come up with a strategy that traverses resources in that order.

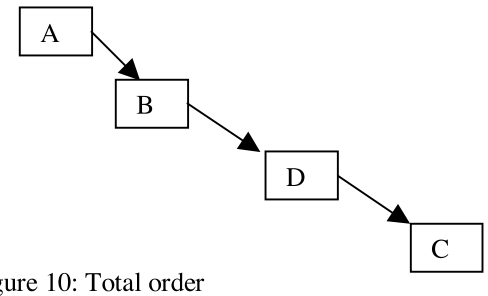

Figure 10: Total order

Deadlock on two dimensions:

Consider the case of a mesh:

If the packets route minimally, can that deadlock?

Yes, the example is shown in figure 11 below. There are four packets coming into the four nodes (one packet into each node) at the corners of the square. Each packet wants to route to the corresponding node's neighbor in the anti-clockwise direction. So each packet is holding onto the incoming channel and asking for an outgoing channel. This introduces a cyclic dependency in this particular configuration, leading to deadlock.

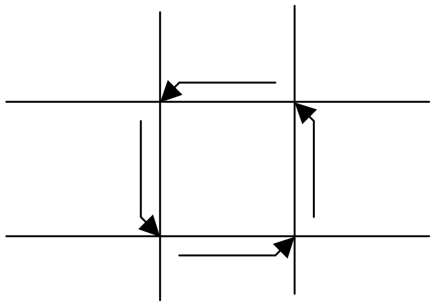

Figure 11. Gridlock

This is like a traffic condition called "gridlock" where traffic blocks around a block.

If the routing function is arbitrary (can go from any channel to any other channel) then there are arcs from an incoming channel to all the all the outgoing channels (except the reverse channel) in the dependency graph. This leads to multiple cyclic dependencies in a mesh network (like the one shown above) and unless explicit effort is made to avoid deadlock, the network will probably deadlock. Another way to state the same problem is that since the resources (channels in this case) are not ordered so there is potential for deadlock. Prof. Dally said that there are multiple examples of people who designed meshes that did not display deadlocks in simulation but when the real router was built it deadlocked.

Turn Model

One way to avoid deadlock in meshes/tori is to disallow certain turns. We will look at 2 dimensional meshes. The treatment can be generalized to multiple dimensions.

Let's say we do not allow N-W turn to break the cycle shown in Figure 11. So the allowed turns are:

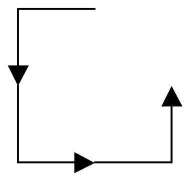
Figure 12.

Then we will not have deadlocks with just anti-clockwise cycles but we could still have deadlocks in combination with clock-wise paths. We need to disallow one turn in the clockwise path also.

Can we disallow any one turn and avoid deadlock? No, if we disallow W-N turn we can still create a cyclic dependency like this:

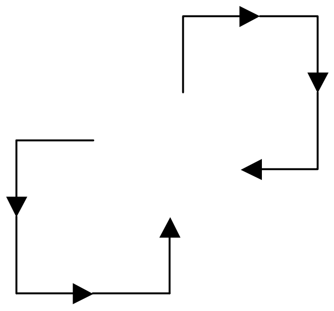
Figure 13.

This is equivalent to making a left turn by making three right turns (N-W is converted to N-E, E-S and S-W).

The other three options are all acceptable and lead to quite different routine policies. If we disallow the S-W turn then we cannot ever turn west (remember we disallowed the N-W turn already). So in this case you always have to route West first. Starting from source the packet has to go at least as far west as the destination (it can overshoot going further west) and then thread its way back to the destination through any arbitrary path (it still cannot go east of the destination).

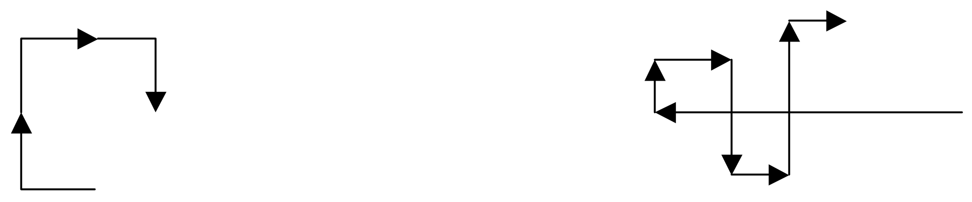
Figure 14. West first routing policy

If we disallow the E-S turn, then the policy is Negative first. That is the packet has to route in negative directions (West and South) first. Once it starts going E or N it cannot turn to W or S.

