Interconnection Networks for High-Performance Systems
ECE 6115 / CS 8803 – ICN
Spring 2022

Lab 3: Deadlock Avoidance [55 pts]

George P. Burdell attended the Routing lecture of Interconnection Networks and felt that a minimal deterministic XY routing is inefficient as the Mesh topology loses its path diversity. He implemented a new minimal *oblivious* routing algorithm that randomly forwards a flit out of one of the ports along its minimal path at each router. Recall that an oblivious routing algorithm randomly chooses one out of available minimal output links (without taking any congestion into account).

But once he started running the network, George noticed that he is unable to inject any new messages into the network after a while and some of the old messages have not been received despite waiting for thousands of cycles. George missed the next couple of lectures of Interconnection Networks where techniques to solve this problem were discussed, because he was cross-registered for multiple classes at the same time[1]!. Your goal is to fix George's network.

Step 0:

Update your copy of gem5 which contains George's network.

```
hg pull -u
```

[You can also clone a fresh copy if you wish]

```
hg clone /nethome/tkrishna3/teaching/simulators/gem5/repo/gem5
```

Now build the simulator.

THIS NEEDS TO BE DONE EVERYTIME YOU CHANGE THE C++ CODE.

```
./my_scripts/build_Garnet_standalone.sh
```

Step 1:

Run Command:

```
./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=random_oblivious
```

The parameters in blue are what you will be varying at various points in this lab.
Note: You will be running experiments on a 64-core system in this lab, with 4 VCs (per VNet) and using gem5.opt (not gem5.debug).

Add the following to a Report.doc/pdf:

[1] https://en.wikipedia.org/wiki/George_P._Burdell

Part A: Deadlock Detection (8 points)

Determine the peak throughput with `XY` and `random_oblivious` routing for the following three traffic patterns: *uniform_random*, *bit_complement* and *tornado.*
(`--routing-algorithm=xy` / `--routing-algorithm=random_oblivious`)

Definition:

- **Reception Rate** (packets/node/cycle) is **total_packets_received/num-cpus/sim-cycles**
- **Peak Throughput.** Maximum reception rate provided by the network.

For XY routing, estimate peak throughput by looking at the *total_packets_received* at the highest possible injection rate (`--injectionrate=1.0`)

For random routing, once the network deadlocks, you will see a deadlock assertion failure.
For random routing, estimate peak throughput using the *total_packets_received* at the injection rate *right before the network first deadlocks.*

A.1. Fill in the following table and add it to your report *(6 points):*

Traffic Pattern	**Peak Throughput with XY (packets/node/cycle)**	**Peak Throughput with Random (packets/node/cycle)**
Uniform Random		
Bit Complement		
Tornado		

A.2. For Bit-Complement traffic, can you come up with an example of four flows (*i.e., src – dest pairs)* that can manifest into a deadlock? Draw this out as an example in your report *(2 points)*

Part B. Deadlock Avoidance using Turn Model (15 points)

Avoid the deadlock by **implementing *the North-last turn model*** in Garnet. Assume the routing algorithm is **oblivious** – i.e., it randomly selects among the ***viable*** output ports for minimal routing. The routing code is here: `src/mem/ruby/network/garnet2.0/RoutingUnit.cc`

Implement the function `outportComputeTurnModelOblivious()`
See how `outportComputeXY()` and `outportComputeRandomOblivious()` are implemented in that file for reference.
You can invoke this by setting `--routing-algorithm=turn_model_oblivious`

Briefly describe the routing scheme you implemented.
Copy and paste your `outportComputeTurnModelOblivious()` implementation into the report.

You will only get points if running the network with `--routing-algorithm=turn_model_oblivious` does not deadlock for any traffic pattern. *We would recommend running your design with different traffic patterns (uniform_random, bit_complement, shuffle, transpose, etc) at extremely high injection rates to make sure your solution is robust and your network does not deadlock.*

Extra Credit (5 points)

Implement an **adaptive** version of the North-last turn model based minimal routing algorithm (**`outportComputeTurnModelAdaptive()`**). Use the number of free VCs at the next router as a proxy for choosing the output port. If all viable output ports have the same number of VCs, choose any of them randomly. *You are free to add any code/functions/header files.*
Hint: The code does not provide any function for getting the total number of free VCs and you will need to implement it. Look at the `has_free_vc()` and `select_free_vc()` functions in `OutputUnit.cc` for reference.

Part C. Deadlock Avoidance using Escape VC (25 points)

Avoid the deadlock by **implementing an escape-*VC based deadlock avoidance scheme*** in Garnet. In other words, you will run the simulation with George's random routing algorithm (`--routing-algorithm=random_oblivious`) but avoid deadlocks by controlling which VC a flit can go into.
Hint: Look at the `has_free_vc()` and `select_free_vc()` functions in `OutputUnit.cc`.
These functions are called from inside `SwitchAllocator.cc`

There are no guidelines on where to add code. In addition to `has_free_vc()` and `select_free_vc()`, depending on your design, you may restrict the routing based on which VC the flit arrived on. You are free to add any code/functions/header files. The only requirement is that the routing algorithm code inside the `outportComputeRandomOblivious()` function should not be changed.

Briefly describe the escape VC scheme you implemented and why it provides deadlock freedom. Copy-paste the relevant functions you changed/added in the code into the report.

You will only get points if running the network with `--routing-algorithm=random_oblivious` does not deadlock for any traffic pattern.
We would recommend running your design with different traffic patterns (uniform_random, bit_complement, shuffle, transpose, etc) at extremely high injection rates to make sure your solution is robust and your network does not deadlock.

Appendix: Debugging with Loupe

In this lab, we introduce a deadlock detection and visualization tool, Loupe, to help you debug your routing algorithm implementation.
See Loupe_Manual.pdf for detailed description and usage.

Part D. Analysis (7 points)

D.1. Plot the reception rate vs injection rate for the following configurations. For all configurations, start at an injection rate of 0.02 (packets/node/cycle) and increment in steps of 0.02.

Graph 1 *(2 points)*
Traffic: **`uniform_random`**
Routing: XY, Turn Model Oblivious, (Turn Model Adaptive), Random Oblivious + EscapeVC.

Graph 2 *(2 points)*
Traffic: **`transpose`**
Routing: XY, Turn Model Oblivious, (Turn Model Adaptive), Random Oblivious + EscapeVC.

Add both graphs to your report.
Clearly label all axes and legends. Otherwise you will not get any points for this part.
In each case, write down which routing algorithm provides the highest throughput.

D.2. Based on the graphs above, comment on the following: ***does better path diversity from your routing algorithm always translate to better throughput?*** If yes, why? If not, why not? For either answer, please construct some toy examples to argue your case. The example does not need to be validated via simulation but should reflect a possible scenario based on the traffic pattern and choice of routing algorithm. *(3 points – 1 point for Yes/No, 2 points for example + explanation).*

What to Submit:
Report.pdf
garnet2.0.tar.gz (a copy of your code)