Lab4 Report

Topic:

Topic 1: Topology & routing algorithm is chosen. A 3D-torus and a customized adaptive deadlock-free routing algorithm are implemented.

Notice:

Each of our group implements a topology with a customized routing algorithm. However, we will only describe the 3D torus and its customized routing in this paper. The 2D Mesh_v3 topology and its routing will not occur in this paper, except in the figures displayed in performane analysis section.

Contributors:

RUIYING MA, 2021010726, Jike 12, IIIS:

- Code: implement 3D-torus and the routing algorithm described in paper An Efficient Adaptive Deadlock-Free Routing Algorithm for Torus Networks.

  Related files:

  - `gem5\configs\network\Network.py`
  - `gem5\configs\topologies\Torus_XYZ.py`
  - `gem5\src\mem\ruby\network\garnet\CommonTypes.hh`
  - `gem5\src\mem\ruby\network\garnet\GarnetNetwork.cc`, `gem5\src\mem\ruby\network\garnet\GarnetNetwork.hh`, `gem5\src\mem\ruby\network\garnet\GarnetNetwork.py`
  - `gem5\src\mem\ruby\network\garnet\InputUnit.cc`, `gem5\src\mem\ruby\network\garnet\InputUnit.hh`
  - `gem5\src\mem\ruby\network\garnet\OutputUnit.cc`, `gem5\src\mem\ruby\network\garnet\OutputUnit.hh`
  - `gem5\src\mem\ruby\network\garnet\Router.cc`, `gem5\src\mem\ruby\network\garnet\Router.hh`
  - `gem5\src\mem\ruby\network\garnet\RoutingUnit.cc`, `gem5\src\mem\ruby\network\garnet\RoutingUnit.hh`: `RoutingUnit::outportComputeXYZ`
  - `gem5\src\mem\ruby\network\garnet\SwitchAllocator.cc`, `gem5\src\mem\ruby\network\garnet\SwitchAllocator.hh`
  - `gem5\src\mem\ruby\network\garnet\VirtualChannel.cc`, `gem5\src\mem\ruby\network\garnet\VirtualChannel.hh`

- Report
- Presentation
- Polish & complement 2 paper reading assignments

CHENGDA LU, 2021010899, Jike 12, IIIS:

- Code: implement 2D Mesh_v3 (his own idea) and its routing algorithm

  Related files:

  - `gem5\configs\network\Network.py`
  - `gem5/configs/topologies/Mesh_v3.py`
  - `gem5/src/mem/ruby/network/garnet/CommonTypes.hh`
  - `gem5/src/mem/ruby/network/garnet/RoutingUnit.cc`, `gem5/src/mem/ruby/network/garnet/RoutingUnit.hh`: `RoutingUnit::outportComputeXYV3`

- Test the performance
- Presentation
- Sketch 2 paper reading assignments

References:

The adaptive deadlock-free routing algorithm tailored to nD-torus is provided by this paper: An Efficient Adaptive Deadlock-Free Routing Algorithm for Torus Networks.

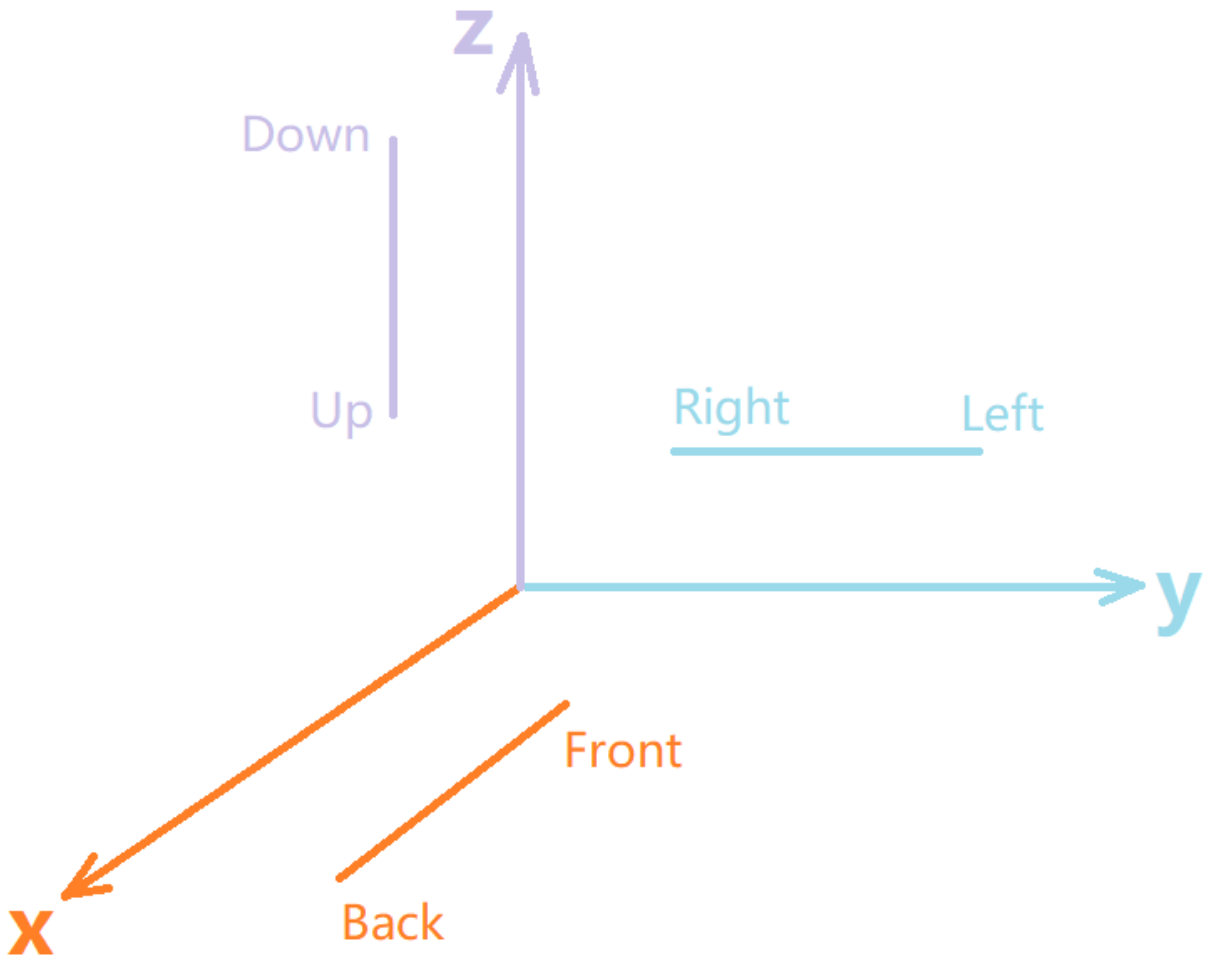
Toplogy

- Description

  The topology we choose is 3D-torus. Each router has 6 ports and 12 `IntLink`s: imagine a router as a cube, of which each surface has two `IntLink`s, one goes into the surface while the other exits the the surface. Each router is assigned with a cache-controller and a directory-controlled, and each controller is connected to its router with one `ExtLink`.

- Implementation

  The implementation of the `IntLink`s of a router looks like this:



The 6 surfaces are named as Front, Back, Left, Right, Up and Down. The line segments in this figure represent the links.

Routing Algorithm

- Algorithm

  Suppose an `IntLink` R has $2v$ virtual channels. We divide an `IntLink` into two virtual links, calling them R_1 and R_2. R_1 occupies the first v virtual channels of R, while R_2 occupies the remaining v virtual channels of R.

  We make it a rule that for a link along the i-axis (i can be one of x, y, z), if it is going along the positive direction along the axis and occupying the j-th virtual link (j can be one of $1, 2$), we denote is as $R_{j,i+}$. Otherwise it is going along the negative direction along the axis, we denote is as $R_{j,i-}$. For example, if a link is along the x-axis, travelling along the positive direction and occupying the first virtual link, it is denoted as $R_{1,x+}$.

  We also make it a rule that a router is denoted by its coordinate (x, y, z).

  We also call link that connect the first and the last routers along each direction as the "wraparound" link.

  Suppose the packet is now at router (x_0, y_0, z_0) and is heading for the destination (x_1, y_1, z_1). The routing algoritm goes like this:

0. The set S records the output links that can be chosen. Initially, $S = \phi$.

1. Decide which outport can be chosen

   Take direction x as an example:

   - if $x_0 = x_1$, neither of the Front of Back outports can be chosen.
   - if $x_0 \neq x_1$, one of the Front of Back outports can be chosen. Whether to choose the Front or the Back outport depends on which outport is closer to the destination along the x-direction. (In other words, we always stick to the minimal paths.)

   The same for direction y and z.

2. Decide whether the two virtual links R_1 or R_2 can be chosen.

   - For R_1 links:

     Take direction x as an example:

     - If $x_0 \neq x_1$, suppose, w.l.o.g., that the Front outport is chosen. This means that the link R_x+ can be taken into consideration. Then, we put its first link into S: $S \leftarrow S \cup R_{1,x+}$.
     - If $x_0 = x_1$, no x-direction links are added into S.

     Same for y, z direction.

   - For R_2 links:

     - If the minimal paths to the destination need to go through some wraparound link(s).

       Suppose the minimal dimension of the inevitable wraparound link is i ($i \in x, y, z$).

       If the coordinate $i_0 == 0$ or $i_0 == k - 1$ (k is the number of routers along dimension i), then the link $R_{2,i+}$ (if $i_0 == n - 1$) or the link $R_{2,i-}$ (if $i_0 == 0$) can be added to S.

     - If the minimal paths to the destination need not to go through any wraparound link.

       Suppose i is the minimal dimension such that coordinate $i_0 \neq i_1$. Then the link $R_{2,i+}$ or $R_{2,i-}$ can be added to S. Whether $+$ or $-$ has been decided in step 1 by the direction of the outport.

   The steps above decide the S sets. These are done in the route-compute unit of a router.

   Later during the process of allocating output virual channels, one of the channels in S which are currently free (for head flits) or currently have free credits (for body flits) will be selected. This selection can be done randomly (or apply other selection function), which is where the adaptivity of this algorithm is applied.

- Implementaion

The computation of the set of available channels S are done in `RoutingUnit::ComputeXYZ`(`gem5/src/mem/ruby/network/garnet/RoutingUnit.cc`). Different from some other routing algorithm that returns one determined outport, `RoutingUnit::ComputeXYZ` returns all the possible outports plus available virtual links.

To support two virtual links $R_{1,2}$ for each physical link R, the virtual channels of each physical link should be divided into two sets as described before. When a virtual link, say R_1 (or R_2), can be chosen, the `SwitchAllocator::torus_send_allowed` process (a process similar to `SwitchAllocator::send_allowed`, implemented by us) should test only the first (or second) half of the virtual channels of R, and only those in S that pass this test can be finally selected.

The details of the implementation can be seen in the code.

- Sketch proof of deadlock freedom

  The details can be seen in the paper mentioned above.

How to run

- Set `--topolgy=Torus_XYZ`
- Set `--torus-xs=a` and `--torus-ys=b` to create a 3D-torus with a routers along the x-axis and b routers along the y-axis.

  Caution: For a $a \times b \times c$ torus, it is required that $a, b, c > 2$.
- Set `--routing-algorithm=3` to use the customized routing.
- Example

```
./build/NULL/gem5.opt configs/example/garnet_synth_traffic.py \
--network=garnet --num-cpus=64 --num-dirs=64 \
--topology=Torus_XYZ --torus-xs=4 --torus-ys=4 --routing-algorithm=3 \
--inj-vnet=0 --synthetic=uniform_random \
--sim-cycles=10000 --injectionrate=0.01
```

Deadlock Test

Set `sim-cycles` as 100000, set `injectionrate` as 1. Change `synthetic` among all five possible choices (uniform_random, neighbor, tornado, transpose, shuffle). For example, run

```
./build/NULL/gem5.opt \
configs/example/garnet_synth_traffic.py \
--network=garnet --num-cpus=64 --num-dirs=64 \
--topology=Torus_XYZ --torus-xs=4 --torus-ys=4 --routing-algorithm=3 \
--inj-vnet=0 --synthetic=uniform_random \
--sim-cycles=100000 --injectionrate=1 \
--vcs-per-vnet=16 \
```

Check whether in some case there is a deadlock occurs. It turns out that no deadlock has ever occurred.

Tests and Performance

Run

```
./build/NULL/gem5.opt configs/example/garnet_synth_traffic.py \
--network=garnet --num-cpus=64 --num-dirs=64 \
--topology=Torus_XYZ --torus-xs=4 --torus-ys=4  \
--inj-vnet=0 --synthetic=uniform_random \
--sim-cycles=10000 --injectionrate=0.01
```

We can get the result:

	64-**Ring**	8×8-**Mesh**	$4\times 4\times 4$ -**Torus**
packets_injected	3165	3165	3165
packets_received	3155	3162	3162
average_packet_queueing_latency	2	2	2
average_packet_network_latency	35.123930	13.557559	8.980076
average_packet_latency	37.123930	15.557559	10.980076
average_hops	16.025674	5.269540	2.985136
reception_rate	0.009859	0.009881	0.009881

Note that here the routing algorithm is the default `TABLE_`.

We can see that topology affects *packets_received*, *average_packet_network_latency*, *average_hops* and *reception_rate*. The peformance of different topologies in descending order is:

$$4\times 4\times 4\text{-Torus} > 8\times 8\text{-Mesh} > 64\text{-Ring}.$$

This is easy to see by estimating the $\mathrm{average\_hops}$ of three different topologies. Suppose there are N routers in total, we have already known that:

$$\mathrm{average\_hops}_{\mathrm{ring}} \approx \frac{N}{4}, \text{ and}$$

$$\mathrm{average\_hops}_{\mathrm{mesh}} = \frac{2}{3}\sqrt{N}.$$

From $\mathrm{average\_hops}_{\mathrm{ring}}$ we can easily deduce that

$$\mathrm{average\_hops}_{\mathrm{torus}} \approx \frac{3\sqrt[3]{N}}{4}.$$

This can explain why torus topology reduces average hops.

Then we apply the customized routing for $4\times 4\times 4$ -Torus for further exploration.

We tests among different synthetic traffics (neighbor, shuffle, tornado, transpose, uniform_random), vcs-per-vnet (2, 4) and topologies (64-Ring, 8×8-Mesh, $4 \times 4 \times 4$ -Torus). If the topology is $4 \times 4 \times 4$ -Torus, the routing algorithm is the newly-implemented customized algorithm; otherwise, the routing algorithm is the default one (`TABLE_`).

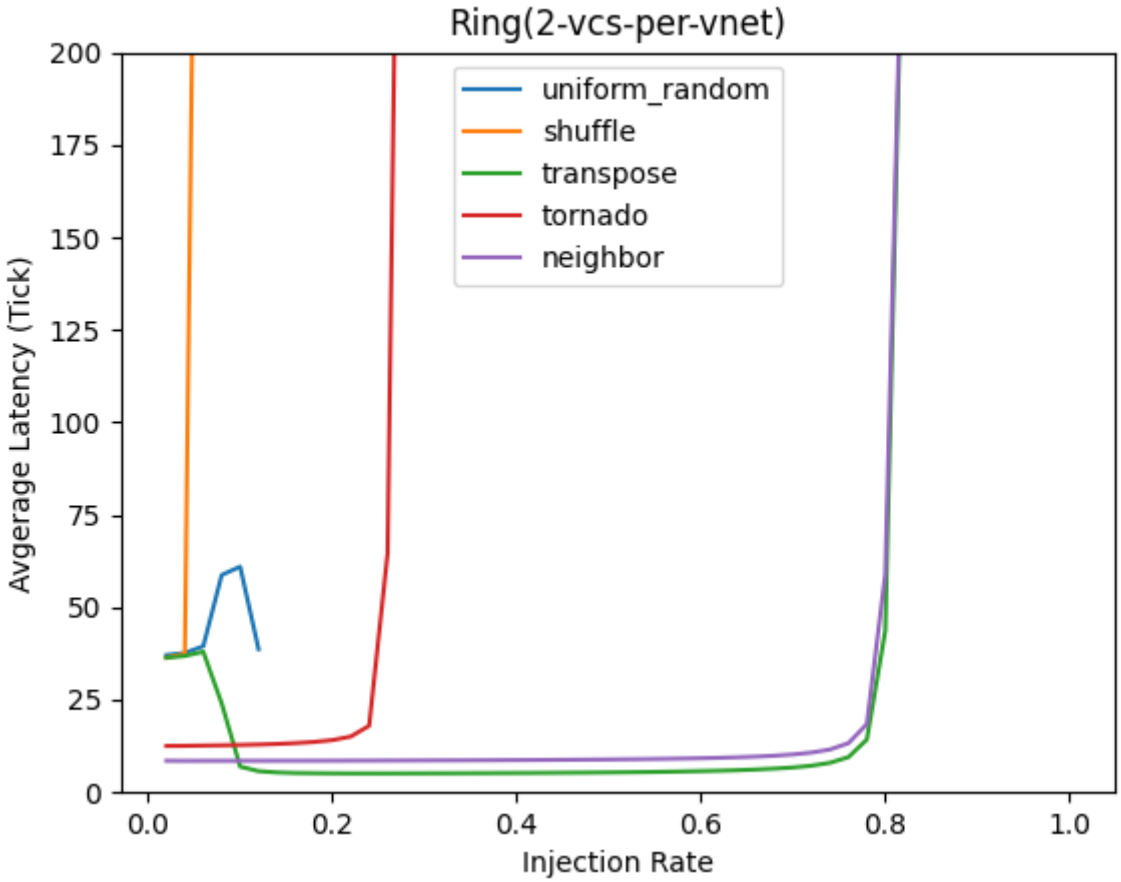
The interval of injection rate is chosen as 0.02, ranging from 0.02 to 1. The tests is forced to terminate as long as $\frac{\text{packets\_received}}{\text{packets\_injected}} \leq 0.4$. We only point those data with average latency less than 200 ticks on the graphs.