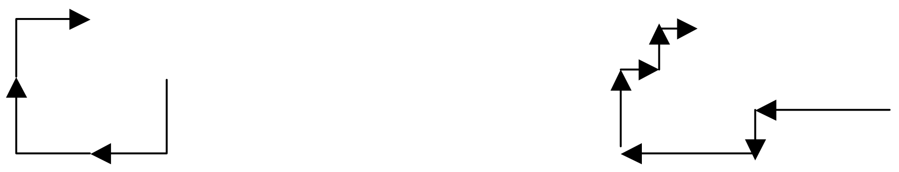
Figure 15. Negative first routing policy

If we disallow the N-E turn, then the policy is North last. Once the packet starts going north it cannot turn in any other direction (as both N-E and N-W turns are disallowed).

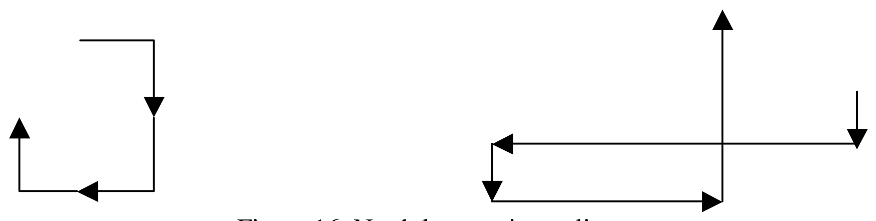
Figure 16. North last routing policy

All of these in isolation reduce the route diversity as they take out some of the available routes. To get the route diversity back we make use of virtual channels. One set of virtual channels would have one turn taken out and another set of virtual channels would have another corner taken out. So using various VCs we can route over all paths.

For two dimensions we have 4 * 2 = 8 turns and we can get deadlock free routing by removing 2 turns out of them. This can be generalized to larger dimensions. It can be proved that we need to remove one fourth of the allowed turns to make the routing deadlock free [1]. In an *n* dimension mesh there are 2*n*(2*n*-2) possible turns and we need to disallow one fourth of them.

Deadlock in Adaptive Routing

R : C X N → P(C)

Here **R** is a relation that returns a set of outgoing channels we can choose from. Note that **R** is a relation instead of a function (as in the case of oblivious routing), and it returns a set of channels instead of one channel.

Does a cycle in the relation lead to deadlock?

Not necessarily. Suppose we never wait on a channel if it is busy. If we always have an escape route then we can not have a deadlock. But if we just flip a coin to decide which channel to take then we can have deadlock. As shown in figure 17 we could flip coins to take the channels in the cycle and lead to deadlock.

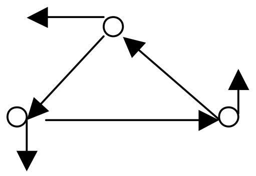
Figure 17. Cyclic dependency in adaptive routing

There will never be a deadlock if Duato's conditions hold.

Duato's Conditions

If there is a set of channels **C1** which is a subset **C** (set of all channels) such that

1. **C1** is connected (We can go from any node to any other node using channels just in **C1**)
2. **C1** is acyclic => **C1** ∩ **R** is acyclic.
3. **C1** is acyclic including the indirect dependencies.

As an example of indirect dependency consider:

$$a \rightarrow p \rightarrow q \rightarrow b$$

The channels 'a' and 'b' are in **C1** and channels 'p' and 'q' are in **C2 = C - C1**. So there is no direct dependency between 'a' and 'b' but there is an indirect dependency because a path exists from 'a' to 'b' through 'p' and 'q'. A very long packet might start at 'a' and the grab the channels 'p' and 'q' and wait at channel 'b' thus creating a dependency from 'a' to 'b'.

An example using Duato's conditions:

Consider a 2 dimensional mesh with 2 virtual channels per physical link. One set of virtual channels **C1** is to be routed dimension order and we change the routing on the other set of channels **C2** and see if we have deadlock.

If we just consider **C1** then it is acyclic and hence deadlock free. To see that **C1** is deadlock free notice that we can induce a partial order (and hence impose a full order) on the channels in **C1.**
In dimension order routing we always route **x** dimension first and then the **y** dimension. So an order would be as follows:
$\mathbf{X_e - X_w - Y_n - Y_s}$.

