ECE6115 Lab 3: Deadlock Avoidance Report

Alvin Li

**A.1.**  //UR deadlocks @ 0.22

//BC deadlocks @ 0.07

//Tornado does not deadlock

|  |  |  |
| --- | --- | --- |
| **Traffic Pattern** | **Peak Throughput with XY (packets/node/cycle)** | **Peak Throughput with Random (packets/node/cycle)** |
| **Uniform Random** | 0.34578125 | 0.20753671875 |
| **Bit Complement** | 0.15278125 | 0.0598640625 |
| **Tornado** | 0.24984921875 | 0.24984921875 |

**A.2.** The middle 4 nodes in any mesh could cause a deadlock in bit-complement traffic. This is because of the potential cycle that can be formed by going corner-to-corner in a 2x2 array. Given the 8x8 mesh: (where all nodes are connected horizontally and vertically to adjacent nodes)

We can analyze the center 4 nodes highlighted in orange:

The bottom left node is 27, bottom right is 28, top left 35, top right 36. In bit-complement traffic:

Destination Router ID = (# nodes in mesh) – Source Router ID – 1

Thus, 27-36, 28-35, 36-27, 35-28 are the four flows. In the case of only clockwise/only counterclockwise routing, there is a clear cycle present, potentially manifesting into a deadlock.

**B.**

turn\_model\_oblivious was implemented basically the same way as random\_oblivious. The only differences are that when the direction is in the northwest (Quadrant II), the packet is routed west only, and when the direction is in the southeast (Quadrant IV), the packet is routed south only. The turn\_model\_oblivious function code is:

int

RoutingUnit::outportComputeTurnModelOblivious(RouteInfo route,

int inport,

PortDirection inport\_dirn)

{

PortDirection outport\_dirn = "Unknown";

int M5\_VAR\_USED num\_rows = m\_router->get\_net\_ptr()->getNumRows();

int num\_cols = m\_router->get\_net\_ptr()->getNumCols();

assert(num\_rows > 0 && num\_cols > 0);

int my\_id = m\_router->get\_id();

int my\_x = my\_id % num\_cols;

int my\_y = my\_id / num\_cols;

int dest\_id = route.dest\_router;

int dest\_x = dest\_id % num\_cols;

int dest\_y = dest\_id / num\_cols;

int x\_hops = abs(dest\_x - my\_x);

int y\_hops = abs(dest\_y - my\_y);

bool x\_dirn = (dest\_x >= my\_x);

bool y\_dirn = (dest\_y >= my\_y);

// already checked that in outportCompute() function

assert(!(x\_hops == 0 && y\_hops == 0));

/////////////////////////////////////////

// ICN Lab 3: Insert code here

if (x\_hops == 0) {

if (y\_dirn > 0)

outport\_dirn = "North";

else

outport\_dirn = "South";

} else if (y\_hops == 0) {

if (x\_dirn > 0)

outport\_dirn = "East";

else

outport\_dirn = "West";

} else {

int rand = random() % 2;

if (x\_dirn && y\_dirn) // Quadrant I

outport\_dirn = rand ? "East" : "North";

else if (!x\_dirn && y\_dirn) // Quadrant II

outport\_dirn = "West";

else if (!x\_dirn && !y\_dirn) // Quadrant III

outport\_dirn = rand ? "West" : "South";

else // Quadrant IV

outport\_dirn = "South";

}

return m\_outports\_dirn2idx[outport\_dirn];

}

**C.**

My code deadlocks or throws the assert state IDLE error. My thinking was to detect illegal turns by seeing if any packet took a non-west-first turn. If this were the case, then the virtual channel the packet was on is incremented to the next (higher #) available virtual channel. Virtual channel 3, the 4th virtual channel, is defined in outportCompute as the escape VC with the same routing scheme as turn\_model\_oblivious. If any packet that takes an illegal turn potentially going toward a deadlock, it will end up in the escape VC before said deadlock can occur. I’m not sure why my code is not working, but it is pasted below:

int

RoutingUnit::outportCompute(RouteInfo route, int inport,

PortDirection inport\_dirn,

int vc)

{

int outport = -1;

if (route.dest\_router == m\_router->get\_id()) {

// Multiple NIs may be connected to this router,

// all with output port direction = "Local"

// Get exact outport id from table

outport = lookupRoutingTable(route.vnet, route.net\_dest);

return outport;

}

//code start

if (vc == 3) {

outport = outportComputeTurnModelOblivious(route, inport, inport\_dirn);

assert(outport != -1);

return outport;

}

//code end

// Routing Algorithm set in GarnetNetwork.py

// Can be over-ridden from command line using --routing-algorithm = 1

RoutingAlgorithm routing\_algorithm =

(RoutingAlgorithm) m\_router->get\_net\_ptr()->getRoutingAlgorithm();

switch (routing\_algorithm) {

case TABLE\_: outport =

lookupRoutingTable(route.vnet, route.net\_dest); break;

case XY\_: outport =

outportComputeXY(route, inport, inport\_dirn); break;

case TURN\_MODEL\_OBLIVIOUS\_: outport =

outportComputeTurnModelOblivious(route, inport, inport\_dirn); break;

case TURN\_MODEL\_ADAPTIVE\_: outport =

outportComputeTurnModelAdaptive(route, inport, inport\_dirn); break;

case RANDOM\_OBLIVIOUS\_: outport =

outportComputeRandomOblivious(route, inport, inport\_dirn); break;

case RANDOM\_ADAPTIVE\_: outport =

outportComputeRandomAdaptive(route, inport, inport\_dirn); break;

// any custom algorithm

case CUSTOM\_: outport =

outportComputeCustom(route, inport, inport\_dirn); break;

default: outport =

lookupRoutingTable(route.vnet, route.net\_dest); break;

}

assert(outport != -1);

return outport;

}

// Check if the output port (i.e., input port at next router) has free VCs.

bool

OutputUnit::has\_free\_vc(int vnet, int invc,

PortDirection inport\_dirn, PortDirection outport\_dirn, RouteInfo route)

{

// ICN Lab 3:

// Hint: invc, route, inport\_dirn, outport\_dirn are provided

// to implement escape VC

bool illegal = false;

if (inport\_dirn != "East" && outport\_dirn == "West") {

illegal = true;

}

int vc\_base = vnet\*m\_vc\_per\_vnet;

for (int vc = vc\_base; vc < vc\_base + m\_vc\_per\_vnet; vc++) {

if ((is\_vc\_idle(vc, m\_router->curCycle()) && vc > invc && illegal) || (is\_vc\_idle(vc, m\_router->curCycle()) && invc == 3 && vc >= invc && illegal))

return true;

}

return false;

}

// Assign a free output VC to the winner of Switch Allocation

int

OutputUnit::select\_free\_vc(int vnet, int invc,

PortDirection inport\_dirn, PortDirection outport\_dirn, RouteInfo route)

{

// ICN Lab 3:

// Hint: invc, route, inport\_dirn, outport\_dirn are provided

// to implement escape VC

bool illegal = false;

if (inport\_dirn != "East" && outport\_dirn == "West") {

illegal = true;

}

int vc\_base = vnet\*m\_vc\_per\_vnet;

for (int vc = vc\_base; vc < vc\_base + m\_vc\_per\_vnet; vc++) {

if ((is\_vc\_idle(vc, m\_router->curCycle()) && vc > invc && illegal) || (is\_vc\_idle(vc, m\_router->curCycle()) && invc == 3 && vc >= invc && illegal)) {

m\_outvc\_state[vc]->setState(ACTIVE\_, m\_router->curCycle());

return vc;

}

}

return -1;

}