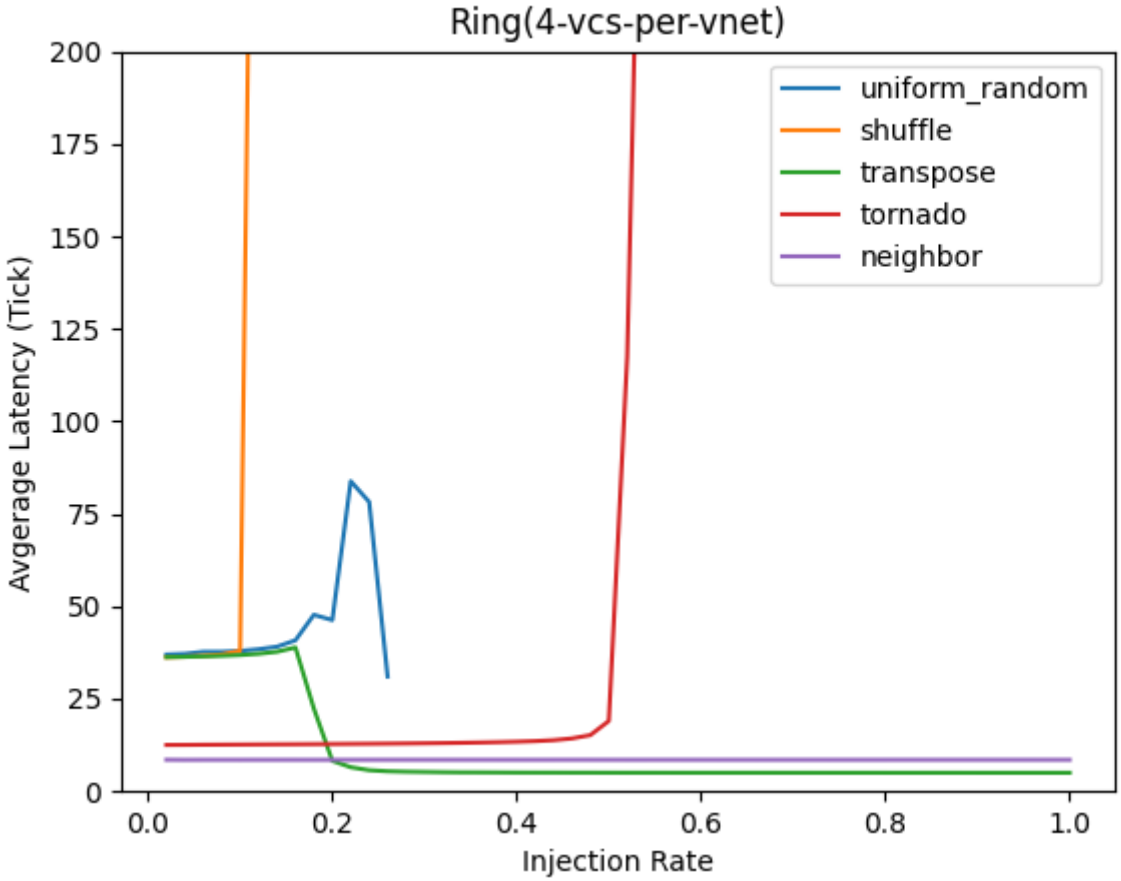
Take `--synthetic=neigbor`, `--vcs-per-vnet=2`, `--topology=Torus_XYZ`, `--torus-xs=4`, `--torus-ys=4`, `--routing-algorithm=3` as an example: run

```
inj_rate=0.020
rec_rate=1.000
echo "neighbor"
while [[ `echo "$inj_rate <= 1.000" | bc` -eq 1 && `echo "$rec_rate > 0.400" | bc`
-eq 1 ]]
do
    echo $inj_rate
    ./build/NULL/gem5.opt \
    configs/example/garnet_synth_traffic.py \
    --network=garnet --num-cpus=64 --num-dirs=64 \
    --topology=Torus_XYZ --torus-xs=4 --torus-ys=4 --routing-algorithm=3 \
    --inj-vnet=0 --synthetic=neighbor \
    --sim-cycles=20000 --injectionrate=$inj_rate \
    --vcs-per-vnet=2 --router-latency=1

    ./network_stats.sh

    packets_injected=$(grep "packets_injected" network_stats.txt | awk -F ' '
'{print $3}')
    packets_received=$(grep "packets_received" network_stats.txt | awk -F ' '
'{print $3}')
    rec_rate=`echo $packets_received $packets_injected|awk '{printf "%0.6f\n",
$1/$2}'`
    grep "average_packet_latency" network_stats.txt | awk -F ' ' '{print $3}' |
sed "s/^/${rec_rate}\t/" | sed "s/^/${inj_rate}\t/" >>
../gem5_test/Torus4x4x4/neighbor/2-vcs-per-vnet.txt
    inj_rate=`echo $inj_rate |awk '{printf "%0.3f\n", $1 + 0.02}'`
done
```

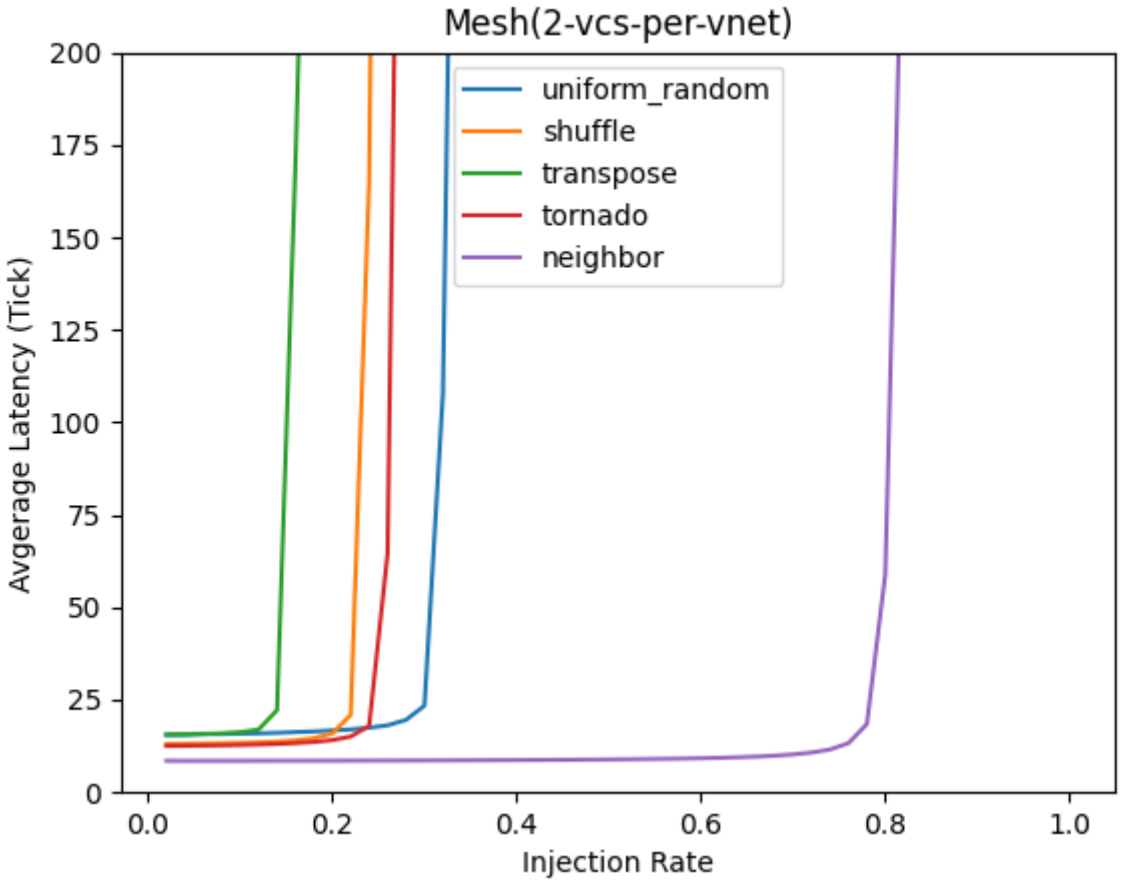
For each topology, when changing synthetic traffic and vcs-per-vnet, the results look like:

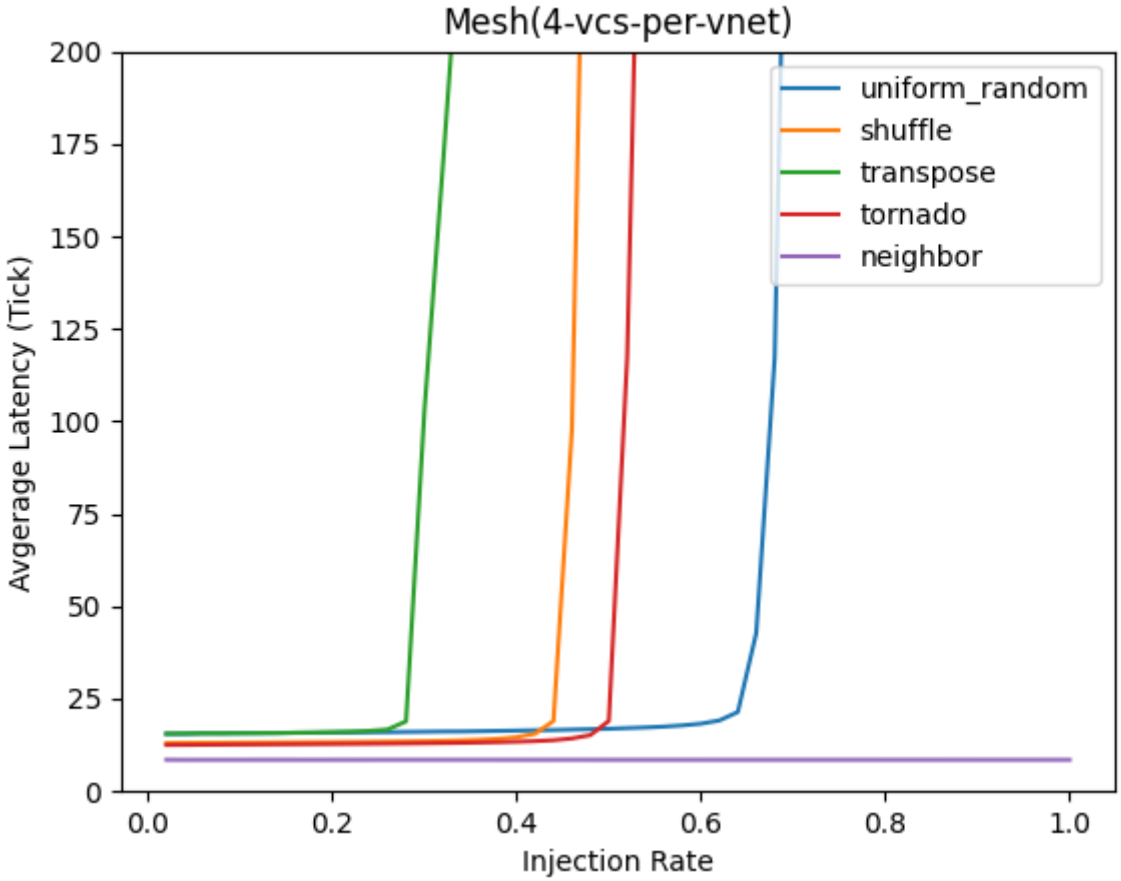
- 64-Ring



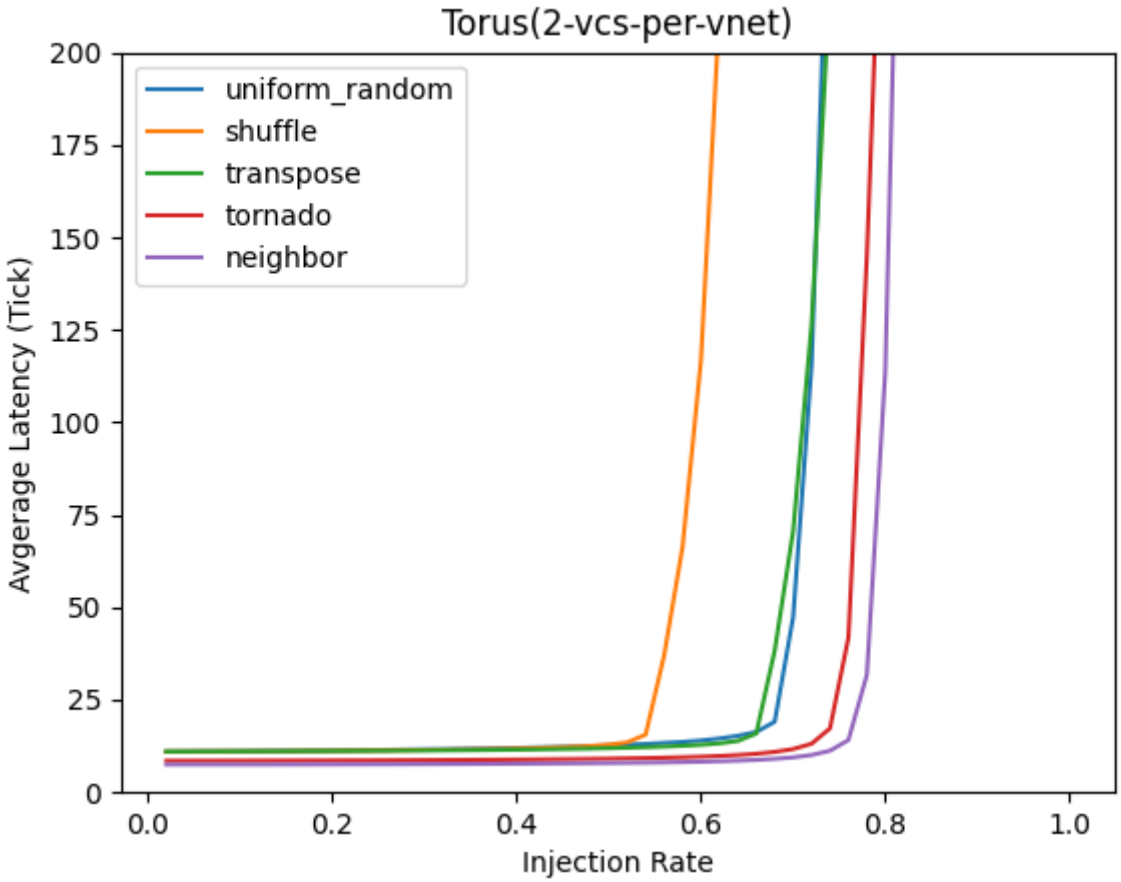


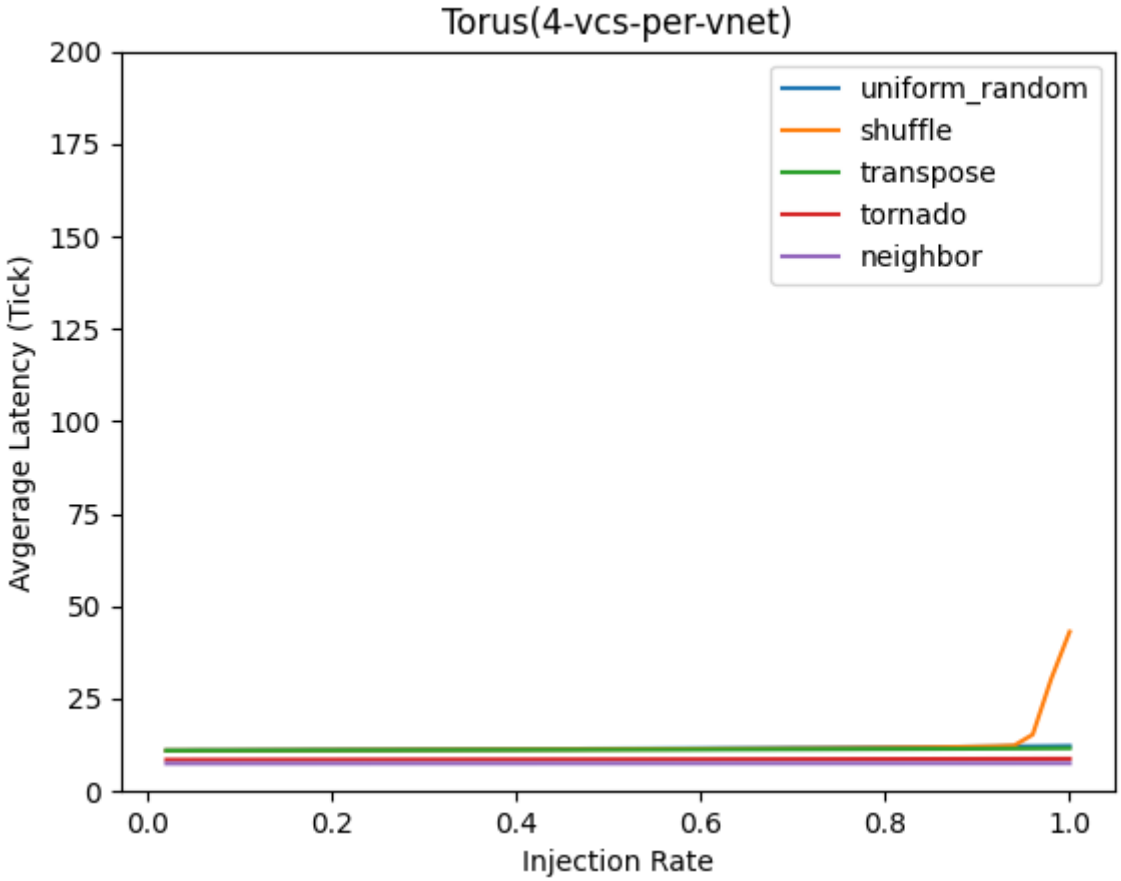
- 8×8-Mesh



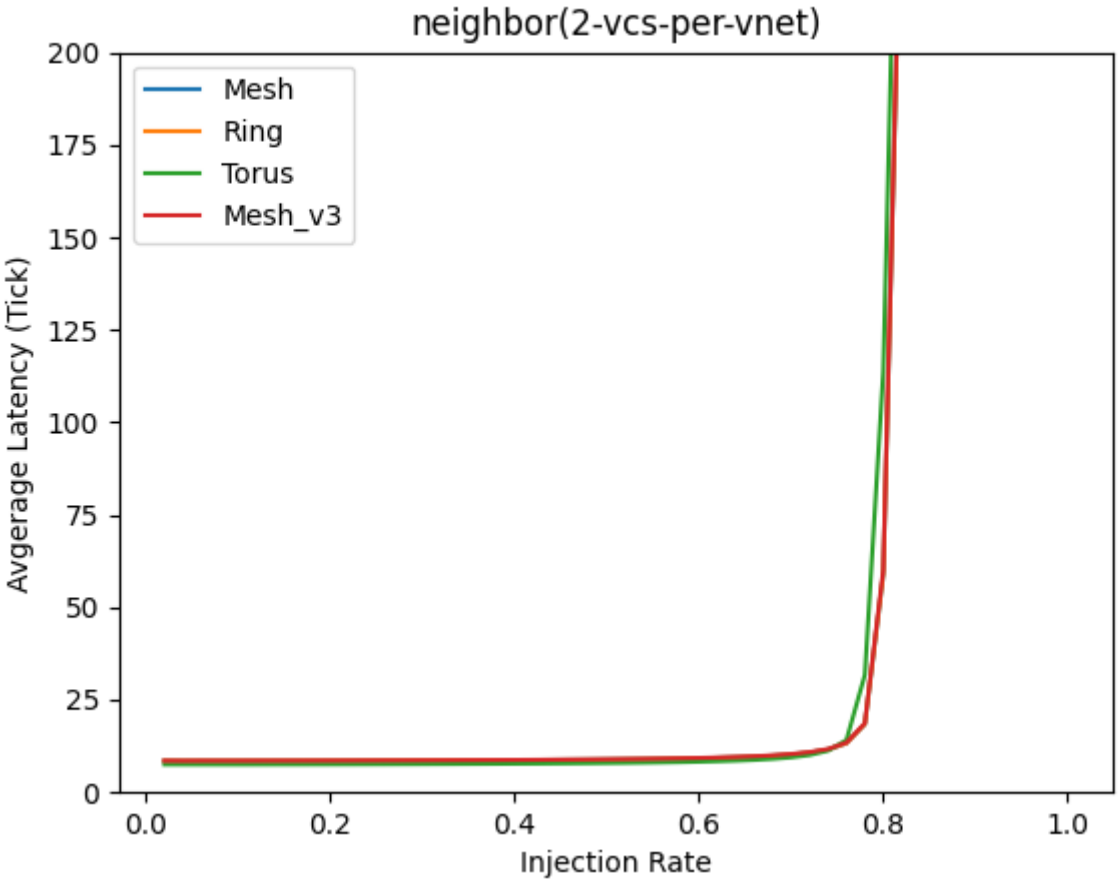


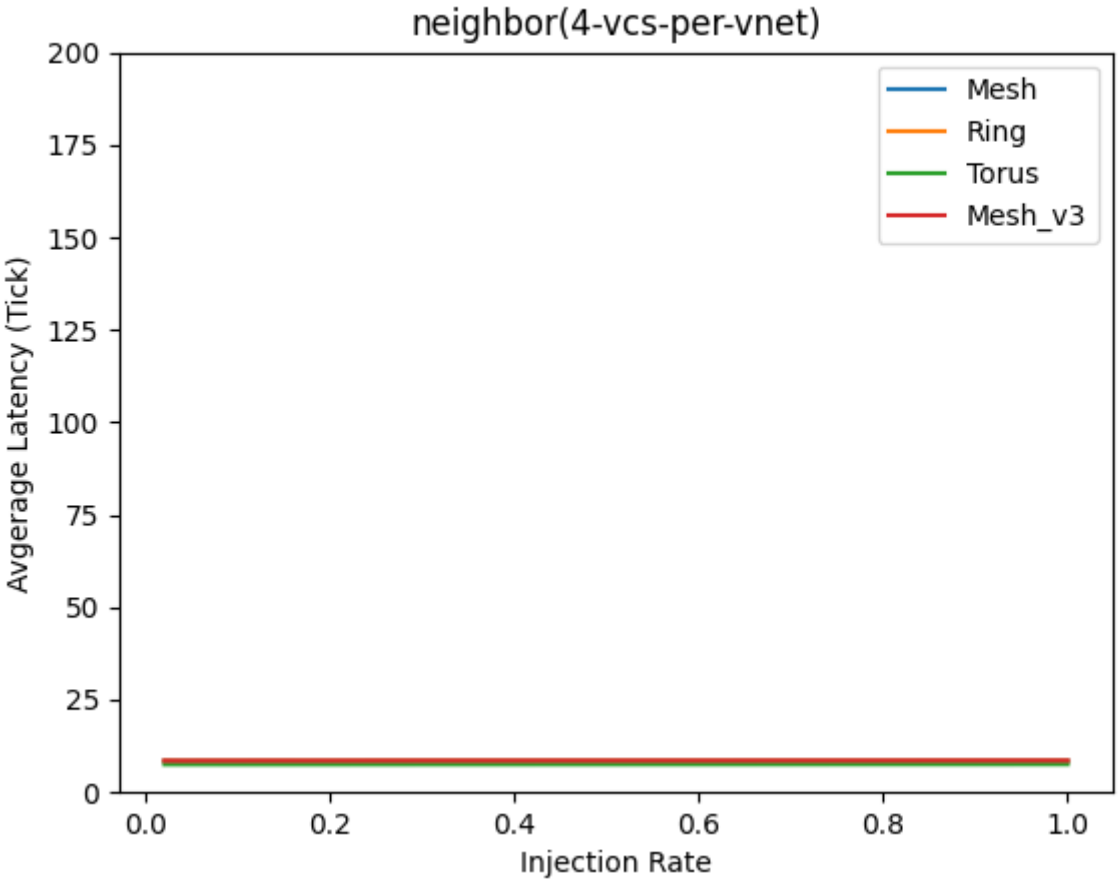
- $4 \times 4 \times 4$ -Torus

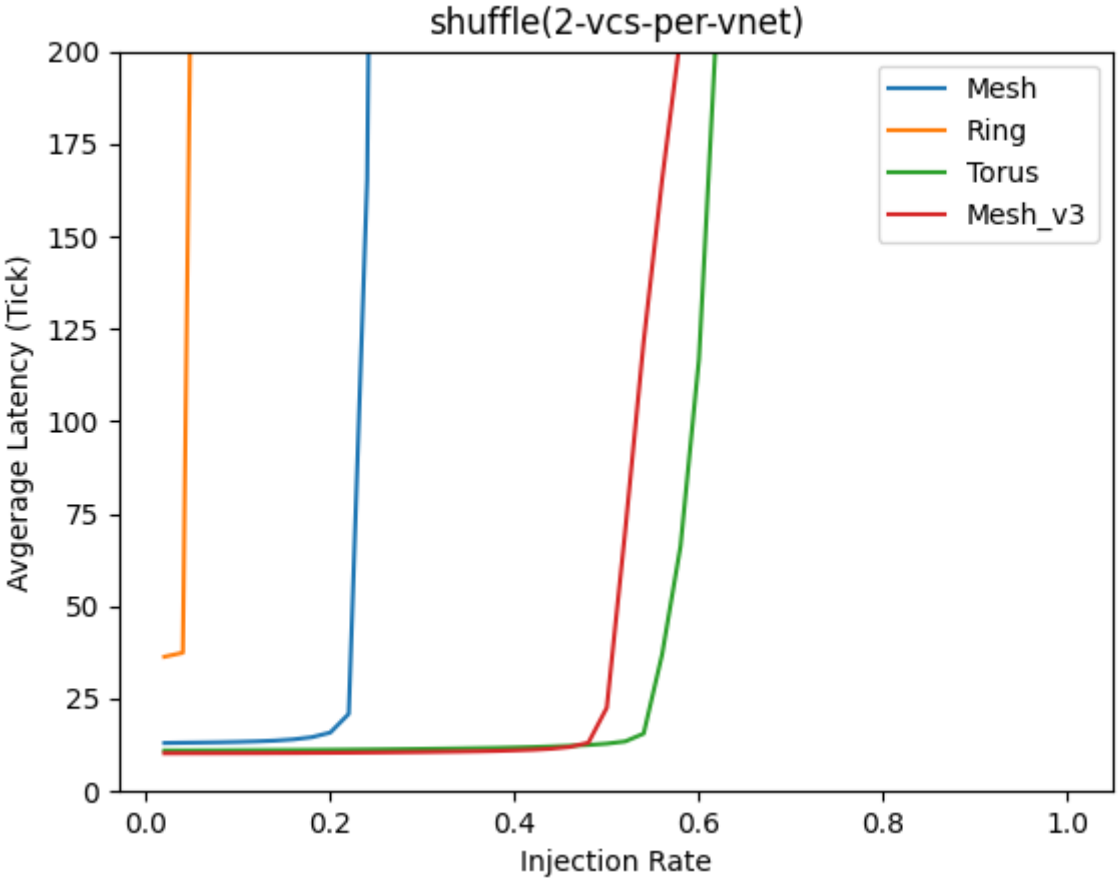

Torus(2-vcs-per-vnet)
uniform_random
shuffle
transpose
tornado
neighbor
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

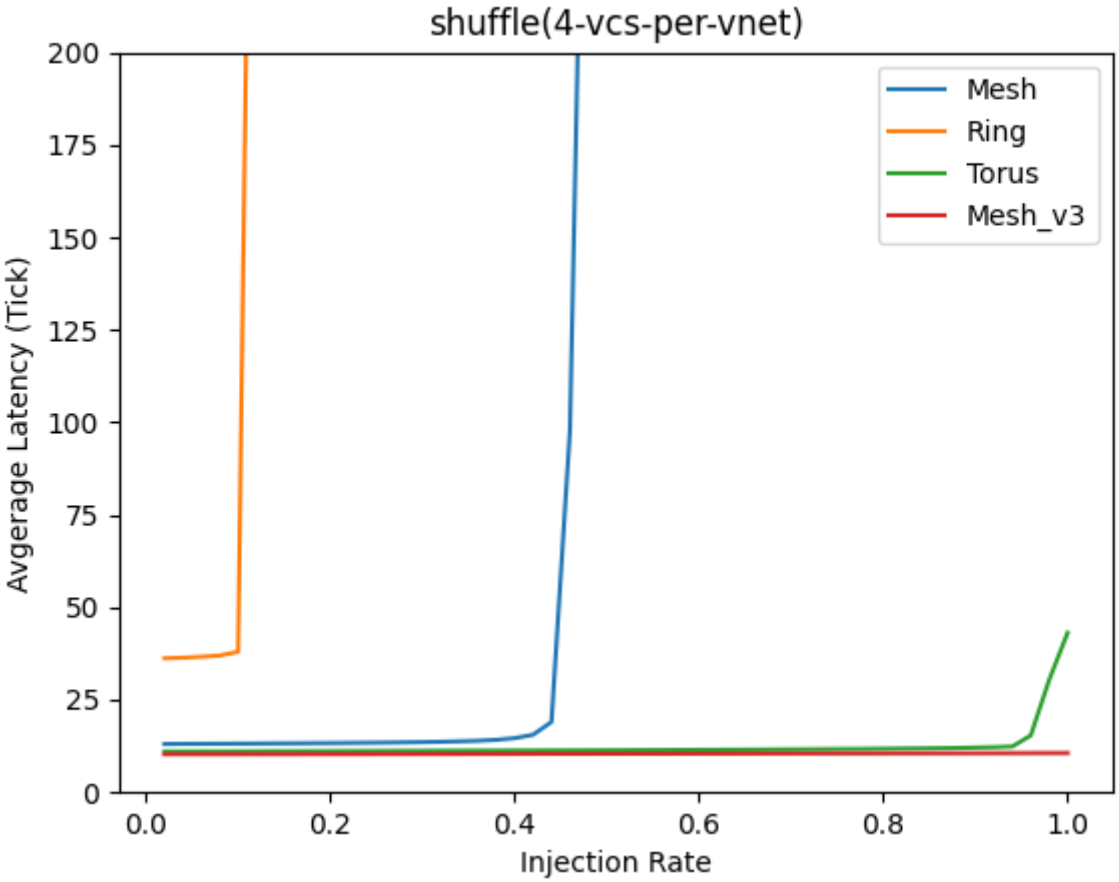

Torus(4-vcs-per-vnet)
uniform_random
shuffle
transpose
tornado
neighbor
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

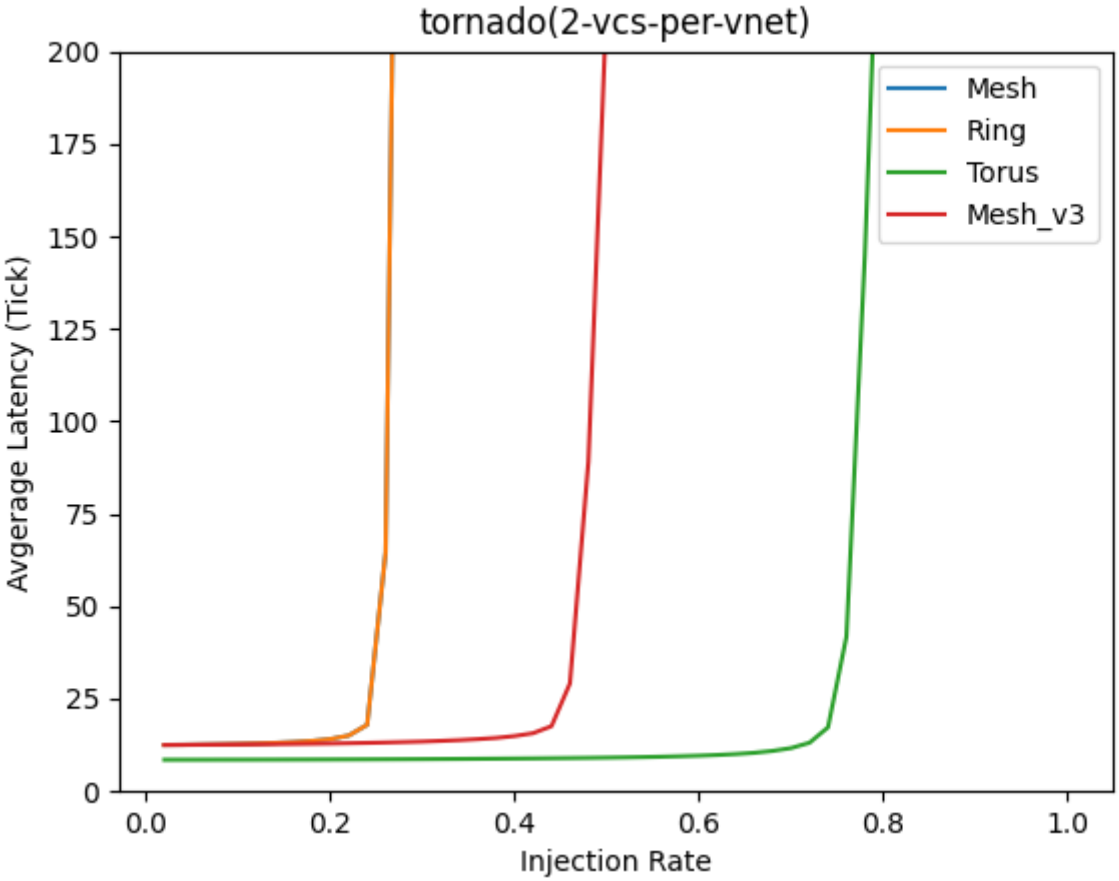
It is clearly that when the network is congested, $4 \times 4 \times 4$ -Torus with customized routing (adaptive and deadlock-free) has lower average latency and is more tolerant to congested network. It is also obvious that with more vcs-pre-vnet, the network can have better performance. For a clearer vision, we compare among different topologies under same conditions:

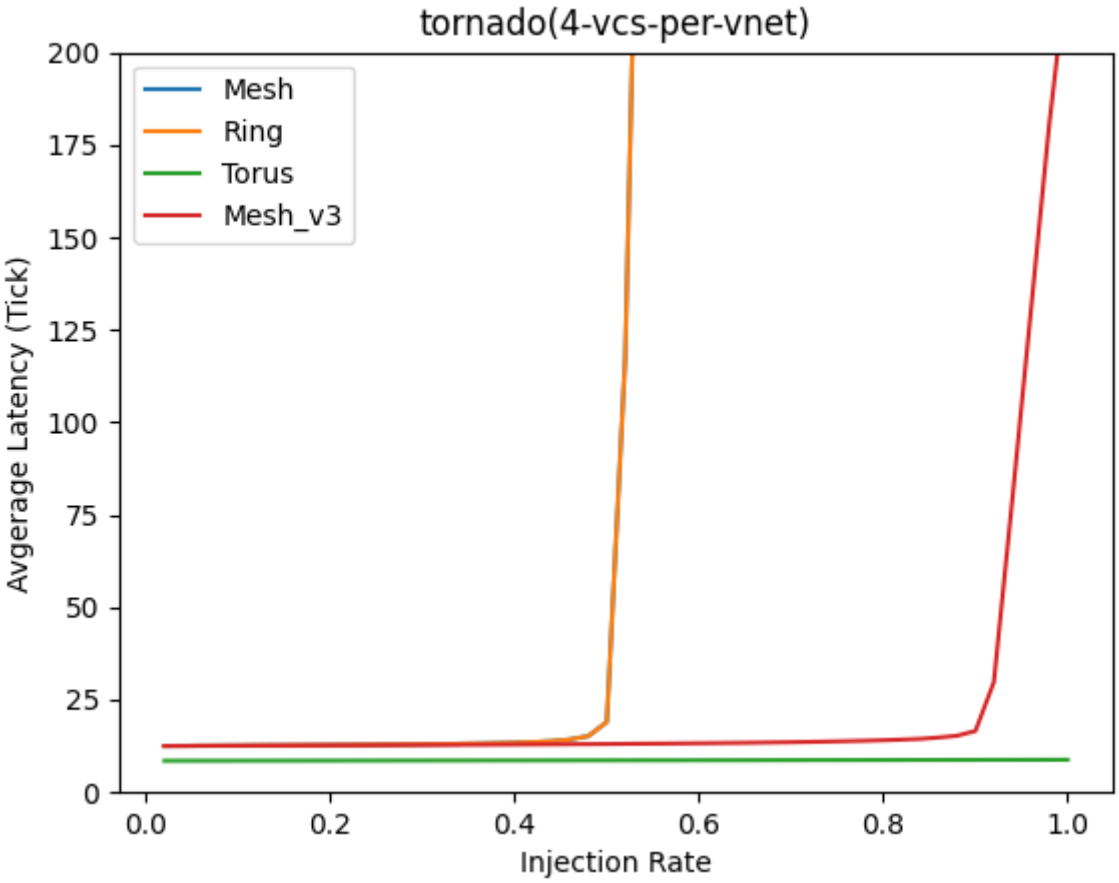


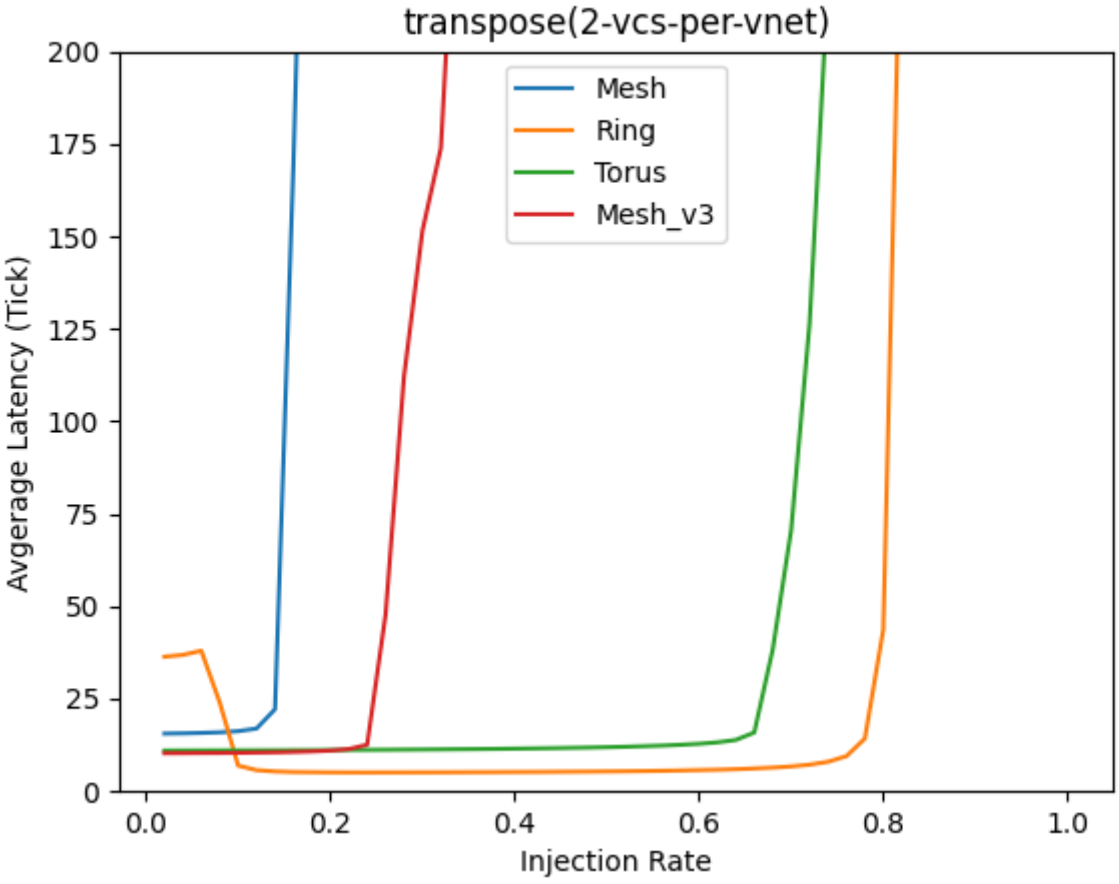

neighbor(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
Mesh
Ring
Torus
Mesh_v3
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

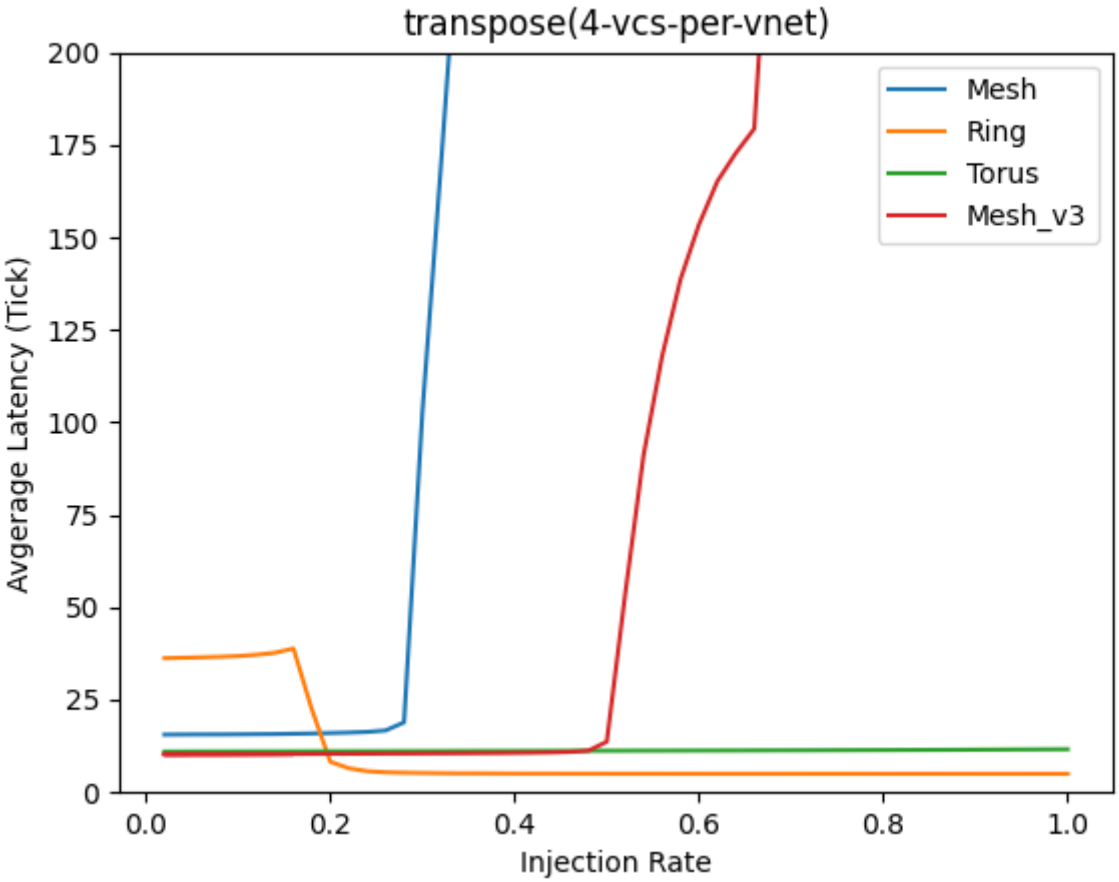

shuffle(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
Mesh
Ring
Torus
Mesh_v3
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