The order in this case is, **x** dimension channels going east (west most channel first), followed by **x** dimension channels going west (eastern channels first), followed by **y** dimension channels going north (south most channel first) followed by **y** dimension channels going south (northern channels first). Notice that there is no implicit order between $\mathbf{X_e}$ and $\mathbf{X_w}$ due to routing relation but we have artificially imposed that order to prove acyclicity. Similarly there is no order among the $\mathbf{X_e}$ channels at the same **x** coordinates, but we can arbitrarily impose a north first ordering (as they are guaranteed never to come in the same path).

Let **C2** consist of all minimal paths. Is this the network still deadlock free? To prove this we have to prove that **C1**'s dependency graph (including indirect dependencies) is still acyclic.
To prove that the dependencies are acyclic we just need to be able to order the channels such that it is not possible to break that ordering. It can be seen that minimal routing still preserves the order that we had determined earlier. To see this note the following:

- No two channels (either **x** or **y**) at the same coordinate can have a dependency through **C2** because we cannot use two channels with the same coordinate in any route. This preserves the arbitrary order we had imposed on the channels at the same coordinates.
- No two channels in opposite directions ($\mathbf{X_e}$ and $\mathbf{X_w}$, $\mathbf{Y_n}$ and $\mathbf{Y_s}$) can have a dependency as no two channels in opposite directions can occur in a minimal route. This preserves the arbitrary ordering between $\mathbf{X_e}$ and $\mathbf{X_w}$, and $\mathbf{Y_n}$ and $\mathbf{Y_s}$.
- No minimal route can take the packet to such a position that it would have to route on the **y**-axis before **x**-axis.

Hence the order is preserved and **C1** remains acyclic including the indirect dependencies contributed by **C2**. It turns out that we can even allow some misroutes in the **C2** channels as long as they do not break the order we have imposed on the routing. We show an example in the figures 18.

- Before reaching the **y** coordinate of the destination we can misroute in the **y** direction as much as we want. This introduces a new dependency in the **x** dimension channels (shown with the curved arrow and labeled **A** in figure 18) but the order is still preserved. This example is shown in figure 18.

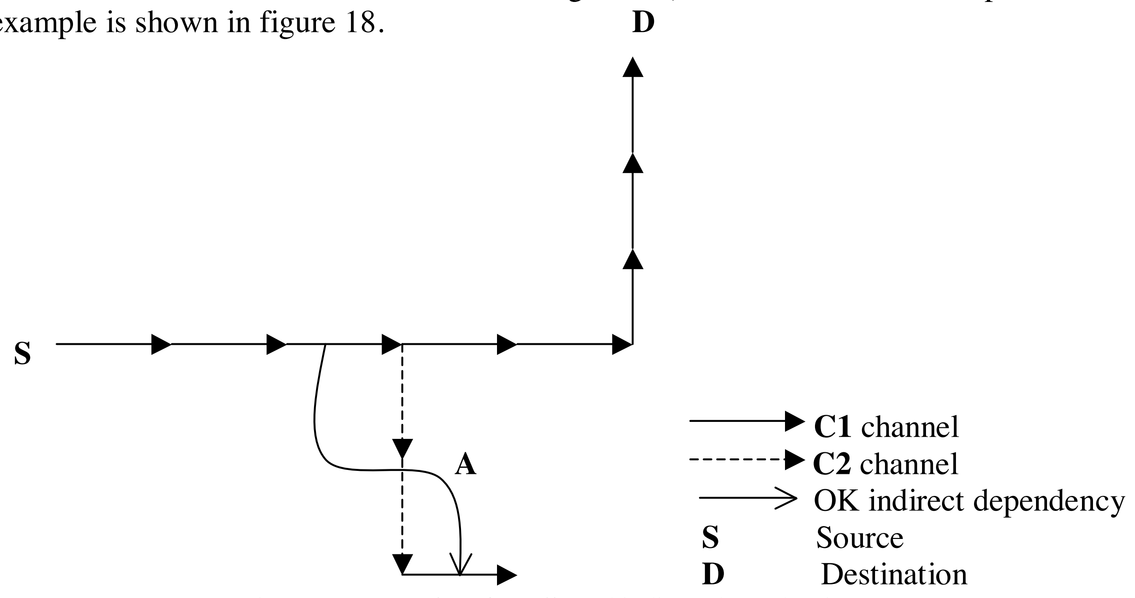

Figure 18. Examples of an allowed indirect dependencies

- Once we reach the right **y** coordinate we cannot misroute otherwise we introduce an indirect dependency from the **y** dimension **C1** channels to the **x** dimension **C1** channels (shown with dashed curved arrow labeled **B** in figure 19). This breaks the ordering between the **x** dimension **C1** channels and the **y** dimension **C1** channels.

