ECE 6115: Interconnection Networks

Lab 3 Report: Deadlock Avoidance

Contents

Part A: Deadlock detection

A.1

Command used:

./build/Garnet_standalone/gem5.opt configs/example/garnet_synth_traffic.py --network=garnet2.0 --num-cpus=64 --num-dirs=64 --topology=Mesh --mesh-rows=8 --sim-cycles=20000 --

synthetic=uniform_random --vcs-per-vnet=4 --inj-vnet=0 --injectionrate=0.02 --routing-algorithm=<xy, random_oblivious>

Traffic Pattern	**Peak Throughput with XY (packets/node/cycle)**	**Peak Throughput with Random (packets/node/cycle)**
Uniform Random	0.3458 (442600/64/20000)	0.2075 (265647/64/20000)
Bit Complement	0.1528 (195560/64/20000)	0.0599 (76626/64/20000)
Tornado	0.2498 (319807/64/20000)	0.2498 (319807/64/20000)

NOTE: For unform random traffic pattern and random oblivious routing algorithm, the injection rate was 0.21 prior to deadlocking at 0.22. This is not in the 0.02 increment, but at 0.01 increments in the injection rates. If the injection rates increment of 0.02 are required for this, then:

Injection rate = 0.2 for Traffic = Uniform Random for Routing = Random Oblivious

Peak Throughput = 0.1997 (255594/64/20000)

A.2

For bit complement traffic pattern, the pattern for 4x4 mesh is:

NOTE: Assumes 4x4 mesh for analysis (as per piazza), scalable to 8x8

Source Node	**Destination Node**
0	15
1	14
2	13
3	12
4	11
5	10
6	9
7	8
8	7
9	6
10	5
11	4
12	3
13	2
14	1
15	0

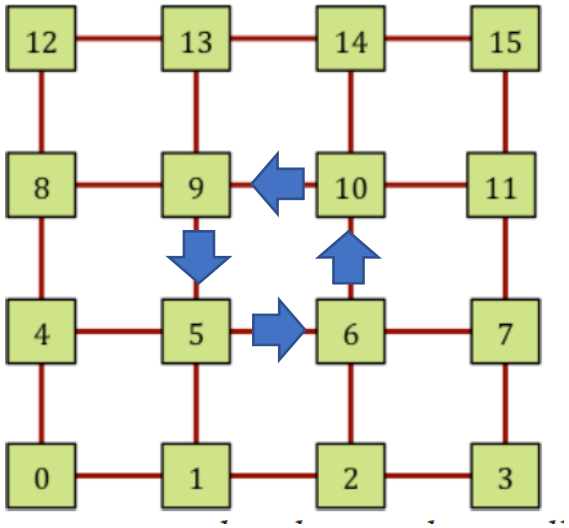


Figure 1: Deadlock cycle in 4x4 mesh

A deadlock can occur in the following src-dest pairs:

1. 5 -> 10
2. 6 -> 9
3. 10 -> 5
4. 9 -> 6

This will create a cycle in the center square in the 4x4 mesh formed by 5, 6, 10, 9 which has resource dependency between each of them, but the flit cannot move forward as the next buffer cannot be emptied, so on.

Part B: Deadlock Avoidance using Turn Model

Turn Model Oblivious (North-last turn model)

Command used: (as per Piazza)

./build/Garnet_standalone/gem5.opt configs/example/garnet_synth_traffic.py --network=garnet2.0 --num-cpus=64 --num-dirs=64 --topology=Mesh --mesh-rows=8 --sim-cycles=50000 --num-packets-max=60 --synthetic=uniform_random --vcs-per-vnet=4 --inj-vnet=0 --injectionrate=0.8 --routing-algorithm=turn_model_oblivious

North-last turn model description: For any routing, the northward turn from any direction, should be the last turn the flit takes. So, there should be no turns possible from north direction to any other direction. This has been implemented in code by:

1. Check x_hops and y_hops (along with direction):
    a. If x_hops == 0, then the flit can only go either north or south, and would not need any turns
    b. If y_hops == 0, then the flit can only go either west or east, and would not need any turns
    c. If the flit needs to go towards north-west, then it must first take west direction, then turn north when x_hops become 0.
    d. If the flit needs to go towards north-east, then it must first take the east direction, then turn north when x_hops become 0.
    e. If the flit needs to go towards south-east/west, then it can take either direction south or east/west based on random selection (as oblivious)
2. Based on the above turns, it will decide the next node to go to.