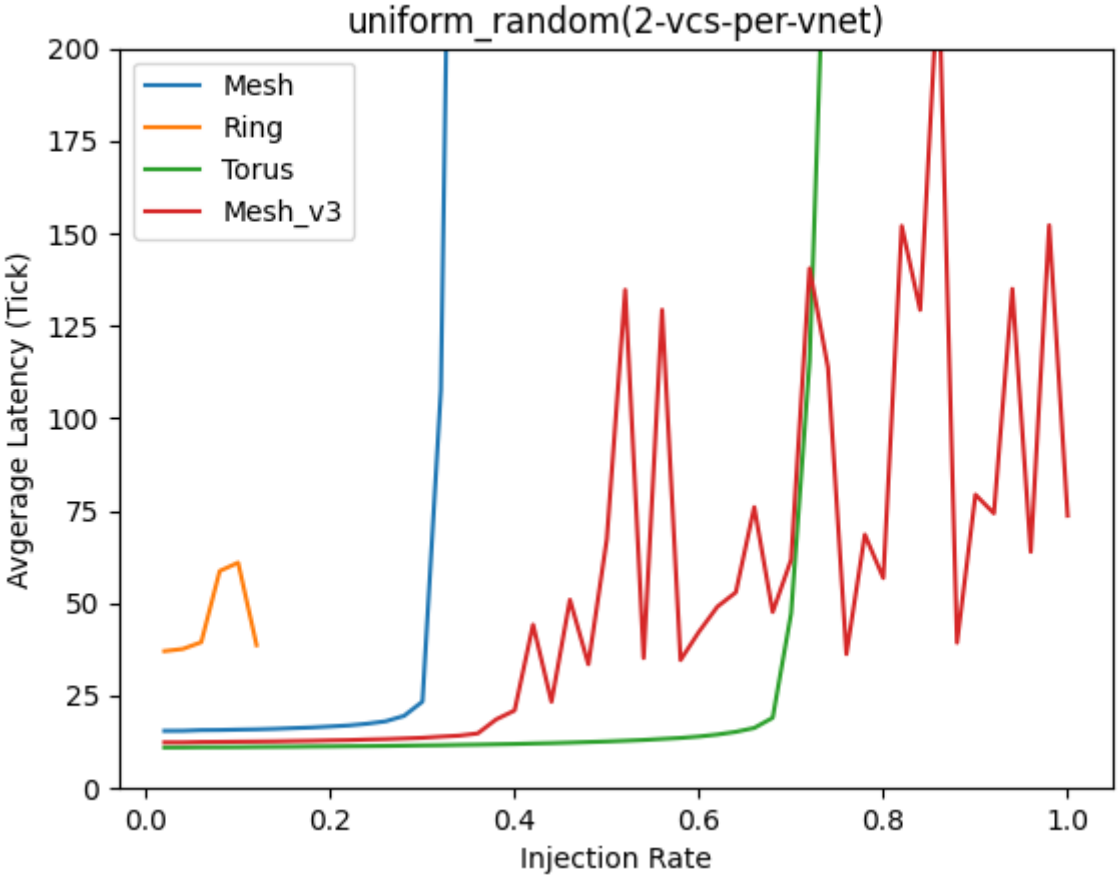

shuffle(4-vcs-per-vnet)
Mesh
Ring
Torus
Mesh_v3
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

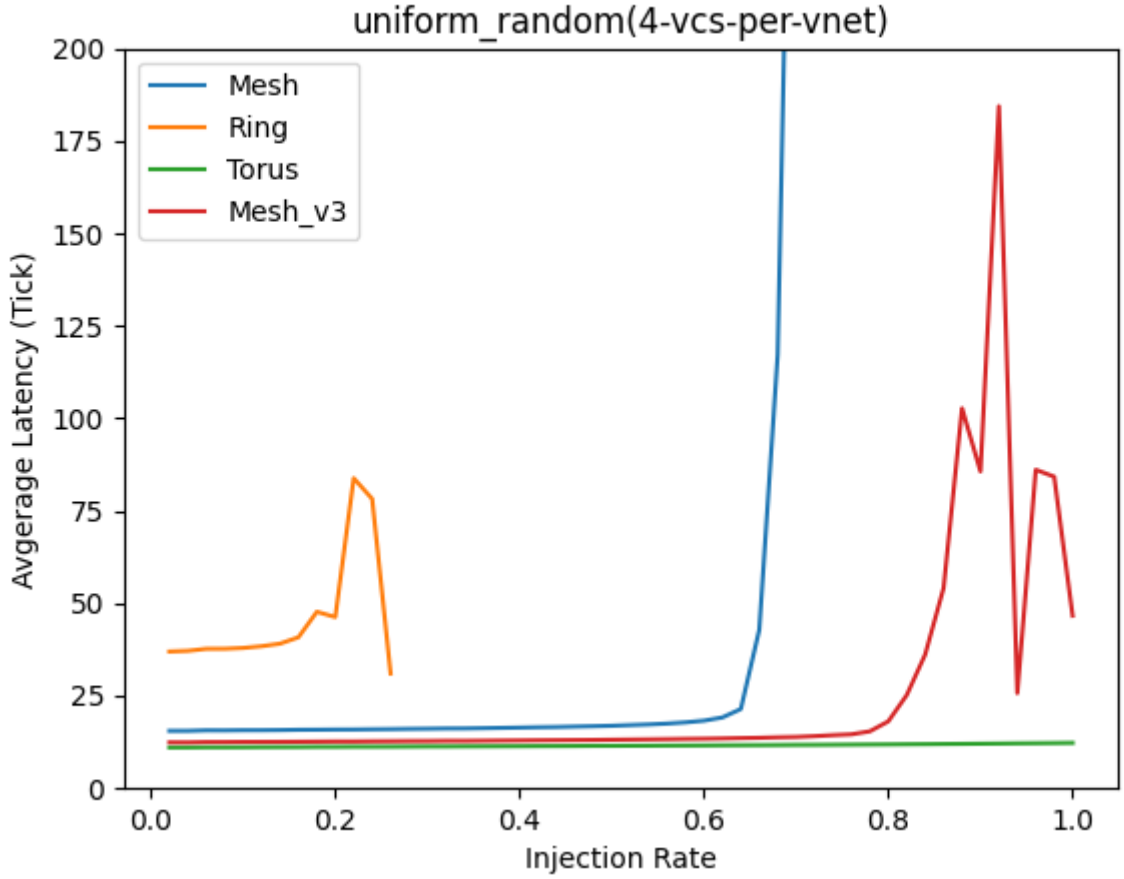

tornado(2-vcs-per-vnet)
Mesh
Ring
Torus
Mesh_v3
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0


tornado(4-vcs-per-vnet)
Mesh
Ring
Torus
Mesh_v3
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0


transpose(2-vcs-per-vnet)
Mesh
Ring
Torus
Mesh_v3
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0


transpose(4-vcs-per-vnet)
Mesh
Ring
Torus
Mesh_v3
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0


uniform_random(2-vcs-per-vnet)
Mesh
Ring
Torus
Mesh_v3
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0



For more detailed data, see Appendix.

Analysis

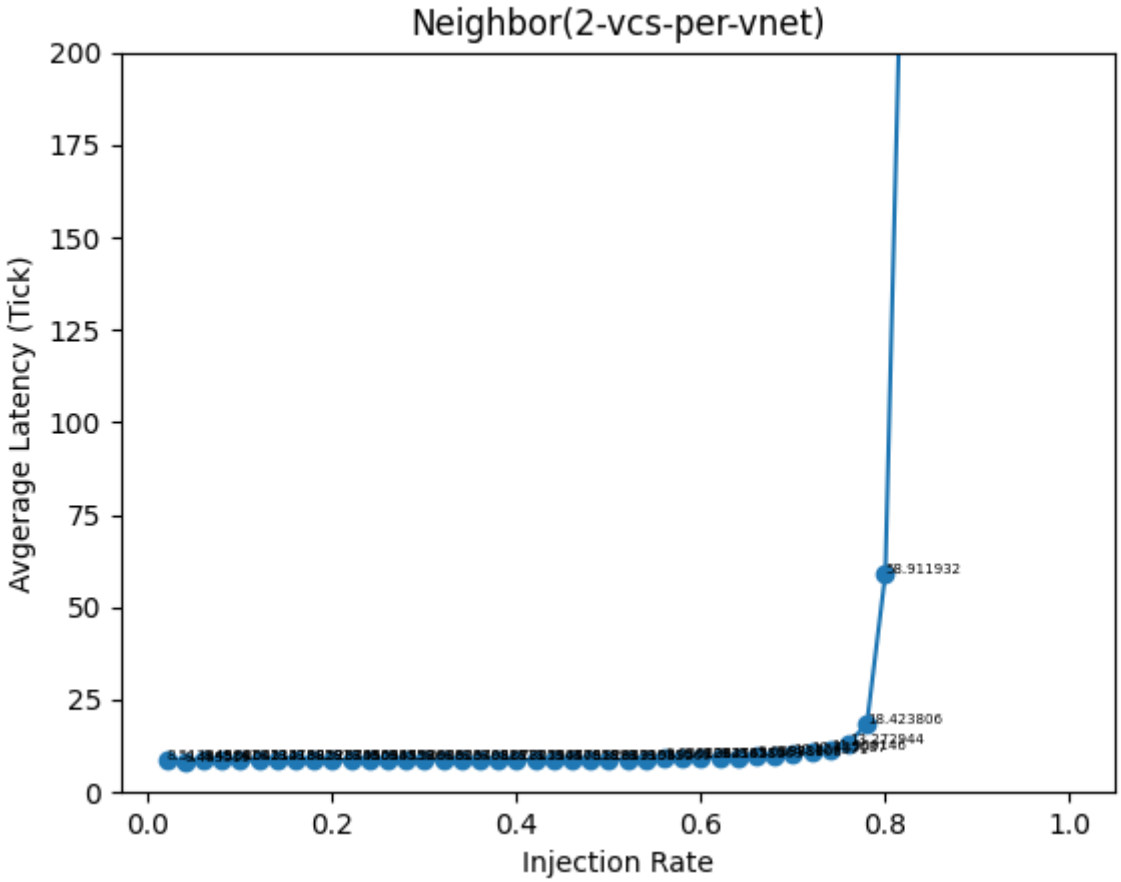
Theoritically, 3D-torus can improve the throughput by reducing the average hops a packet need to travel. Meanwhile, the adaptive routing algorithm also alleviate the problem of congestion when the network has a heavy load. However, to guarantee deadlock freedom, a packet cannot choose arbitrarily among all possible virtuall channels. Rather, virtual channels are divided into two sets R_1 and R_2 such that in some cases only one set of virtual channels can be chosen. This can harm the network performance.

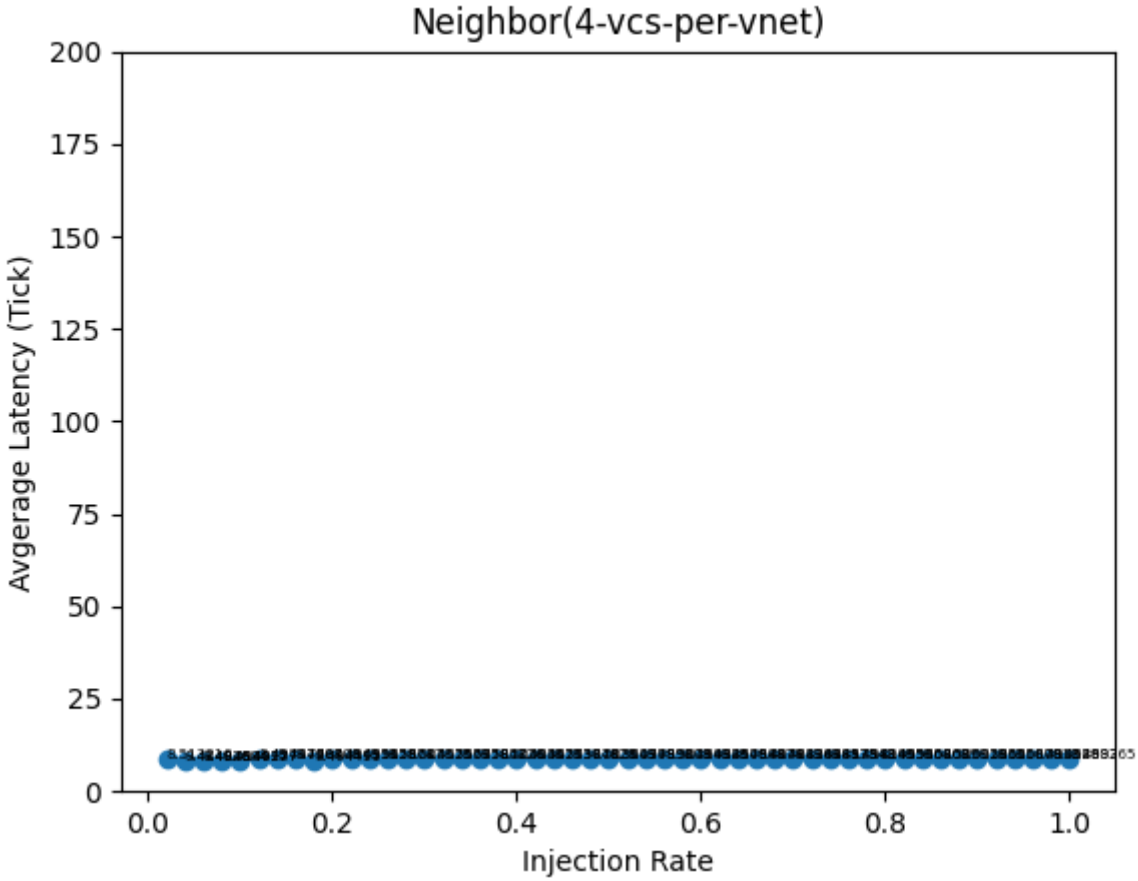
Practically, the results displayed above indicate that in total the $4 \times 4 \times 4$ -Torus with its customized algorithm successfully improves the network's performance in most cases. When the injection rate is high (which means that the network has a heavy load), the torus with adpative routing is less likely to get congested, since it has a higher threshold of injection rate when the curve of througput begins to surge.

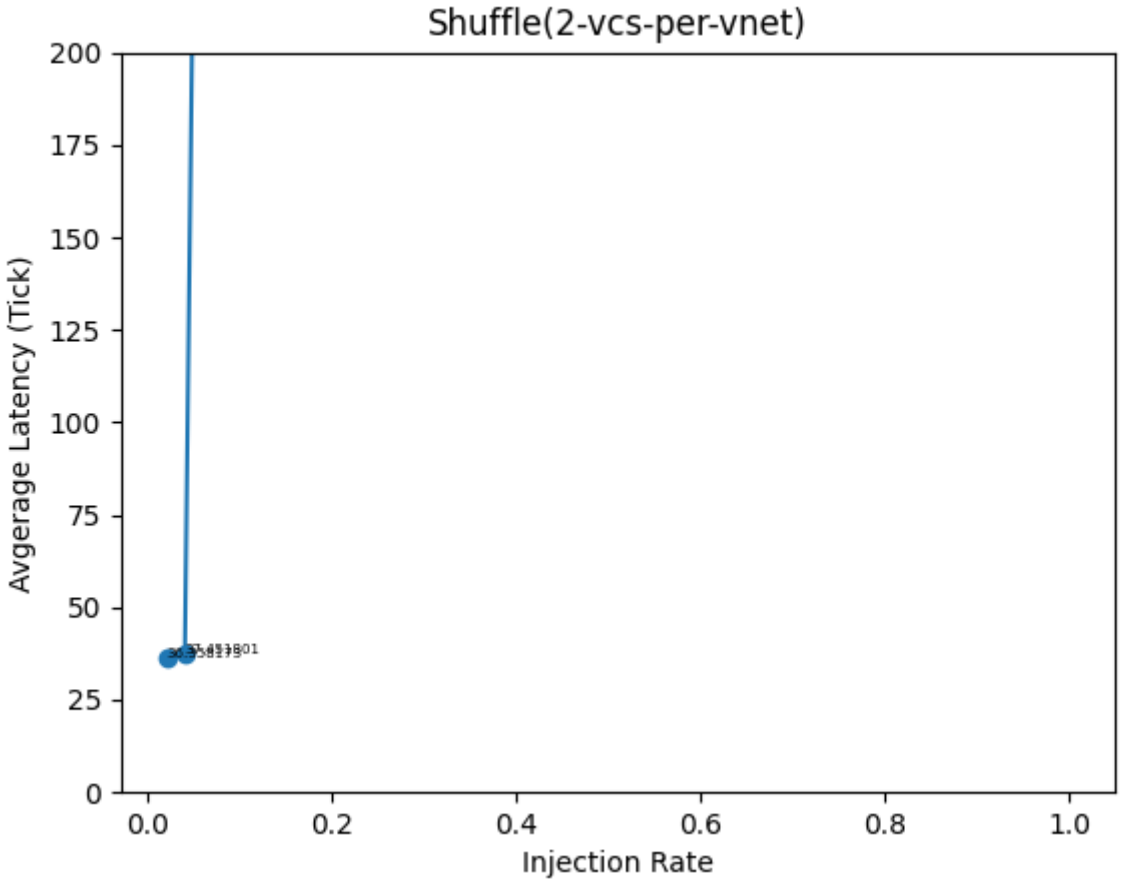
Appendix

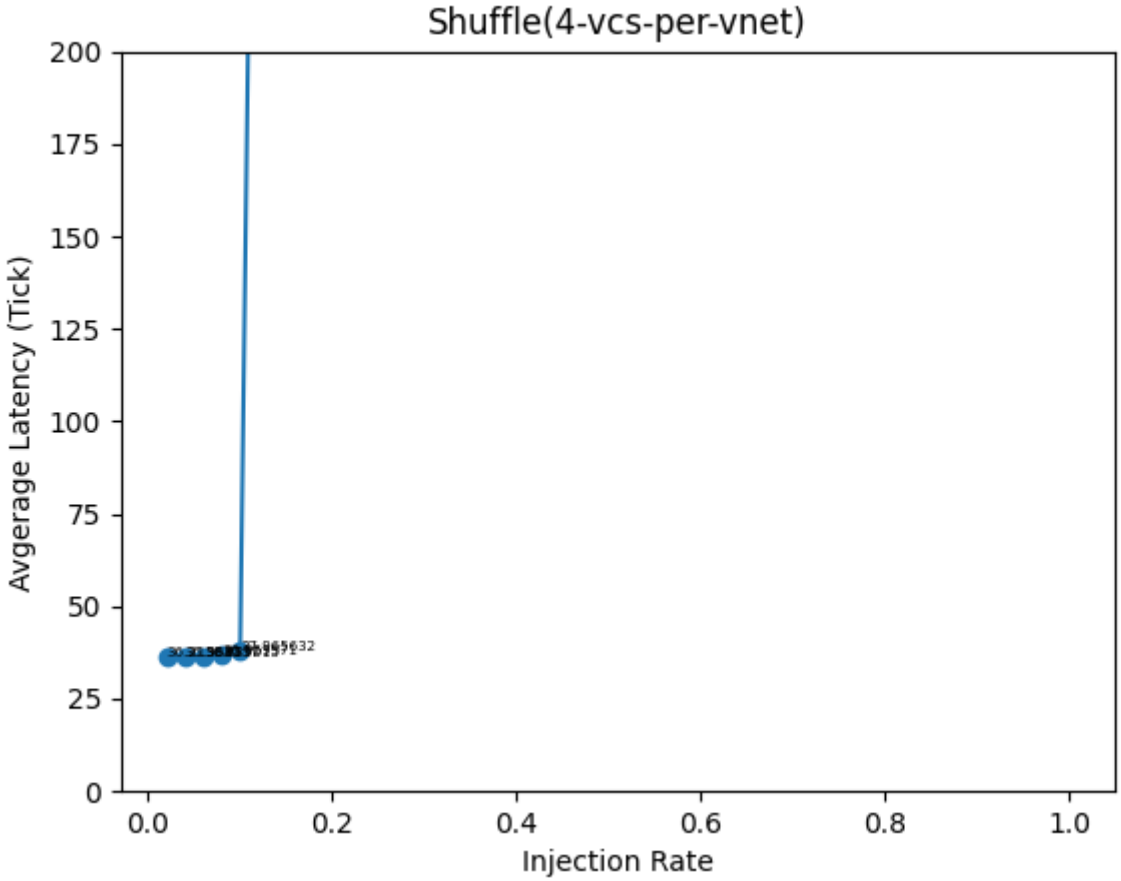
The detailed curves are shown below:

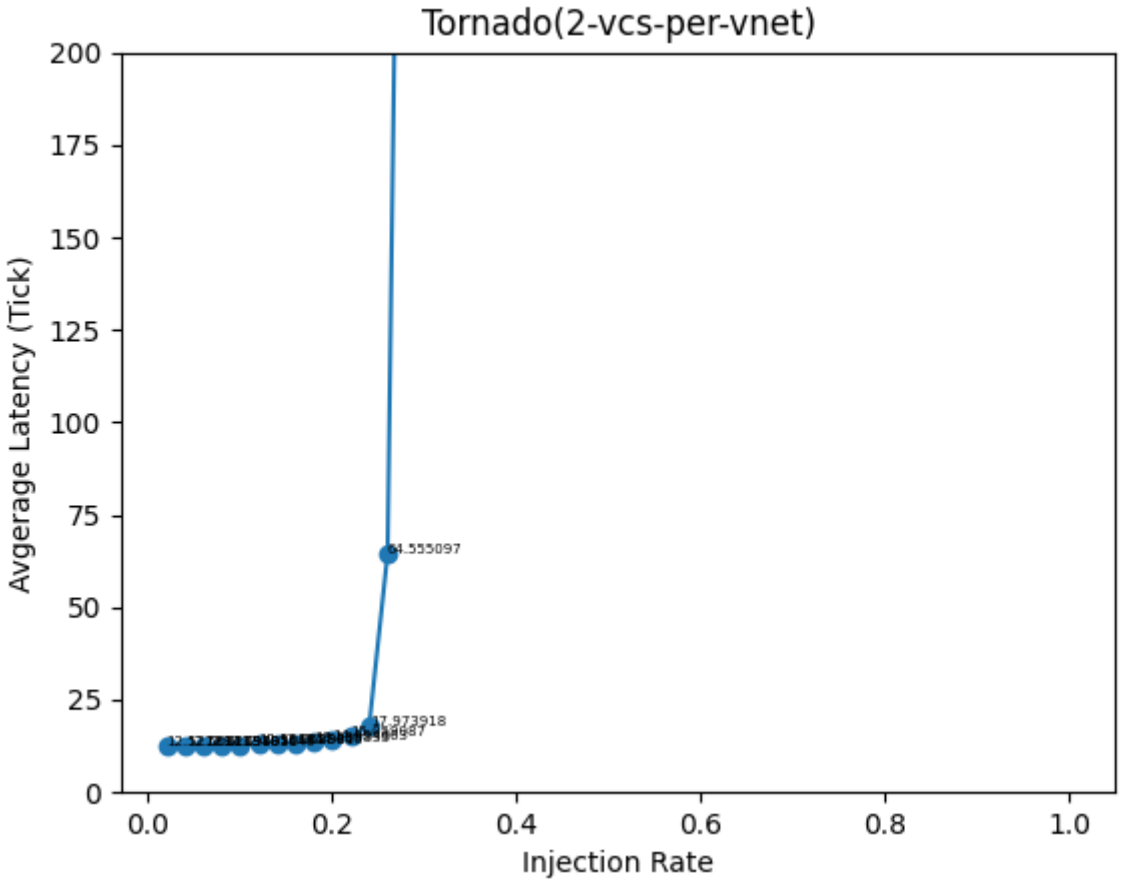
- 64-Ring

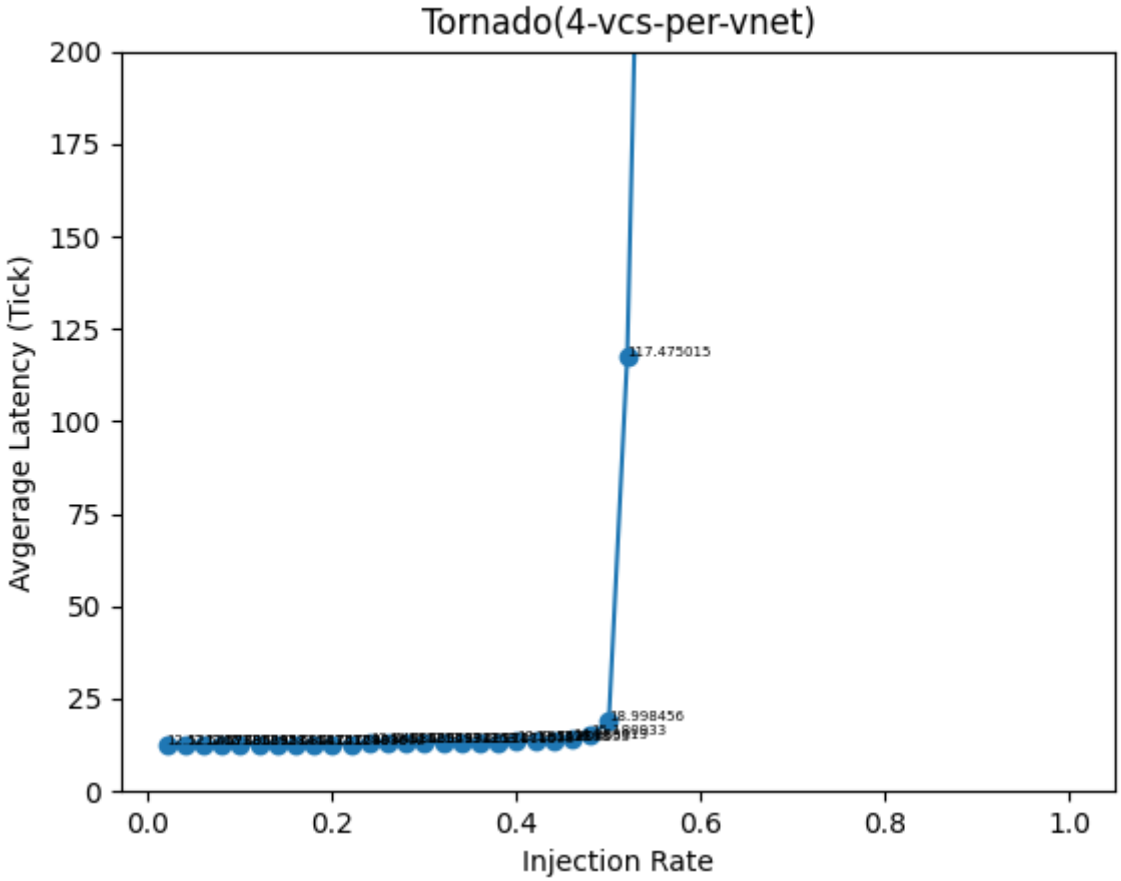

Neighbor(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
58.911932
18.423806

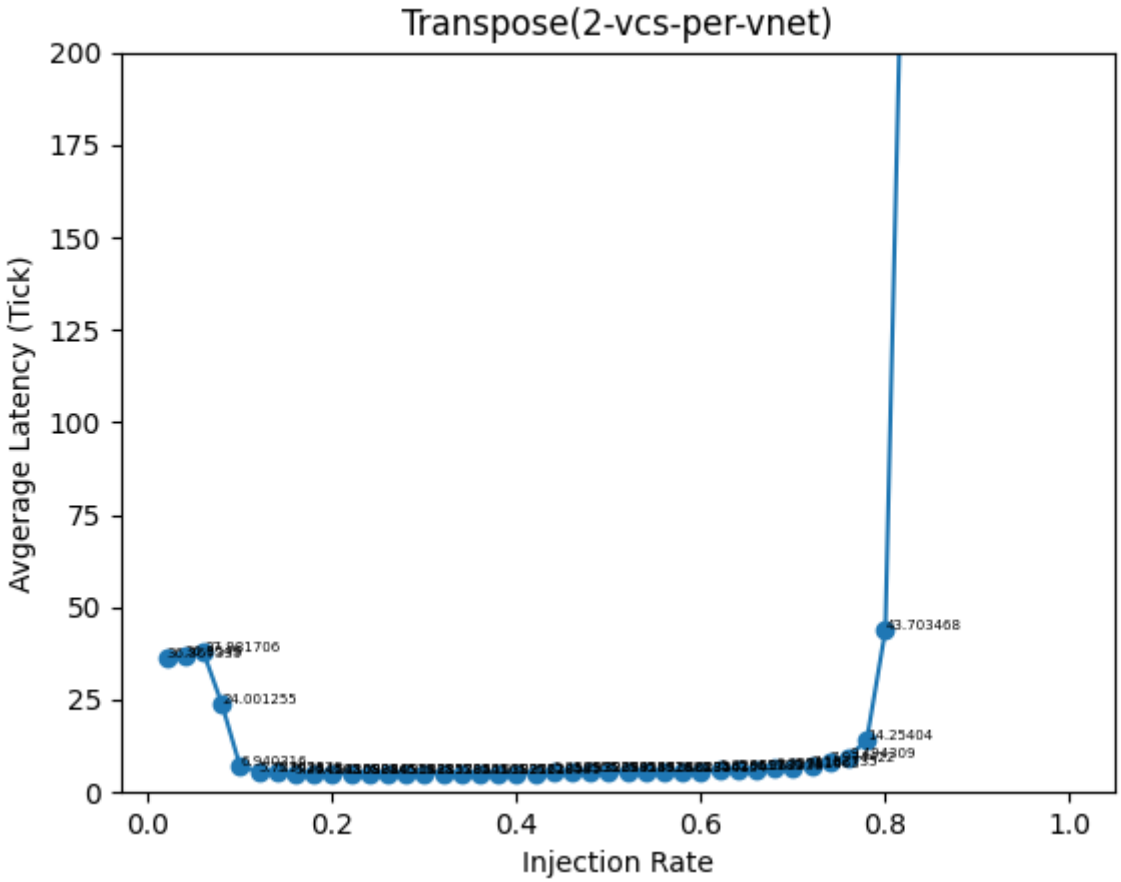

Neighbor(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

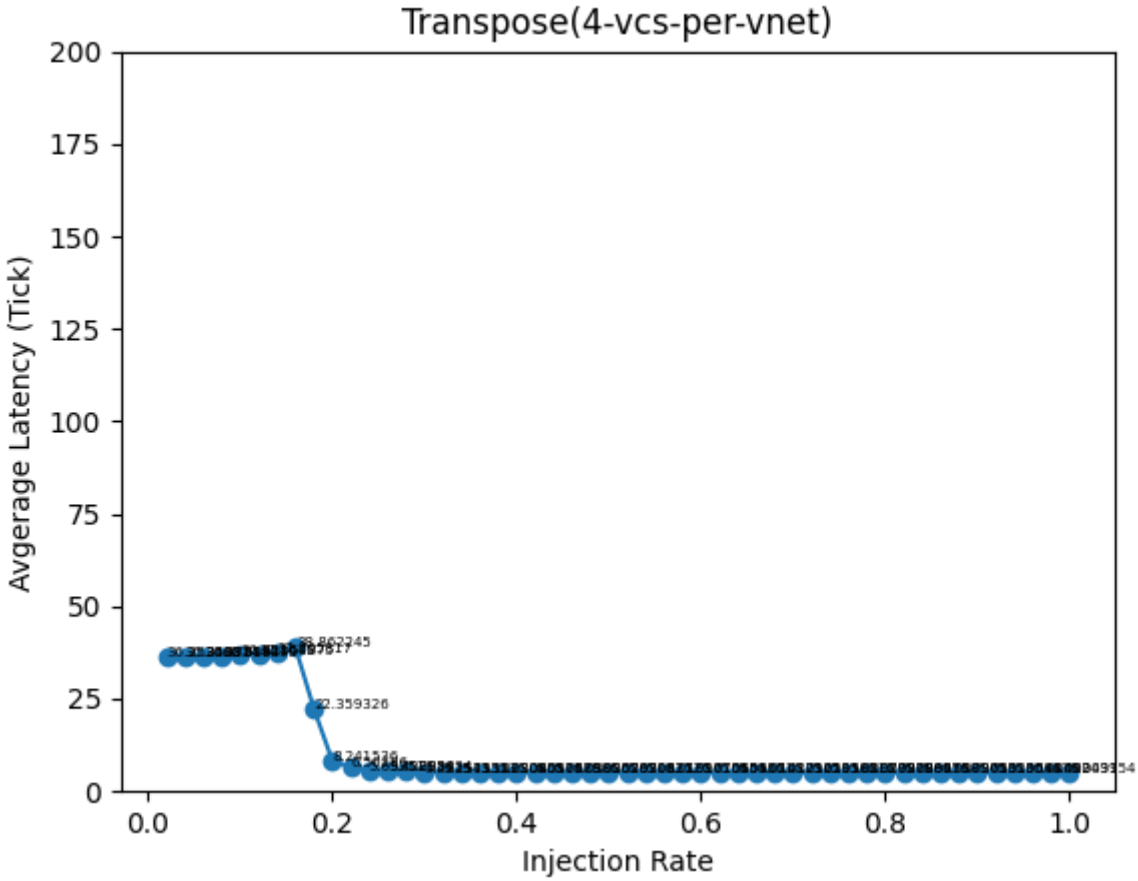

Shuffle(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

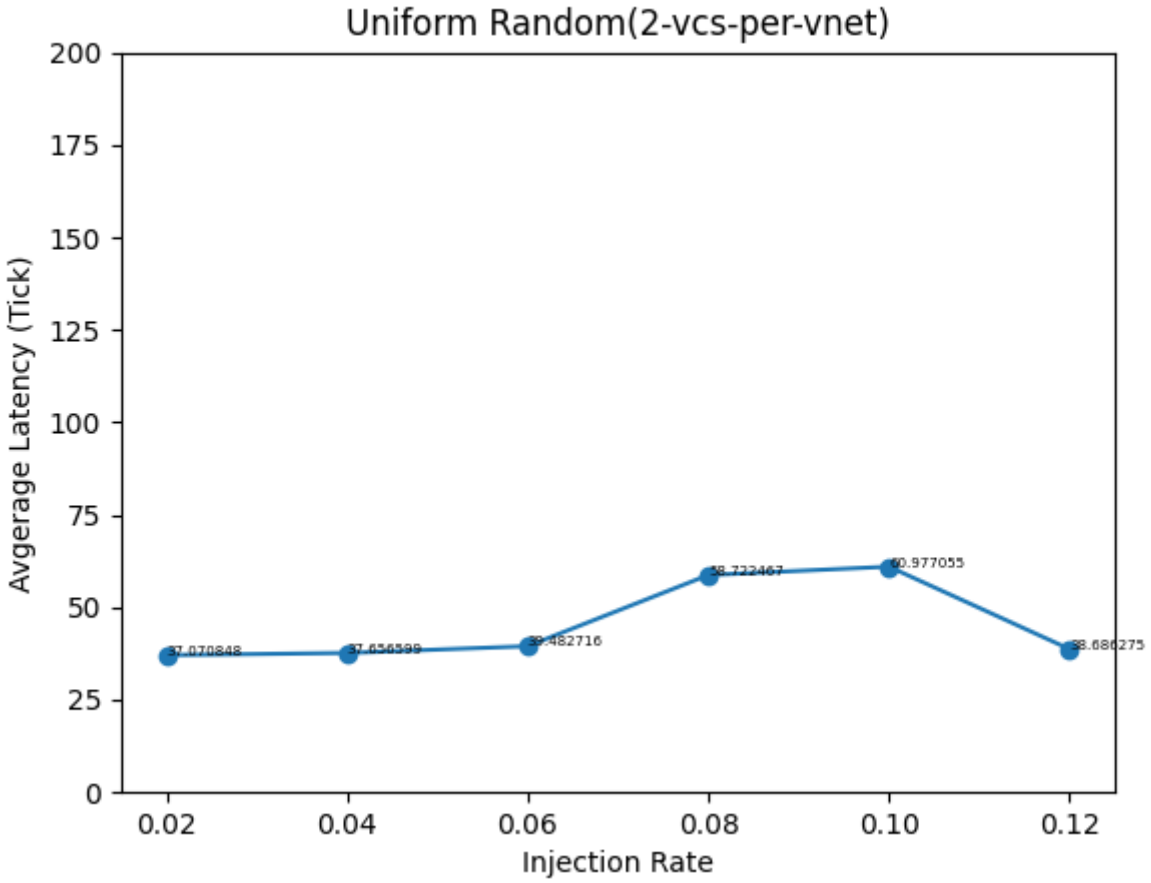

Shuffle(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

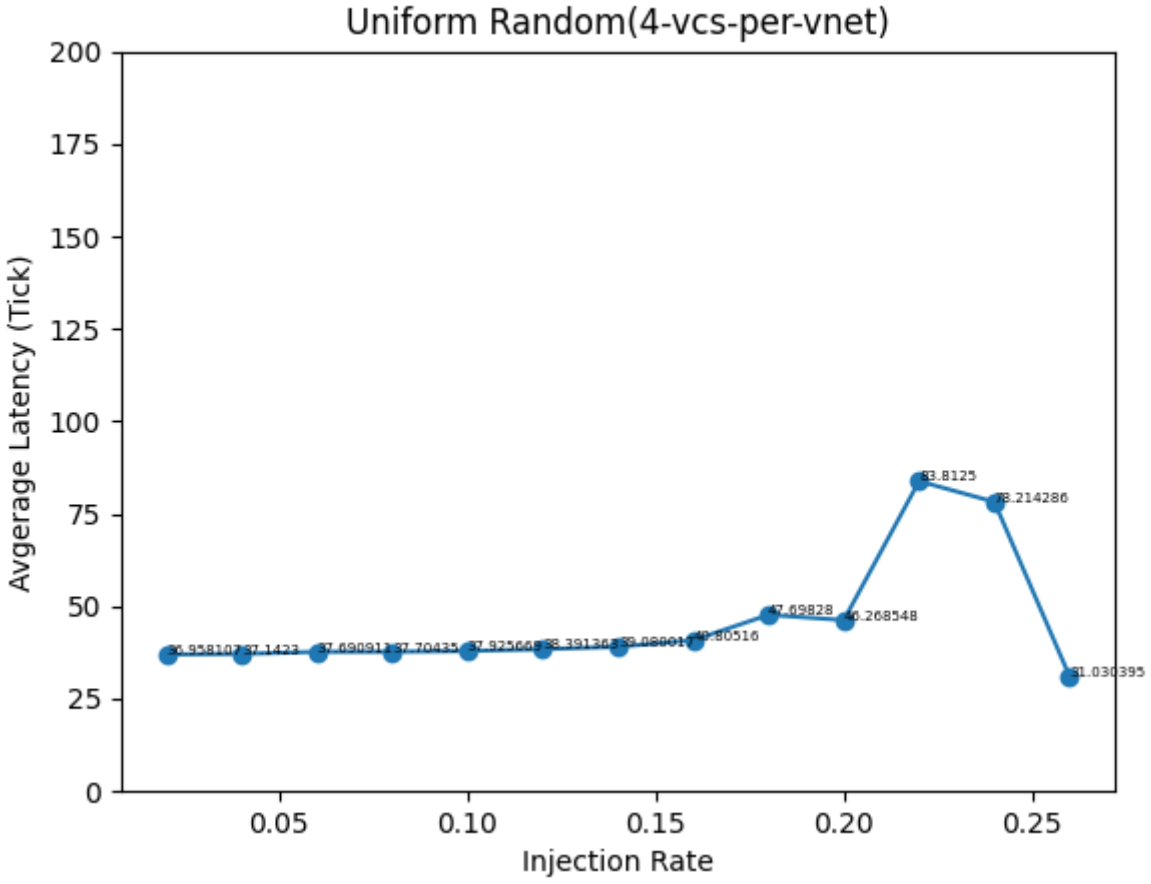

Tornado(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
64.555097
17.973918