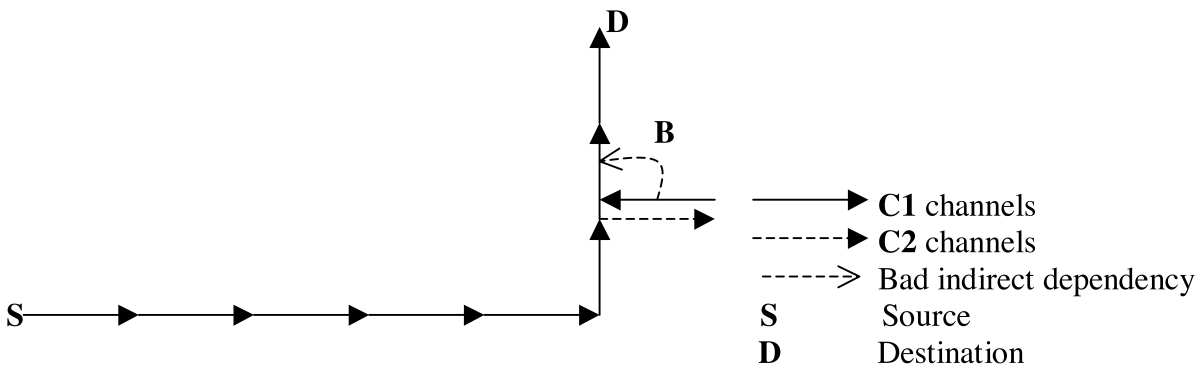

Figure 19. Example of a disallowed indirect dependency

- We also cannot misroute both in **x** and the **y** dimensions. This also introduces an indirect dependency (labeled **C** in figure 20) from the **y** dimension **C1** channels to the **x** dimension **C1** channels.

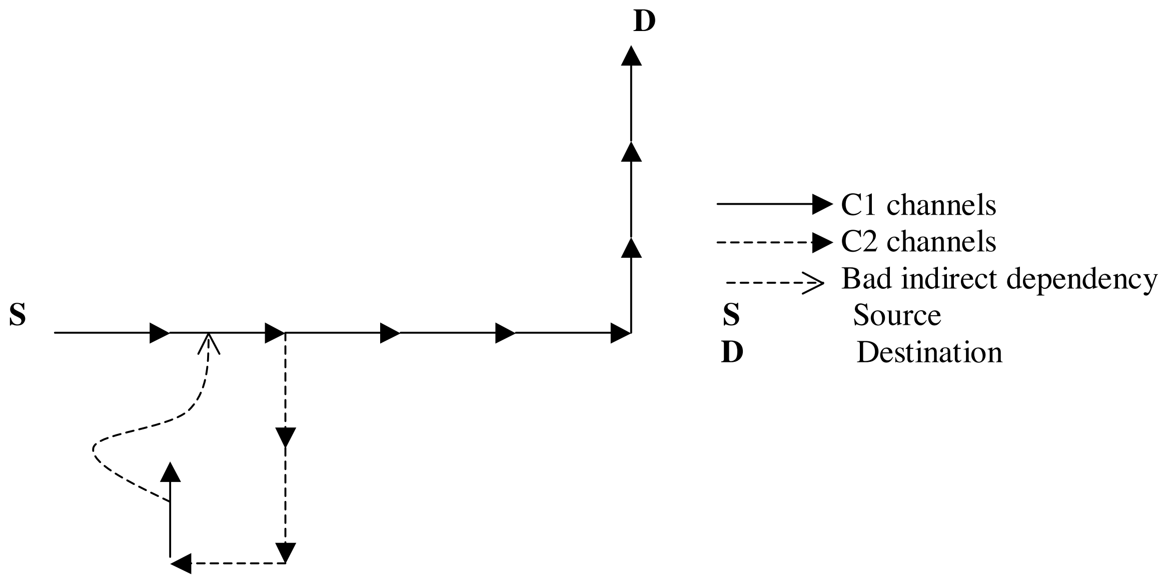

Figure 20. Example of a disallowed indirect dependency due to misrouting in two dimensions.

The above approach restricts us to near minimal routing on **C2** channels. To allow more misroutes we can do the following. Route wherever you want on the **C2** channels and once you get onto the **C1** channels then restrict yourself to the **C1** channels. That way there will not be any indirect dependencies in **C1** and we can always route without deadlock.