Code:

```
// File: RoutingUnit.cc
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));

  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 {
    // Random number for oblivious selection
    int rand = random() % 2;
    if (x_dirn && y_dirn) // Quadrant I
      outport_dirn = "East";
    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 = rand ? "East" : "South";
  }
  return m_outports_dirn2idx[outport_dirn];
}
```

Extra Credit: Turn Model Adaptive (North-last turn model)

Command used: (as per Piazza)

./build/Garnet_standalone/gem5.opt configs/example/garnet_synth_traffic.py --network=garnet2.0 --num-cpus=64 --num-dirs=64 --topology=Mesh --mesh-rows=8 --sim-cycles=50000 --num-packets-max=60 --synthetic=uniform_random --vcs-per-vnet=4 --inj-vnet=0 --injectionrate=0.8 --routing-algorithm=turn_model_adaptive

Adaptive North-last turn model description: For any routing, the northward turn from any direction, should be the last turn the flit takes. Additionally, the choice of next turn from the available turns is done based on available free VCs in the next router:

1. Check x_hops and y_hops (along with direction):
    a. If x_hops == 0, then the flit can only go either north or south, and would not need any turns
    b. If y_hops == 0, then the flit can only go either west or east, and would not need any turns
    c. If the flit needs to go towards north-west, then it must first take west direction, then turn north when x_hops become 0.
    d. If the flit needs to go towards north-east, then it must first take the east direction, then turn north when x_hops become 0.
    e. If the flit needs to go towards south-east/west, then it can take either direction south or east/west based on:
        i. Check number of free VCs in the router in the south direction and east/west direction.
        ii. Select the router which has larger number of available free VCs.
2. Based on the above turns, it will decide the next node to go to.

Code:

Function to get the free VCs in the output unit:

```
// File: OutputUnit.cc
// Get the number of free vcs in the output direction
int OutputUnit::get_free_vc_count(int vnet) {
   int vc_free_count = 0;
   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_free_count;
      }
   }
   return vc_free_count;
}
```

Function to get the free VCs for the output unit in a given direction:

```
// File: Router.cc
// Expose the free vc counter function from the Output unit to the router
int Router::get_free_vc_count(int outport, int vnet) {
   return m_output_unit[outport]->get_free_vc_count(vnet);
}
```

Updated routing algorithm to use the above 2 functions for the adaptive nature, in which it selects the direction based on number of free VCs. If the number of free VCs is the same, then it randomly chooses one. The North-last model is maintained similar to the last section.

```
// File: RoutingUnit.cc
int RoutingUnit::outportComputeTurnModelAdaptive(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));

  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 {
     // Get the number of free ports in each directions (apart from North)
     int EastPort = m_outports_dirn2idx["East"];
     int WestPort = m_outports_dirn2idx["West"];
     int SouthPort = m_outports_dirn2idx["South"];

     int EastVcCount = this->m_router->get_free_vc_count(EastPort, route.vnet);
     int WestVcCount = this->m_router->get_free_vc_count(WestPort, route.vnet);
     int SouthVcCount = this->m_router->get_free_vc_count(SouthPort, route.vnet);

     int rand = random() % 2;

     if (x_dirn && y_dirn) // Quadrant I
        outport_dirn = "East";
     else if (!x_dirn && y_dirn) // Quadrant II
        outport_dirn = "West";
     else if (!x_dirn && !y_dirn) // Quadrant III
        if (WestVcCount > SouthVcCount) {
           outport_dirn = "West";
        } else if (WestVcCount < SouthVcCount) {
           outport_dirn = "South";
        } else {
           outport_dirn = rand ? "West" : "South";
        }
     else // Quadrant IV
        if (EastVcCount > SouthVcCount) {
           outport_dirn = "East";
        } else if (EastVcCount < SouthVcCount) {
           outport_dirn = "South";
        } else {
           outport_dirn = rand ? "East" : "South";
        }
  }
  return m_outports_dirn2idx[outport_dirn];
}
```