Tornado(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
117.475015
18.998456

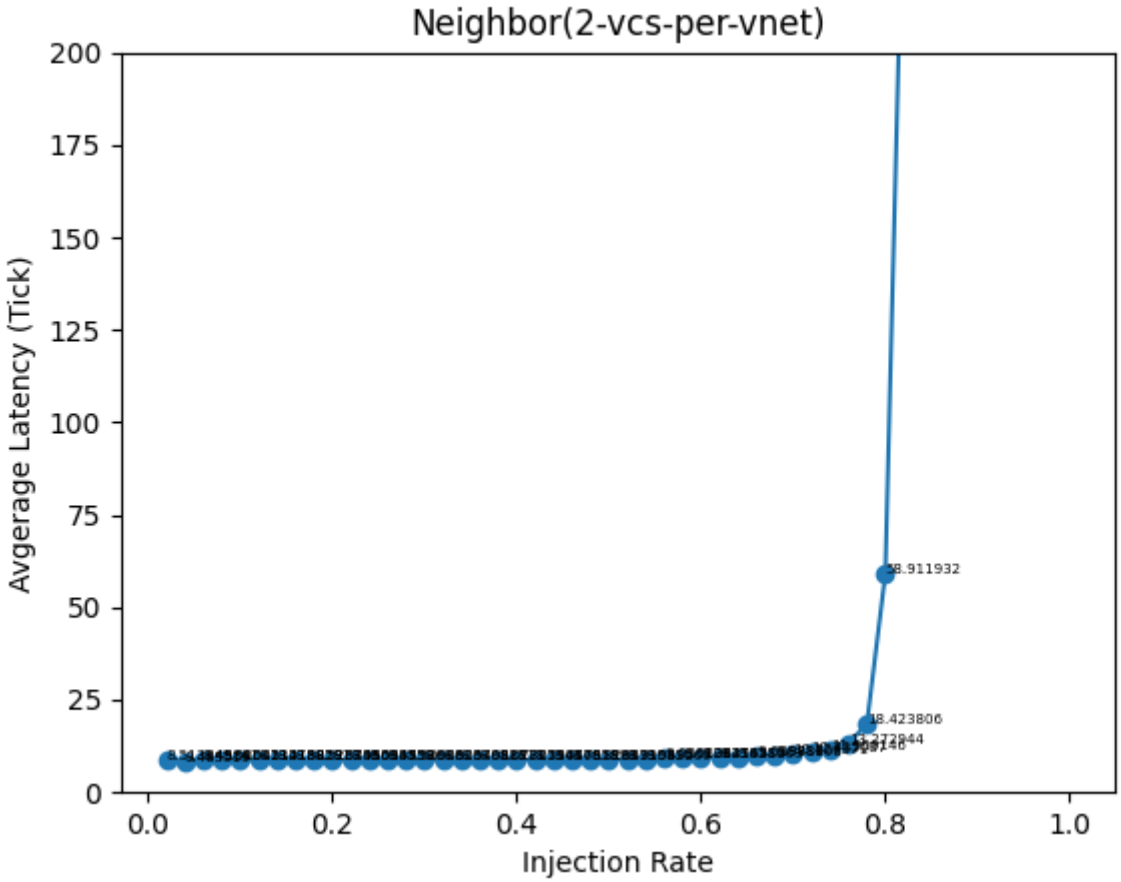

Transpose(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
43.703468
24.001255
14.25404

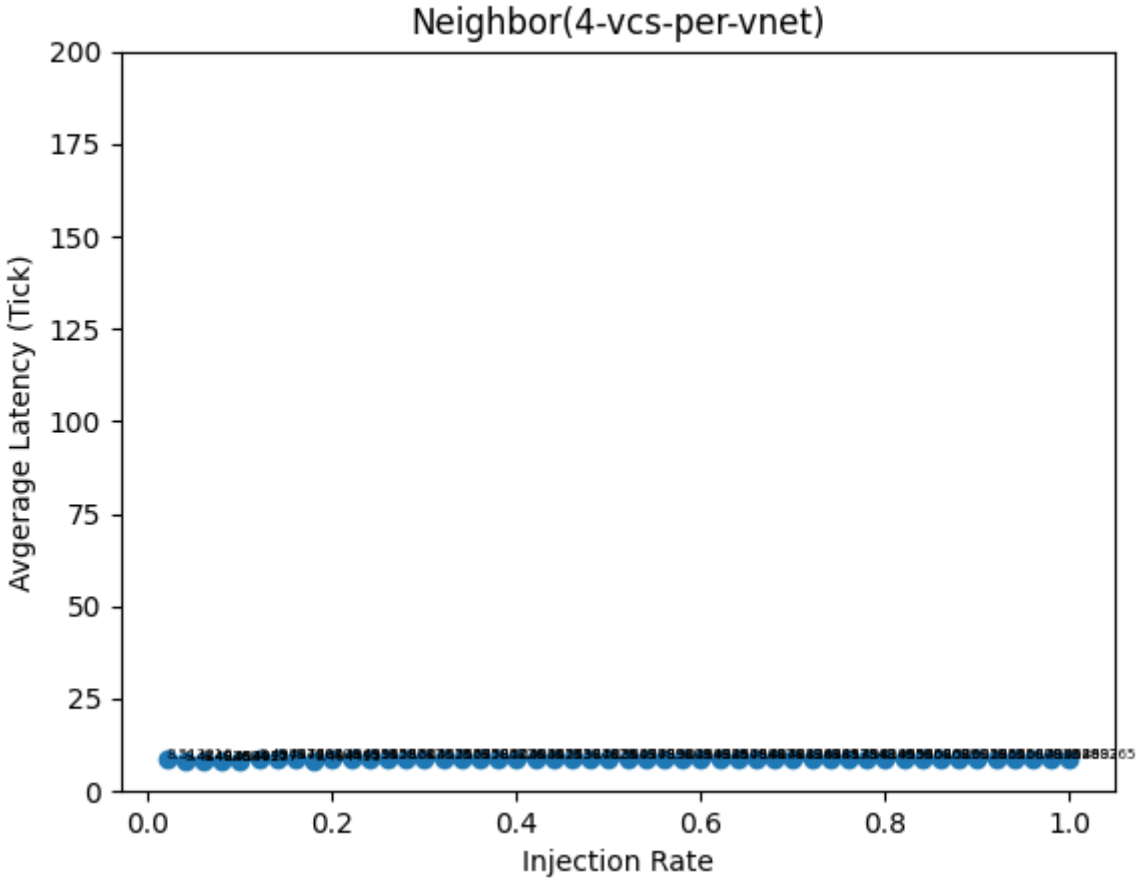

Transpose(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
22.359326

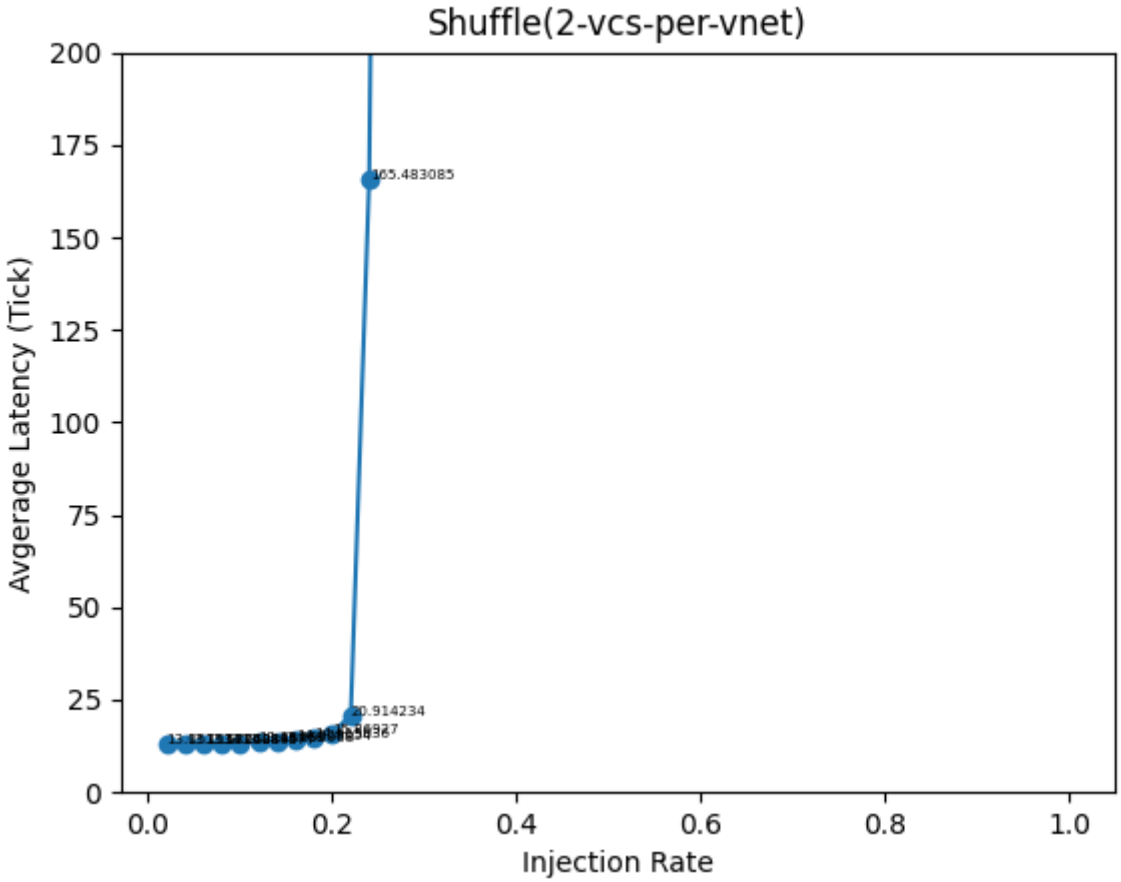


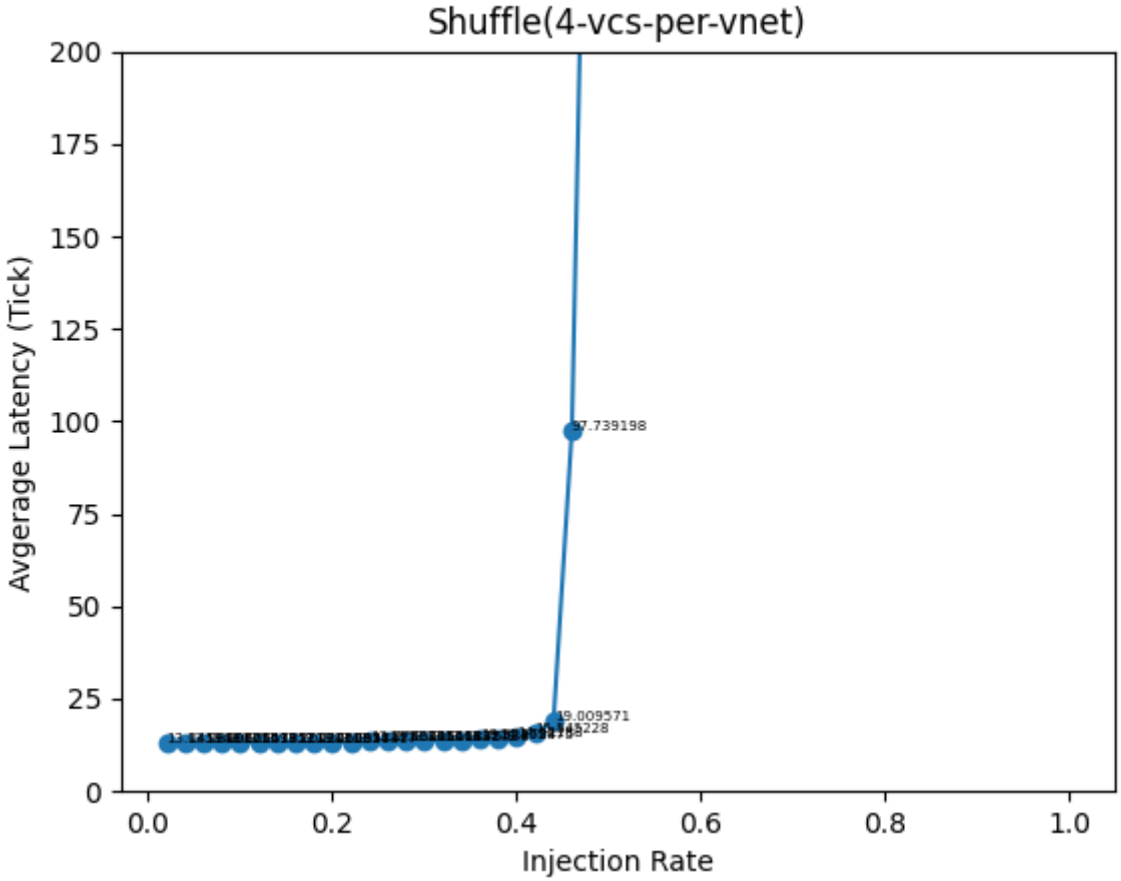


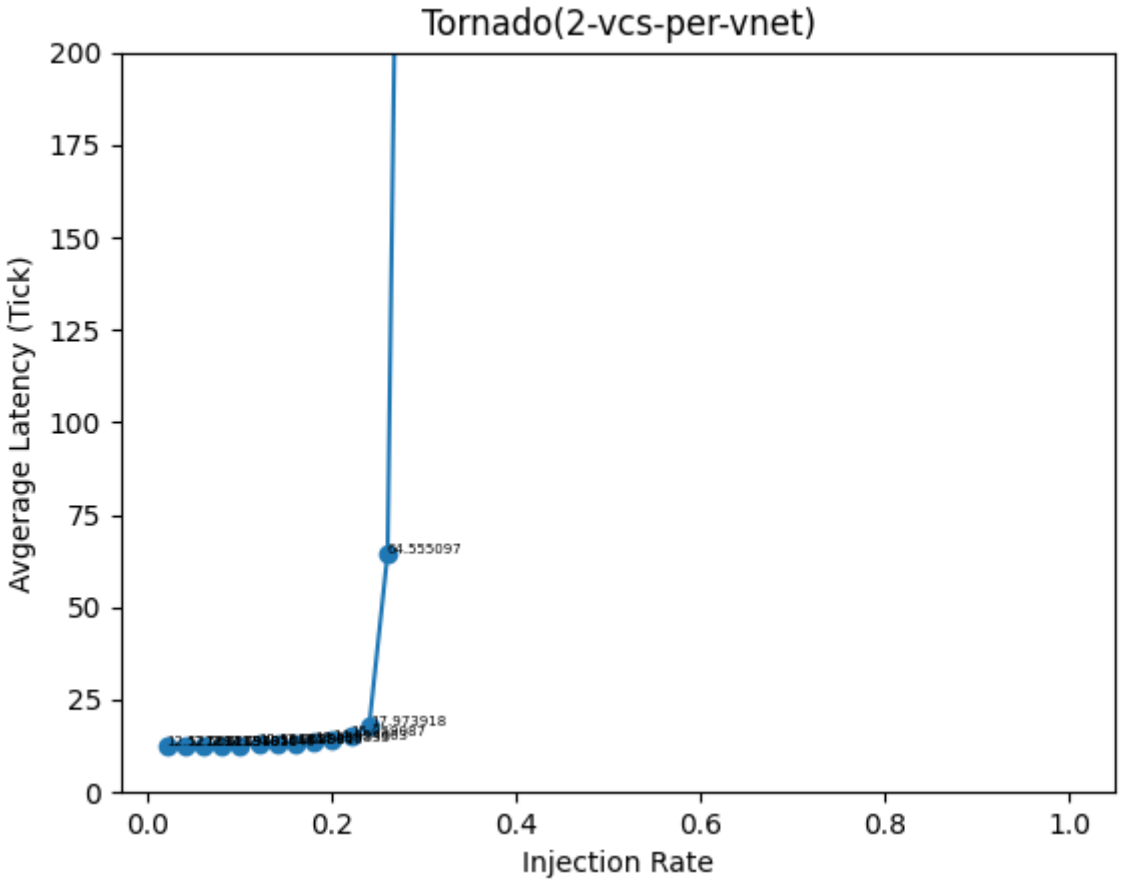
- 8×8-Mesh

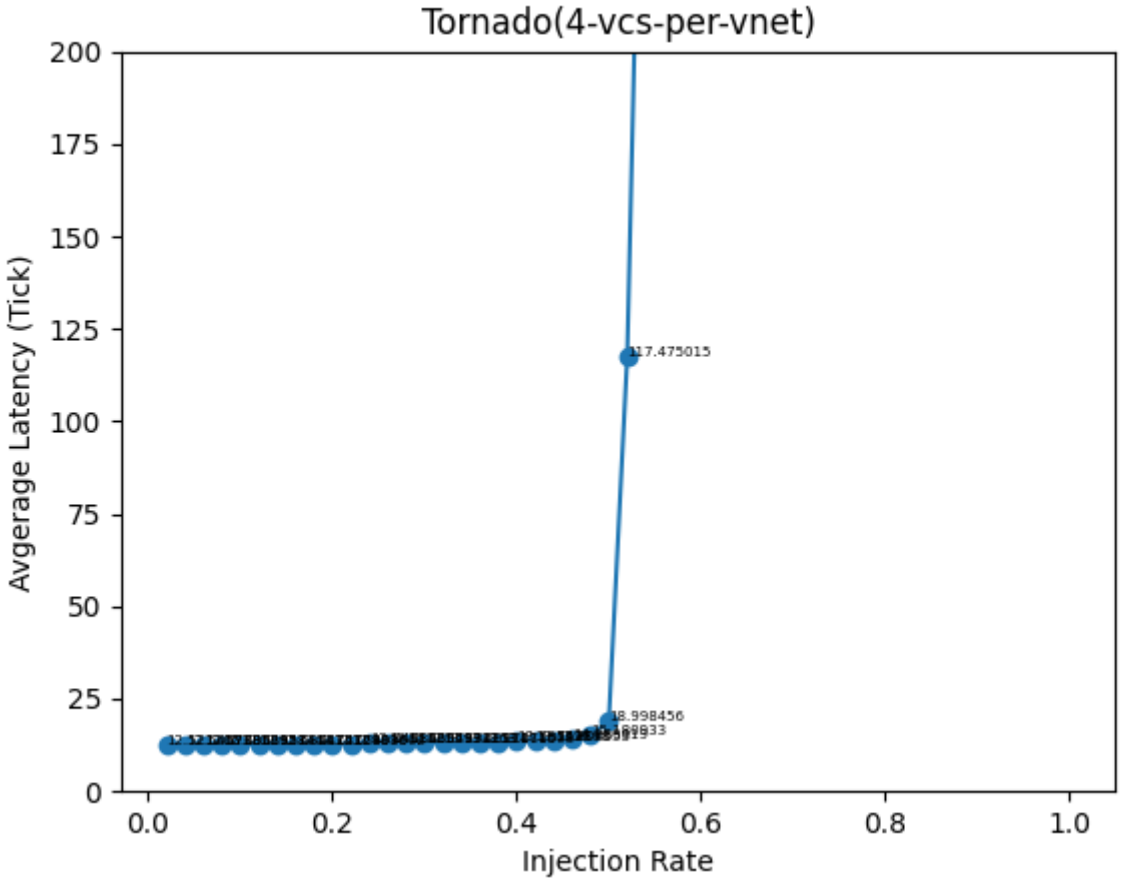

Neighbor(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
58.911932
18.423806