Another option is to add more virtual channels. We could add one set of virtual channels (say **X'**) to misroute in the **x** coordinate and another set (say **Y'**) to misroute on the **y** coordinate. Then if we need to come from the **y** coordinate back to **x** coordinate we could do that over the third set of virtual channels and avoid the dependency in **C1**. The ordering in that case would be as follows:

$\mathbf{X_e - X_w - Y_n - Y_s - X_e' - X_w' - Y_n' - Y_s'}$

Everything we have looked at so far was about deadlock avoidance but there is another way out of deadlock.

Deadlock Recovery

The philosophy is that deadlock rarely happens so there is no point in giving up the flexibility in routing to avoid this rare case. Instead we can deal with the deadlock after it happens. The idea is to not worry about the rare case and optimize the common case to get better routing (more optimal, more route diversity). Prof. Dally believes that the argument is fallacious, as there exist methods to avoid deadlocks without any appreciable decrease in the quality of routing.

To implement deadlock recovery we need to do two things:

1. **Detect Deadlock**: To detect real deadlock we need global knowledge, which is not available sitting at the router. It could possibly be done but would be very complex and expensive. An easier option is to approximate deadlock condition by using timeout. So if a particular channel has been unable to send a message for a large amount of time the router can assume that it is in a deadlock. The timeout can also be because of a bottleneck.
2. **Recover from Deadlock**:

- *Drop the packet*: Depends on the higher level protocol. In IP it may be acceptable to drop packets but it not feasible for processor-memory interconnect.
- *Have an escape path*: According to Prof. Dally, this is just like Duato's algorithm where we have a set of acyclic channels. The escape channels in this case can be looked upon as that set because they behave exactly like VCs (they have state, have buffering). Each node has *n+1* channels, *n* channels are used for normal routing and 1 channel is restricted for escape. All the escape channels are linked in a hamiltonian cycle (a cycle that goes through all the nodes once and only once). You would still need to be careful as this is nothing but 'N' (where N is the number of nodes in the network) ring which itself is cyclic. Another problem with this approach is that with heavy traffic (when deadlocks are most likely to happen) many packets will timeout and all of them will get queued up on this escape path clogging it. It will then take a long time to get out of deadlock.
- Another method from last year's scribed lecture is to buffer the packet on the node itself and wait for the channels to become free. This is not possible because generally the memory bandwidth (and capacity) on the node is far less than the network bandwidth.

Deadlock due to Higher Level Dependencies:

Even if the interconnection network is designed to be deadlock free in itself, the higher level protocol may introduce additional dependencies that could lead to deadlock. An example of this a cache coherence protocol working over an interconnection network. The processor sends a request over a certain set of channels **A** and the memory sends a reply over a certain other set of channels **B**, thus introducing a dependency **d** from **A** to **B** (shown in figure 21, the channels in **A** will not be released till **B** is free). So now although the interconnection network might itself be deadlock free the higher level protocol may deadlock. The way out of this is to use one set of virtual channels (panacea for all interconnection network problems!!) for requests from the processor and another set of virtual channels for reply from the memory. This way the additional dependencies will not be within the same set of VCs and hence avoid a cycle.

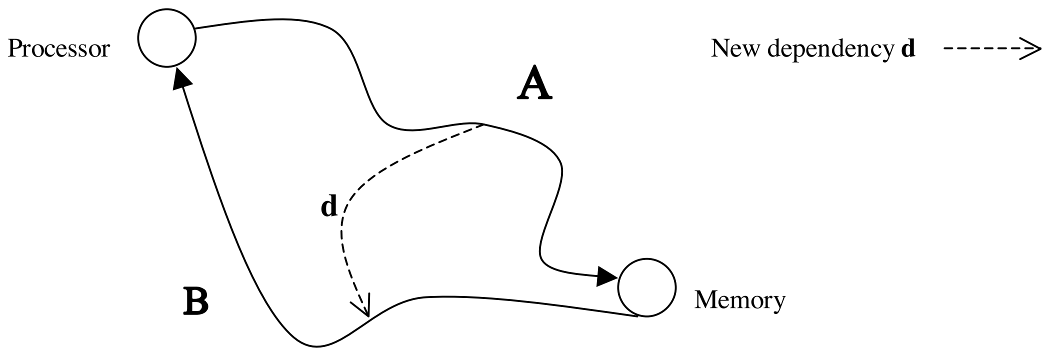

Figure 21. Higher level dependencies

Reference:
Glass, C.J.; Ni, L.M., "The Turn Model for Adaptive Routing", *Proceedings of The 19th Annual International Symposium on Computer Architecture*, pp. 278-287, 1992