Part C: Deadlock Avoidance using Escape VC

Command used: (as per Piazza)

./build/Garnet_standalone/gem5.opt configs/example/garnet_synth_traffic.py --network=garnet2.0 --num-cpus=64 --num-dirs=64 --topology=Mesh --mesh-rows=8 --sim-cycles=50000 --num-packets-max=60 --synthetic=uniform_random --vcs-per-vnet=4 --inj-vnet=0 --injectionrate=0.8 --routing-algorithm=random_oblivious

Escape VC Description: Implemented an escape Virtual Channel with a deadlock free routing algorithm (adaptive west-first turn).

1. Implemented and tested the adaptive west-first turn model.
    a. Check x_hops and y_hops (along with direction):
        i. If x_hops == 0, then the flit can only go either north or south, and would not need any turns
        ii. If y_hops == 0, then the flit can only go either west or east, and would not need any turns
        iii. If the flit needs to go towards north-west, then it must first take west direction, then turn north.
        iv. If the flit needs to go towards south-west, then it must first take the west direction, then turn south.
        v. If the flit needs to go north/south-east, then it can take either direction north/south or east based on:
            1. Check number of free VCs in the router in the south direction and east/west direction.
            2. Select the router which has larger number of available free VCs.
    b. Based on the above turns, it will decide the next node to go to.
    c. Tested the turn model for all configuration to verify its deadlock free behavior.
2. Updated the VC allocation logic to:
    a. Select any free VC as the output VC.
    b. Reserve the last VC (as per index) as the escape VC.
    c. If escape VC is selected => no other VC free as it is the last in the index during idle VC checks.
        i. Check if the flit is entering the next VC (escape VC) in the north/south direction.
        ii. If so, then check whether it needs to go towards the west direction in the future using the x_hops and x_dirn logic. If such a turn exists, then it is an illegal turn and it will violate the west-first turn model, which states that the flit must not turn towards west direction from north/south.
        iii. If it is making an illegal turn, then return that no VC is free as escape VC cannot be allocated even if free, due to illegal turn.
3. In the routing algorithm computation:
    a. If the VC is not escape VC, then it will use random oblivious routing algorithm.
    b. If the VC is escape VC, then it will use the adaptive west-first turn model.

Code:

NOTE: Used adaptive algorithm for escape VC as it provided better deadlock avoidance across multiple test-runs.

Function updated for incorporating the escape VC routing algorithm for output computation:

```
// File: RoutingUnit.cc
int RoutingUnit::outportCompute(RouteInfo route, int inport, PortDirection inport_dirn, int vc) {
```

```
    ... // Other parts same as exiting code, truncated for readability.
    // Get the escape VC number
    int vnet = route.vnet;
    int vc_base = vnet * m_router->get_vc_per_vnet();
    int vc_escape1 = vc_base + m_router->get_vc_per_vnet() - 1;

    switch (routing_algorithm) {
        ... // Other parts same as exiting code, truncated for readability.
        case RANDOM_OBLIVIOUS_:
            if (vc != vc_escape1) {
                outport =
                outportComputeRandomOblivious(route, inport, inport_dirn);
            } else {
                outport =
                outportComputeTurnModelWestFirst(route, inport, inport_dirn);
            }
            break;
        ... // Other parts same as exiting code, truncated for readability.
}
```