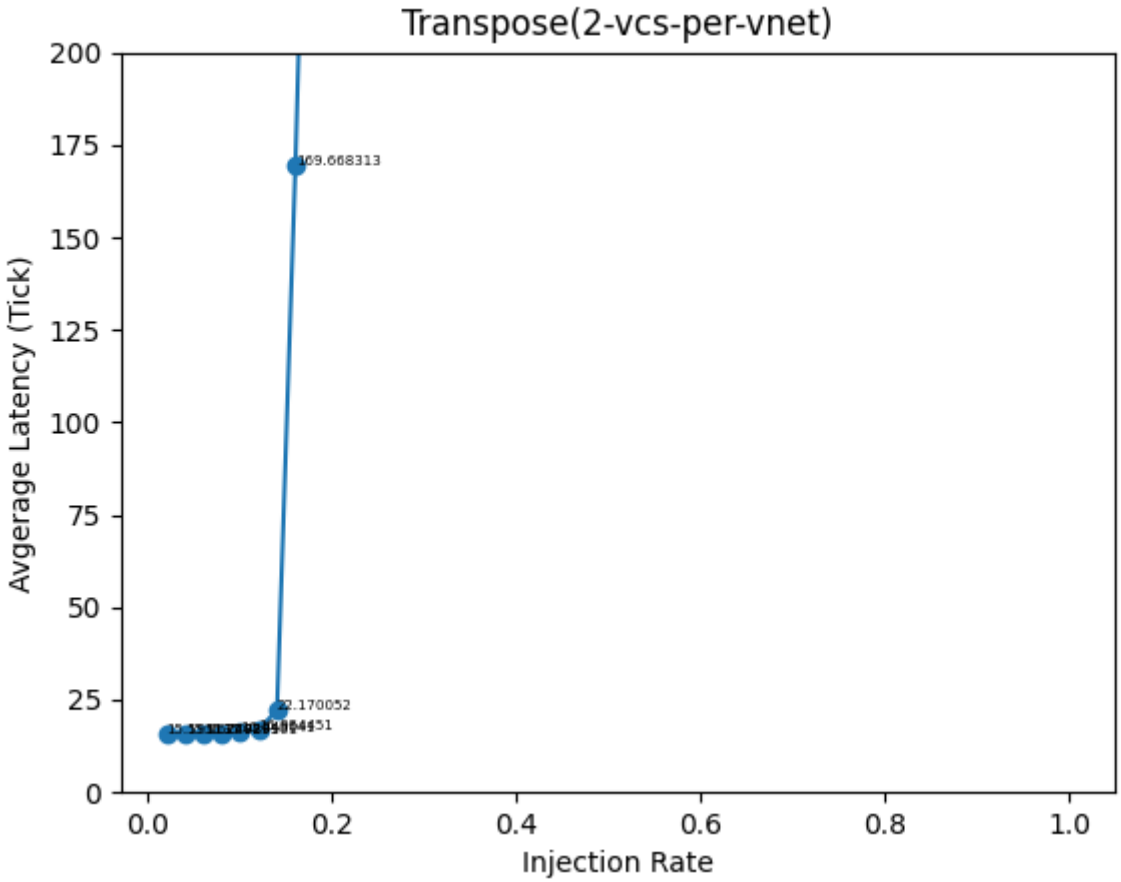

Neighbor(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

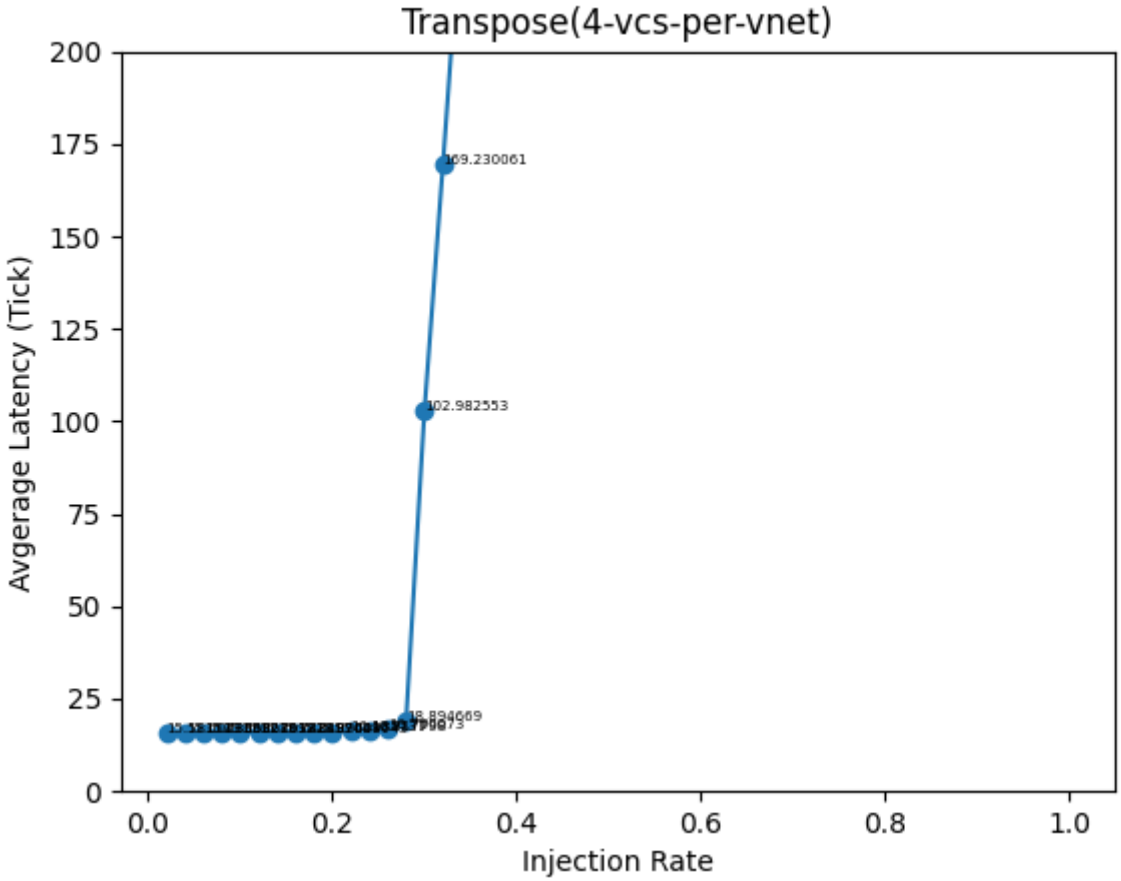

Shuffle(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
165.483085
20.914234

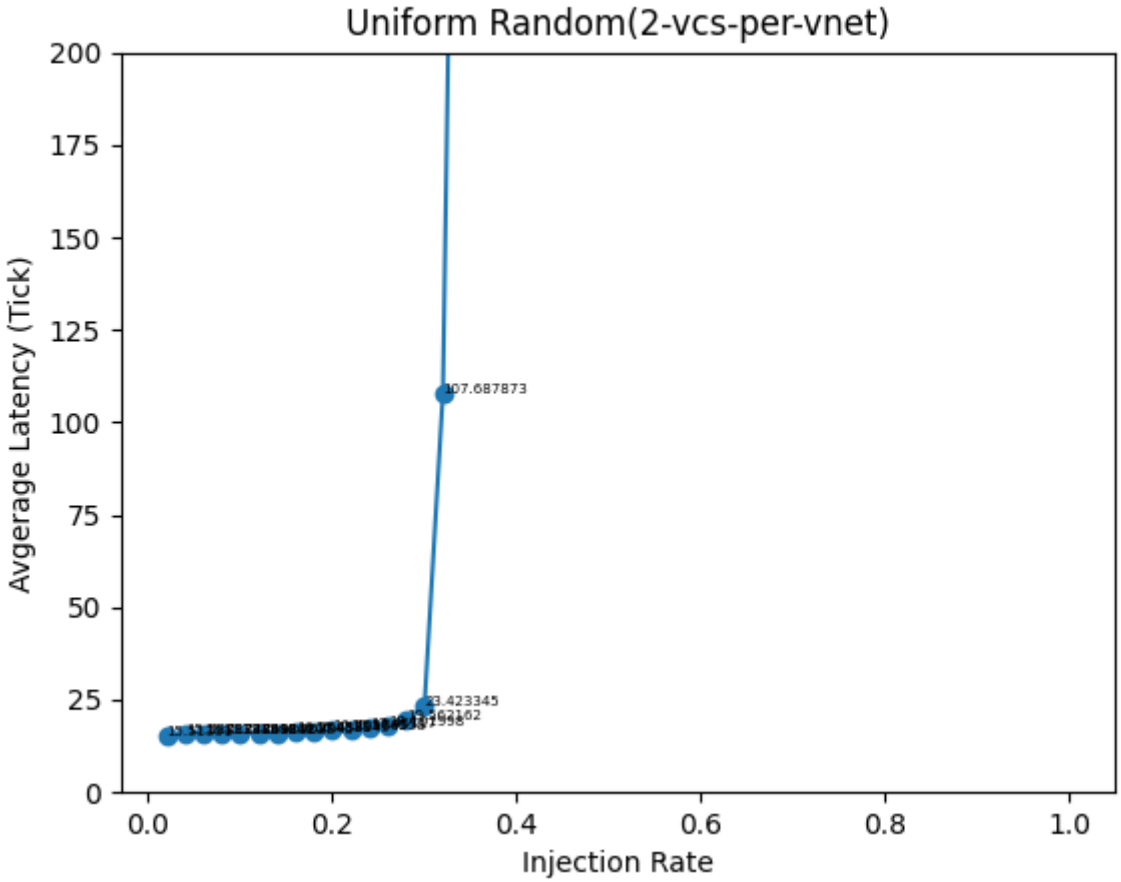

Shuffle(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
97.739198
19.009571

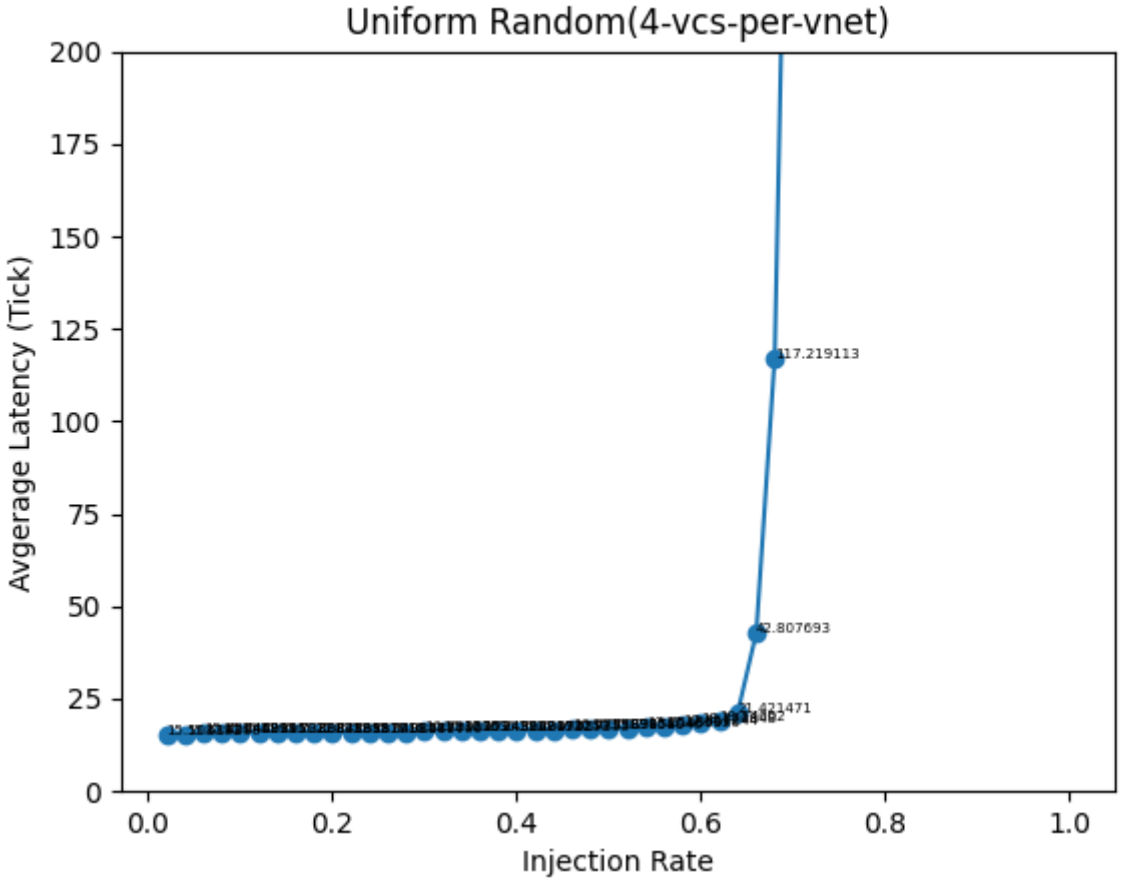

Tornado(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
200
175
150
125
100
75
50
25
0
0.0
0.2
0.4
0.6
0.8
1.0
64.555097
17.973918


Tornado(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
200
175
150
125
100
75
50
25
0
0.0
0.2
0.4
0.6
0.8
1.0
117.475015
18.998456

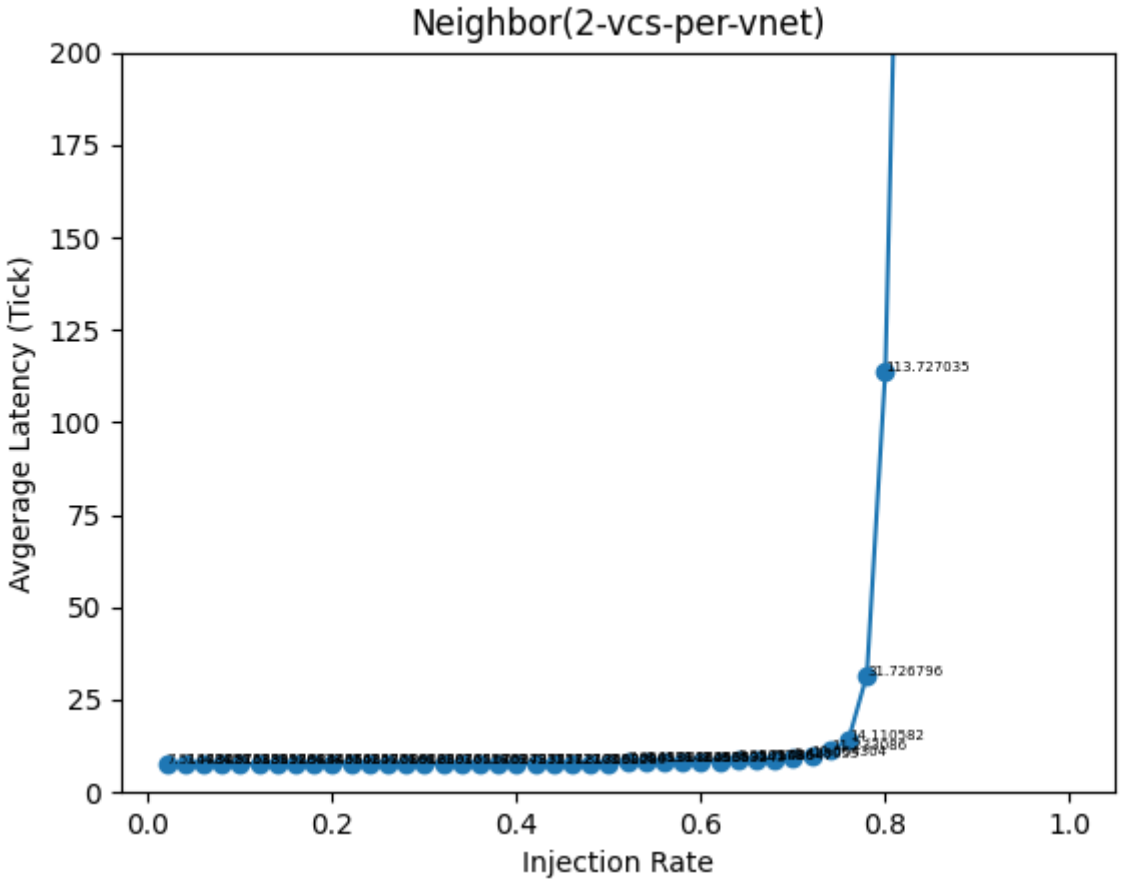

Transpose(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
200
175
150
125
100
75
50
25
0
0.0
0.2
0.4
0.6
0.8
1.0
169.668313
22.170052

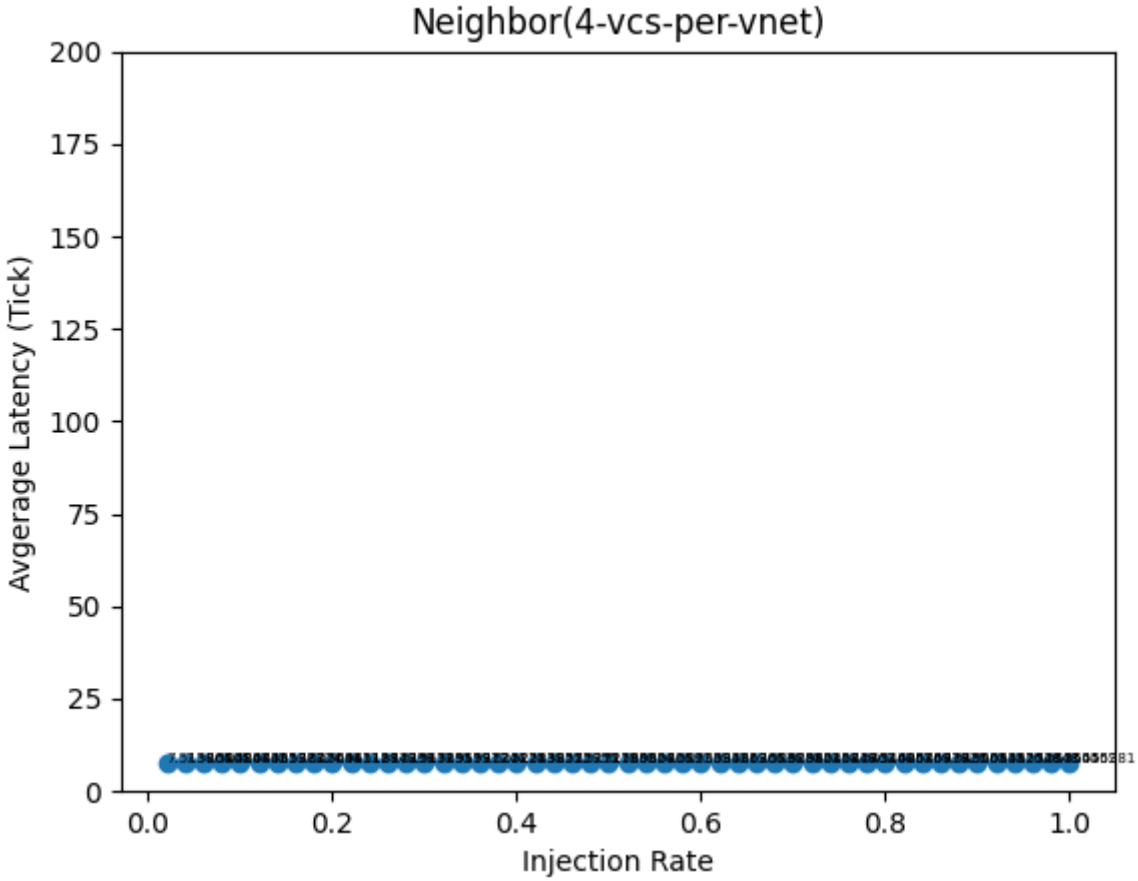

Transpose(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
200
175
150
125
100
75
50
25
0
0.0
0.2
0.4
0.6
0.8
1.0
169.230061
102.982553