Implemented the west-first turn model to be used for escape VCs:

```
// File: RoutingUnit.cc
// Adaptive west-first turn model (adative nature -> select direction based on free VCs)
int RoutingUnit::outportComputeTurnModelWestFirst(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));
```

```
  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 {
    // Get the number of free ports in each directions (apart from North)
    int EastPort = m_outports_dirn2idx["East"];
    int NorthPort = m_outports_dirn2idx["North"];
    int SouthPort = m_outports_dirn2idx["South"];

    int EastVcCount = this->m_router->get_free_vc_count(EastPort, route.vnet);
    int NorthVcCount = this->m_router->get_free_vc_count(NorthPort, route.vnet);
    int SouthVcCount = this->m_router->get_free_vc_count(SouthPort, route.vnet);
    int rand = random() % 2;

    if (x_dirn && y_dirn) { // Quadrant I
      if (EastVcCount < NorthVcCount) {
        outport_dirn = "North";
      } else if (EastVcCount > NorthVcCount) {
        outport_dirn = "East";
      } else {
        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 = "West";
    else { // Quadrant IV
      if (EastVcCount < SouthVcCount) {
        outport_dirn = "South";
      } else if (EastVcCount > SouthVcCount) {
        outport_dirn = "East";
      } else {
        outport_dirn = rand ? "East" : "South";
      }
    }
  }
  return m_outports_dirn2idx[outport_dirn];
}
```

Updated the output unit to support:

1. Function to find free VCs using the escape VC turn restriction logic so that it does not allow an illegal turn by turning from North or South into the west direction.
2. Used the same function into the has_free_vc and select_free_vc to maintain same functionality.

```
// File: OutputUnit.cc
// Inner function used to find the vc index of the free vc based on the algorithm
// Used by has_free_vc and select_free_vc which now act as wrapper functions for this
int OutputUnit::find_free_vc(int vnet, int invc, PortDirection inport_dirn, PortDirection outport_dirn,
RouteInfo route) {
    int vc_base = vnet * m_vc_per_vnet;

    int outvc = -1;
    int vc_escape = vc_base + m_vc_per_vnet - 1;

    // Look for next vc with equal fairness
    for (int vc = vc_base; vc < vc_base + m_vc_per_vnet; ++vc) {
        if (is_vc_idle(vc, m_router->curCycle())) {
            outvc = vc;
            // Found a free VC => use it, instead of checking further
            break;
        }
    }

    RoutingAlgorithm routing_algorithm =
        (RoutingAlgorithm) m_router->get_net_ptr()->getRoutingAlgorithm();
    if (routing_algorithm == RANDOM_OBLIVIOUS_) {
        // Check for future turn direction
        int num_cols = m_router->get_net_ptr()->getNumCols();

        int my_id = m_router->get_id();
        int my_x = my_id % num_cols;

        int dest_id = route.dest_router;
        int dest_x = dest_id % num_cols;

        int x_hops = abs(dest_x - my_x);

        bool x_dirn = (dest_x > my_x);

        // Check if outvc is escape vc
        if (outvc == vc_escape) {
            if (x_hops > 0 && x_dirn == false) {
                // It will take a westward turn in the future
                if (outport_dirn == "North" || outport_dirn == "South") {
                    // It will take an illegal turn in the future
                    return -1;
                }
            }
        }
    }
    return outvc;
}
```

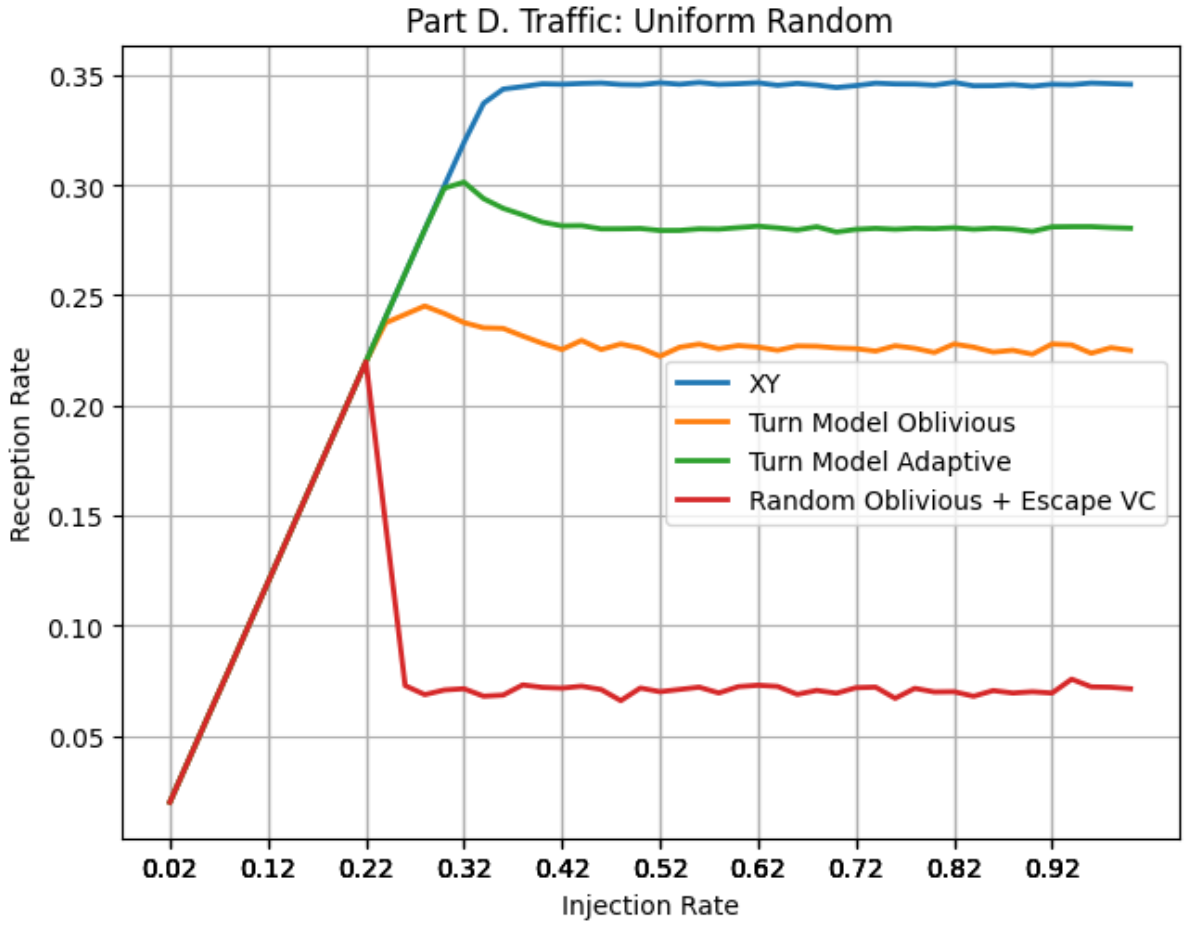
```
// Check if the output port (i.e., input port at next router) has free VCs.
bool OutputUnit::has_free_vc(int vnet, int invc, PortDirection inport_dirn, PortDirection outport_dirn,
RouteInfo route) {
    int outvc = this->find_free_vc(vnet, invc, inport_dirn, outport_dirn, route);
    return outvc != -1;
}

// 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) {
    int outvc = this->find_free_vc(vnet, invc, inport_dirn, outport_dirn, route);
    if (outvc != -1) {
        m_outvc_state[outvc]->setState(ACTIVE_, m_router->curCycle());
    }
    return outvc;
}
```

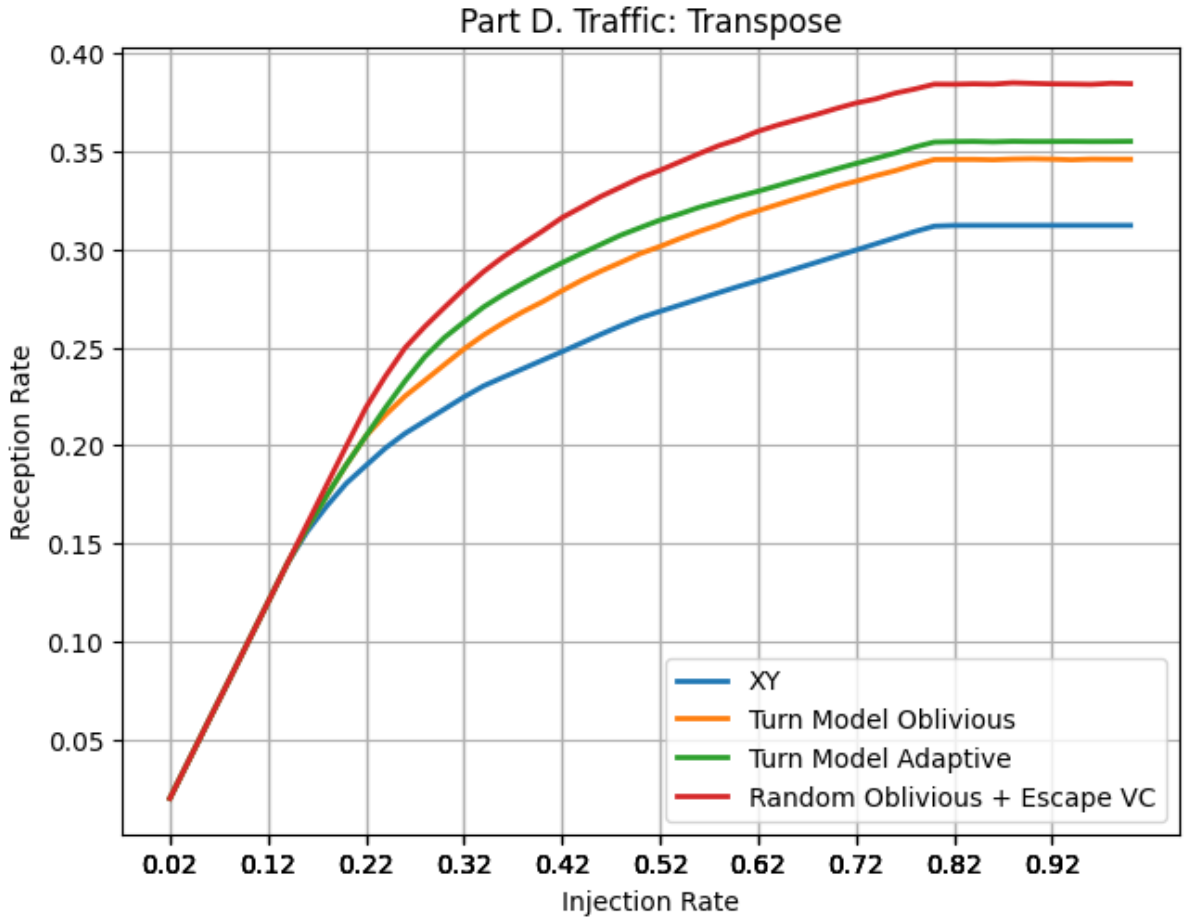
Part D: Analysis

D.1: Graphs (reception rate vs injection rate)

Graph for Uniform Random traffic pattern



Graph for Transpose traffic pattern



Throughput analysis for the above graphs for both traffic patterns:

Traffic Pattern	**Routing Algorithm with highest throughput**
Uniform Random	XY
Transpose	Random Oblivious + Escape VC

D.2: Path diversity vs throughput analysis

As per the results obtained and graphs plotted:

Throughput Ranking (Highest first)	**For Uniform Random Traffic**	**For Transpose Traffic**
1	XY	Random Oblivious
2	Turn Model Adaptive	Turn Model Adaptive
3	Turn Model Oblivious	Turn Model Oblivious
4	Random Oblivious + Escape VC	XY

Assumption: Mesh topology. Used 4x4 to explain the idea, which is scalable to 8x8 (as per piazza)

The table clearly shows that XY has the highest throughput in uniform random while lowest for transpose. Similarly, ranking flip is seen for random oblivious + escape VC (lowest in one, highest in the other). In the current case, we see that the turn model adaptive is highly ranked for both traffic pattern. But adaptive algorithms take the decision based on local information and not global congestion

information, so they could also lead to sub-optimal choices which would also depend on the traffic pattern and injections rates.

Path diversity analysis, XY offers the least path diversity where as north-last turn models provide more diversity with a restriction that the north turn should be the last and the random oblivious model gives the highest path diversity (with escape VC using west-first which limits path diversity for escape VC).

This shows that for the current cases studied, higher path diversity does not always lead to better throughput. It highly depends on the traffic pattern being considered.

The communication pattern in transpose traffic leads to communication between nodes on the opposite sides of the black lines drawn below.

Source Node	Destination Node
0	0
1	4
2	8
3	12
4	1
5	5
6	9
7	13
8	2
9	6
10	10
11	14
12	3
13	7
14	11
15	15

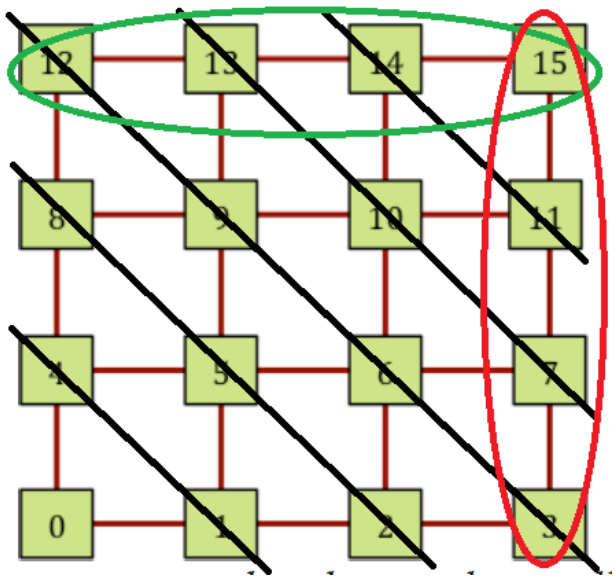


Figure 2: Transpose traffic pattern analysis

Example of congestion in XY:

Taking the **transpose traffic pattern**, all nodes in the top row send packets to their paired nodes, which lie in the right-most column. So, the packets will follow the X direction path marked in green and Y direction marked in red. So, there is only 1 path for 4 nodes communication to 4 other nodes in the topology. This greatly limits the throughput.

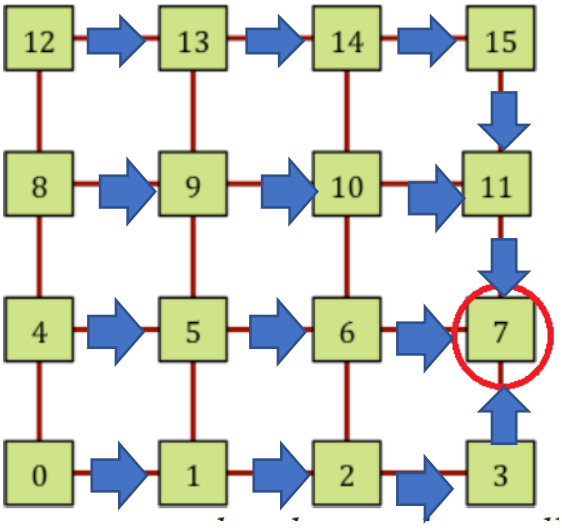


Figure 3: Uniform Random Traffic pattern with XY routing

Whereas, for the similar routing algorithm in **uniform random traffic**, any of the other nodes in the mesh could be sending their messages to say node 7, marked in red. Similar behavior can be seen for other nodes in the right most column, currently being considered. So, in this case even though the path diversity is the same, the origin node is randomized which in turn takes benefit from the XY model while in the transpose model leads to queuing of messages as it uses. The pattern shown below, also uses all inport directions while for transpose all the messages only come from the north port.

Thus, for the examples in consideration XY could be better for a random traffic pattern whereas be worse for a transpose traffic pattern.

Example of congestion in adaptive model:

Similarly, we can show that adaptive model could lead to sub-optimal results. Taking the same example of **transpose traffic pattern**, say Node 12 is sending packets to node 3. It could see that the nodes in the box marked in the red box communicate with a node in the same red box. So, the path outside the should be less congested (for the direction for current packet). It would end up taking this path. Thus, in turn avoiding congestion while some flits would still be sent through the path in the red box and use the path diversity along with using the lower congestion path to its advantage.

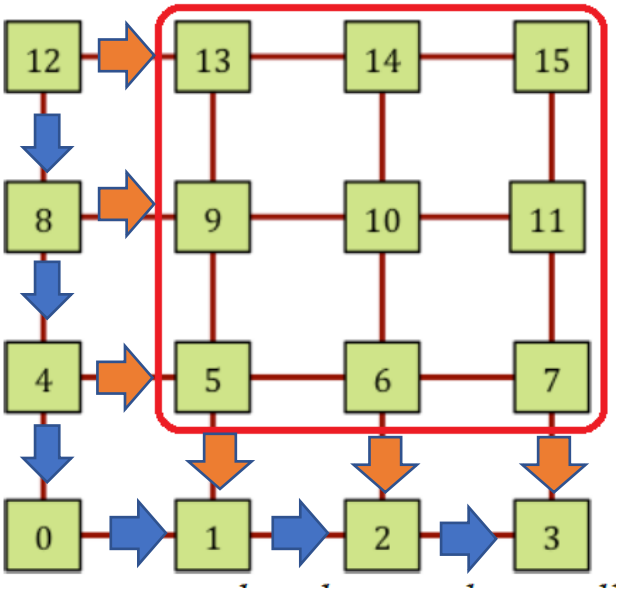


Figure 4: Transpose traffic pattern with adaptive model routing

But a similar scenario cannot be created for **uniform random traffic** as the pattern is non-deterministic. So, in this pattern, the decision based on local information might not be as good a reflection of further traffic as in the case of transpose. It could end up taking a turn decision which would be detrimental to congestion a few hops down the line. Say, in uniform random traffic pattern below, 12 sends packets to 3. (red => highly congested, orange => light congested, green => idle). So, based on local information, 12 sends its packet to 8 => 4 => 0. Now, it will have to enter a highly congested red rectangle to maintain minimal routing and could get stuck there for a long time. Whereas the path through the orange rectangle through the green one, could have been more beneficial.

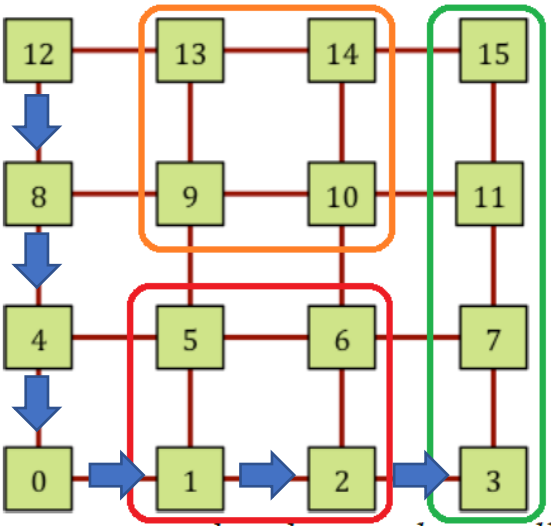


Figure 5: Uniform random traffic pattern with adaptive routing model

Thus, for the adaptive model discussed above (in the mentioned situations), it could be better for a transpose traffic pattern which is more deterministic but worse for the uniform random traffic pattern.

Similarly, for oblivious patterns, they could be a superset of the others in specific conditions. Additionally, completely non-deterministic approaches like random oblivious could led to deadlock scenarios which require additional support of escape VCs. So, a non-deterministic algorithm may or may not be a good option, as it randomly chooses the turns:

1. It could end up sending some packets towards the congested part of the network even when the other path could be less congested
2. If it has a uniform random distribution when choosing turns, then it could end up sending at least some packets towards less congested areas, which say, adaptive might skip based on the example above due to its nature of using local information.

This shows, it is probabilistic whether one algorithm performs better in a given scenario based on the traffic pattern and injection flows.

On the other hand, a deterministic algorithm (XY) which would have a streamlined flow of packets but might have queuing at routers based on traffic pattern. So, the lack of path diversity could be a bottleneck in some traffic patterns (like uniform-random) while it might not be an issue as packets end up entering better regions in the path.

This shows that the throughput differs on a case-by-case basis on multiple other factors like traffic patterns (which leads to different congestion patterns), information correlation between neighboring routers and global network (for adaptive), etc. A blanket statement that higher path diversity always leads to higher throughput cannot be made, as utilizing the higher path diversity also depends on the traffic pattern.

Thus, [NO], path diversity from our routing algorithm does not always lead to better throughput.