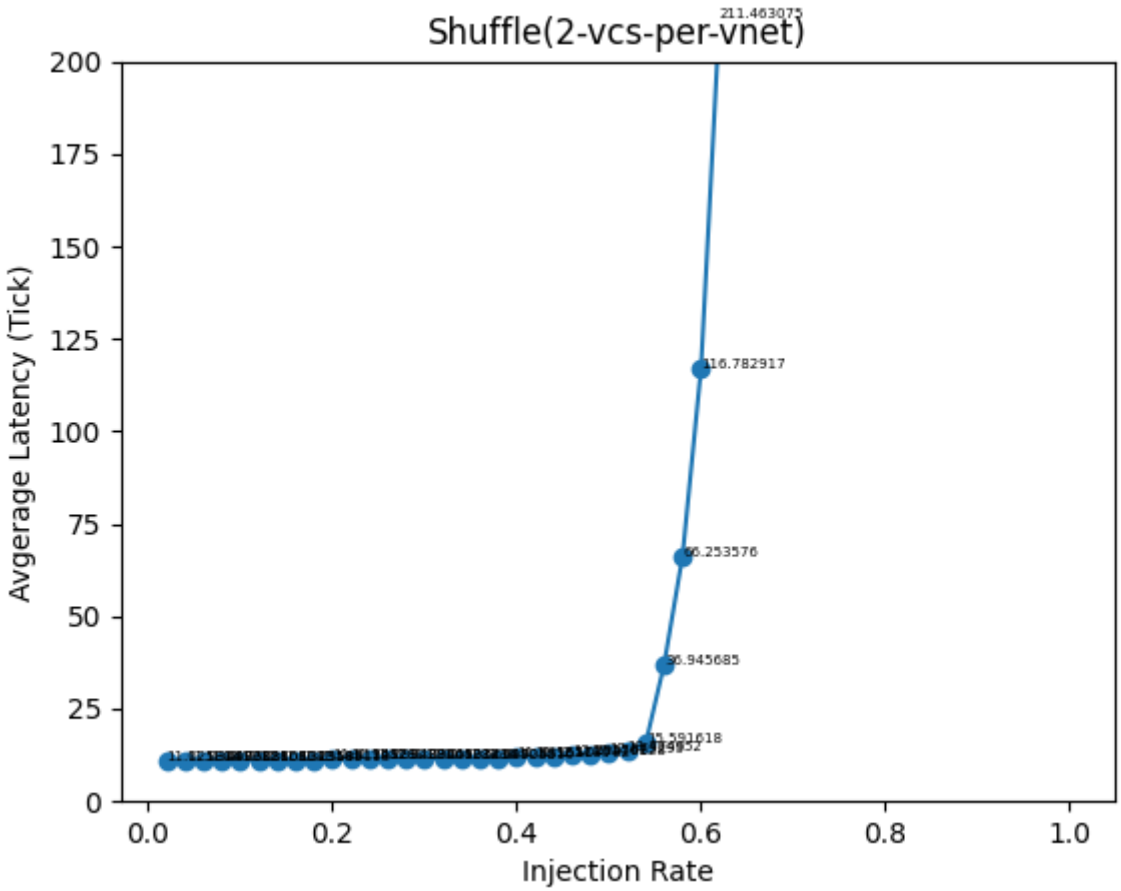


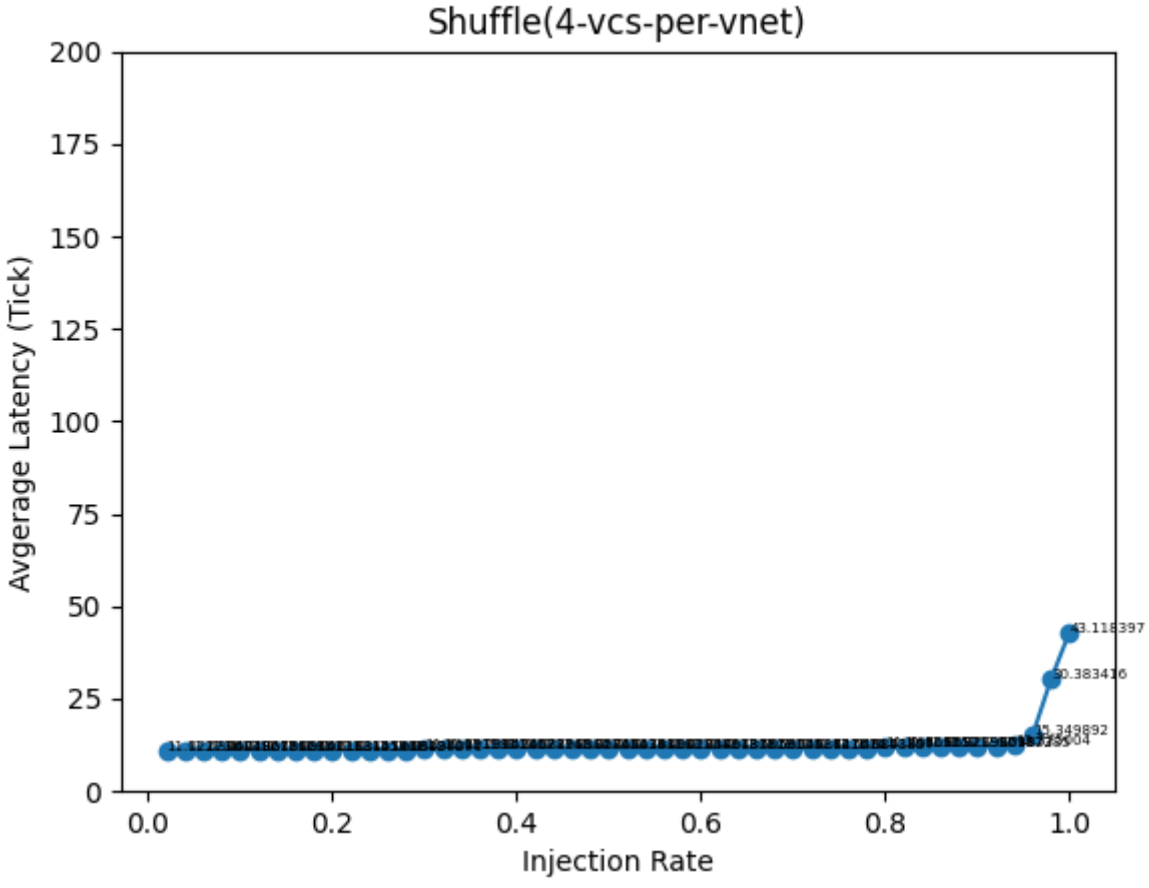


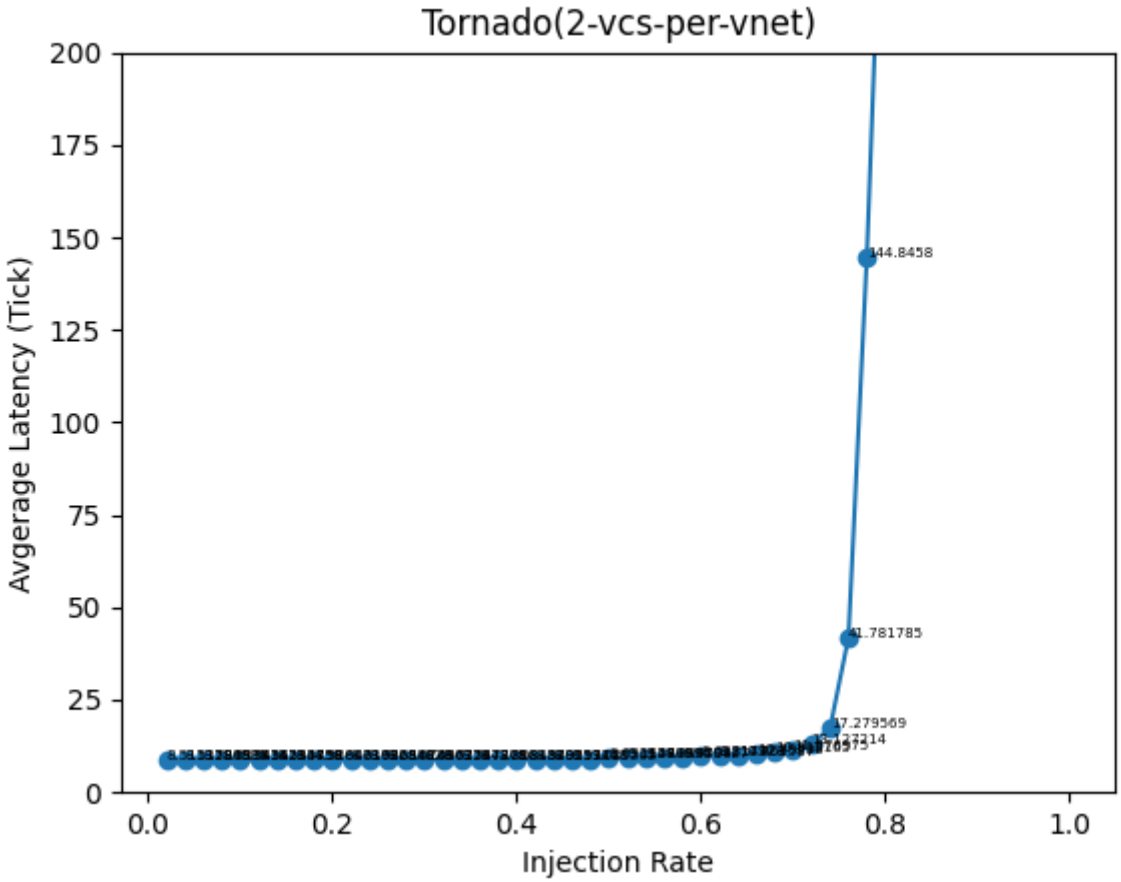
- $4 \times 4 \times 4$ -Torus

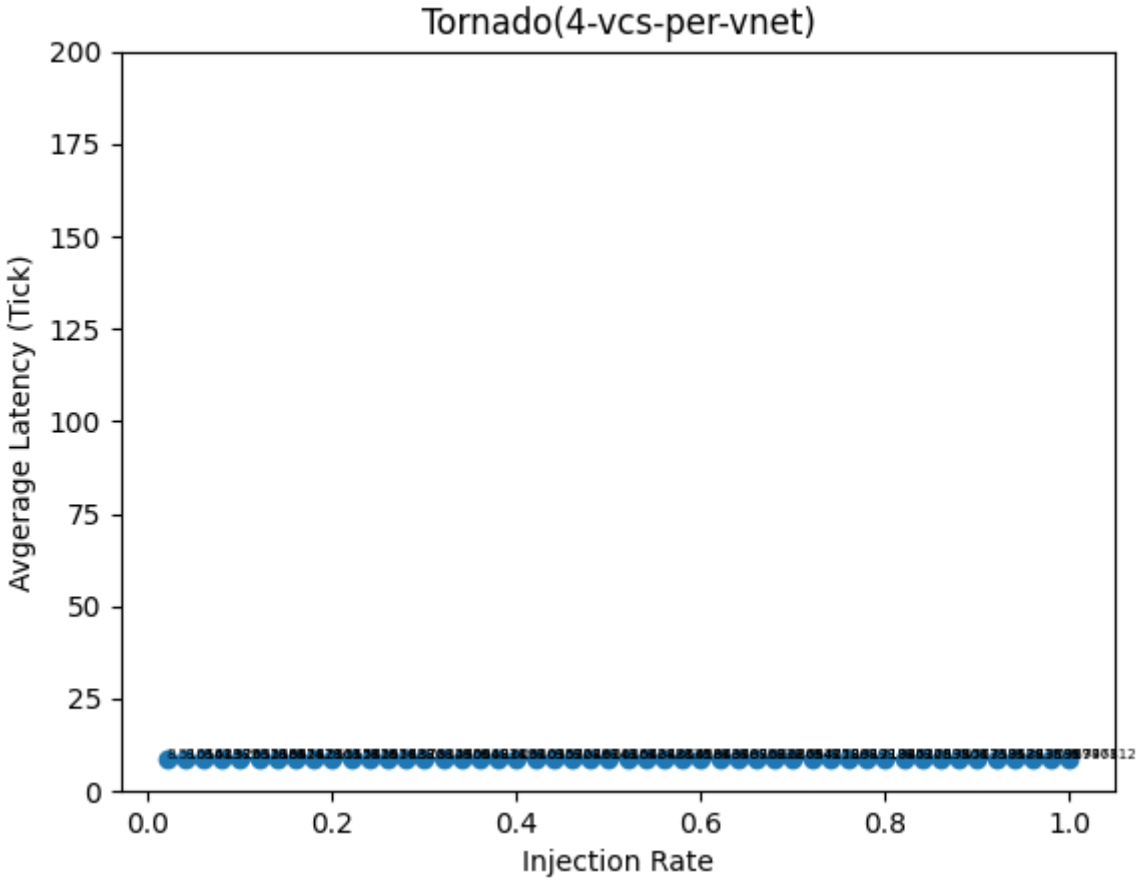

Neighbor(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
113.727035
31.726796

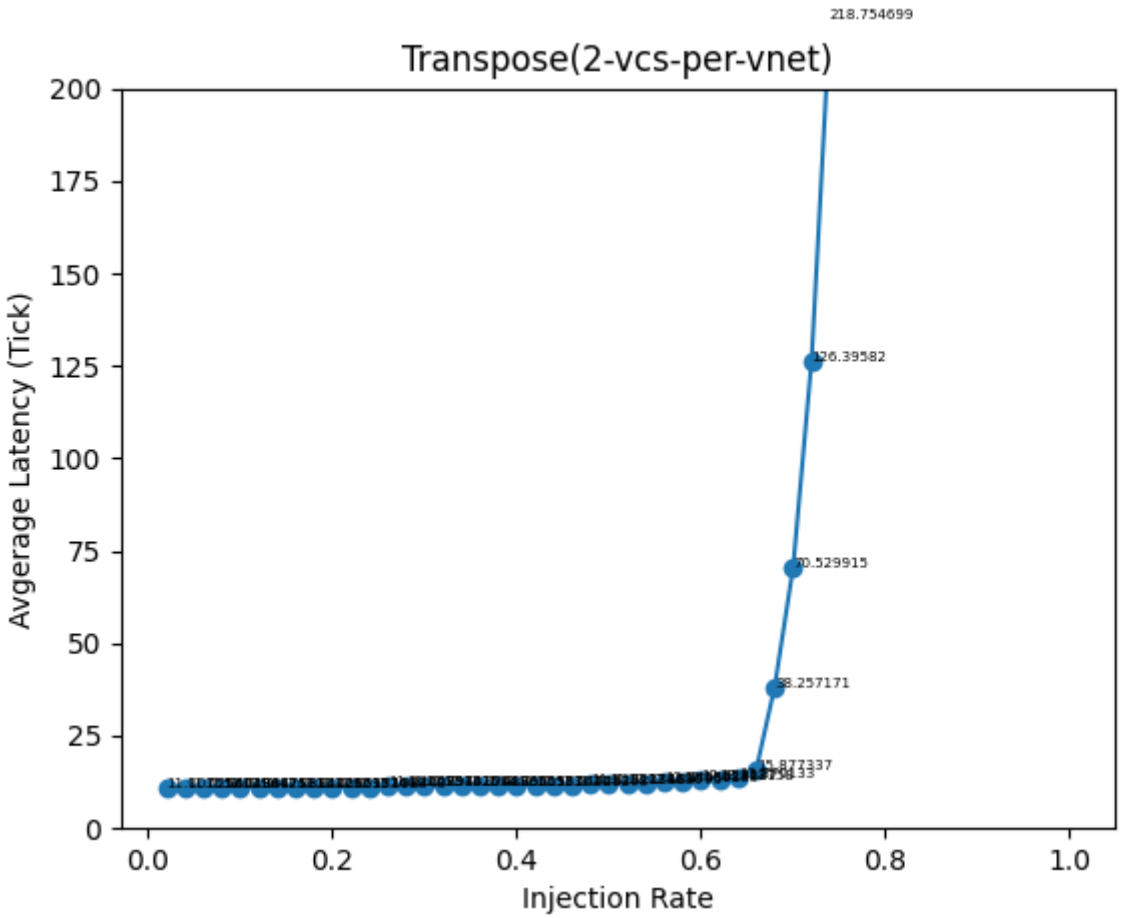

Neighbor(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

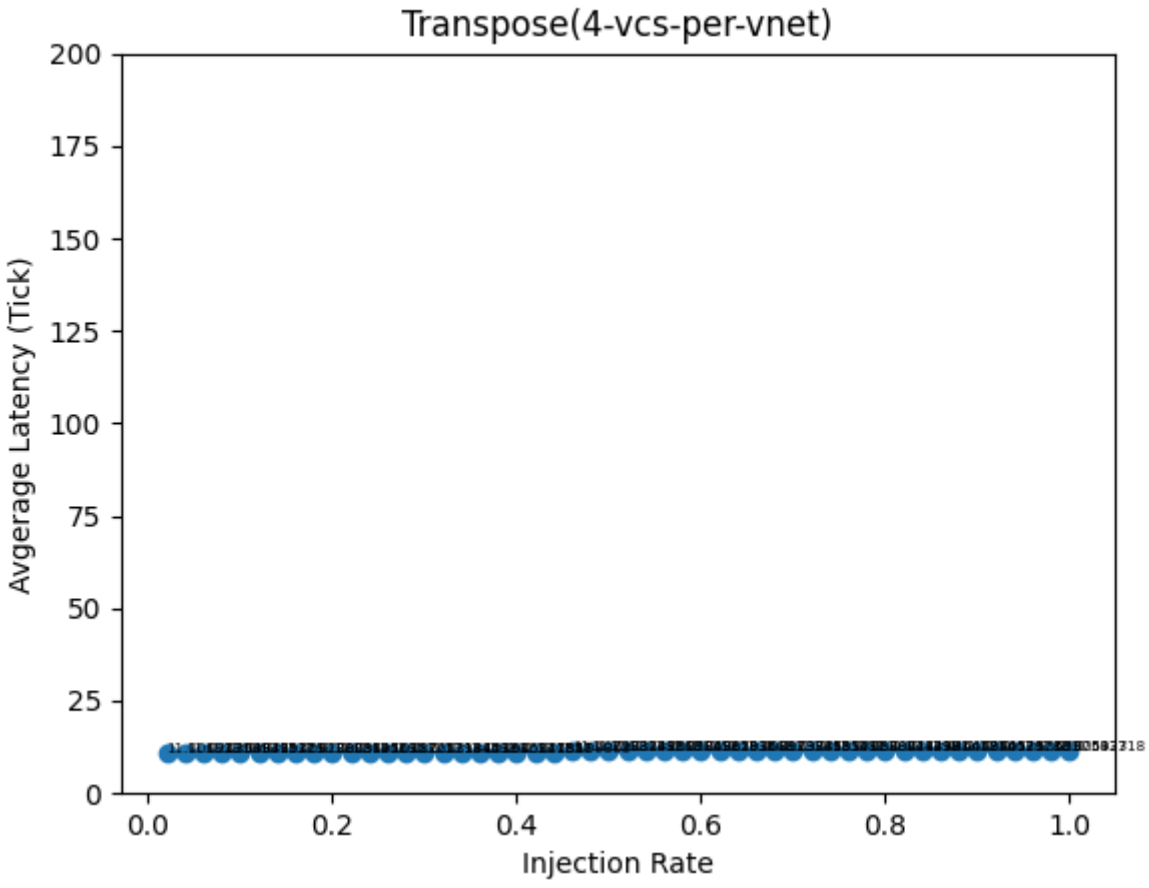

Shuffle(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
211.463075
116.782917
66.253576
36.945685

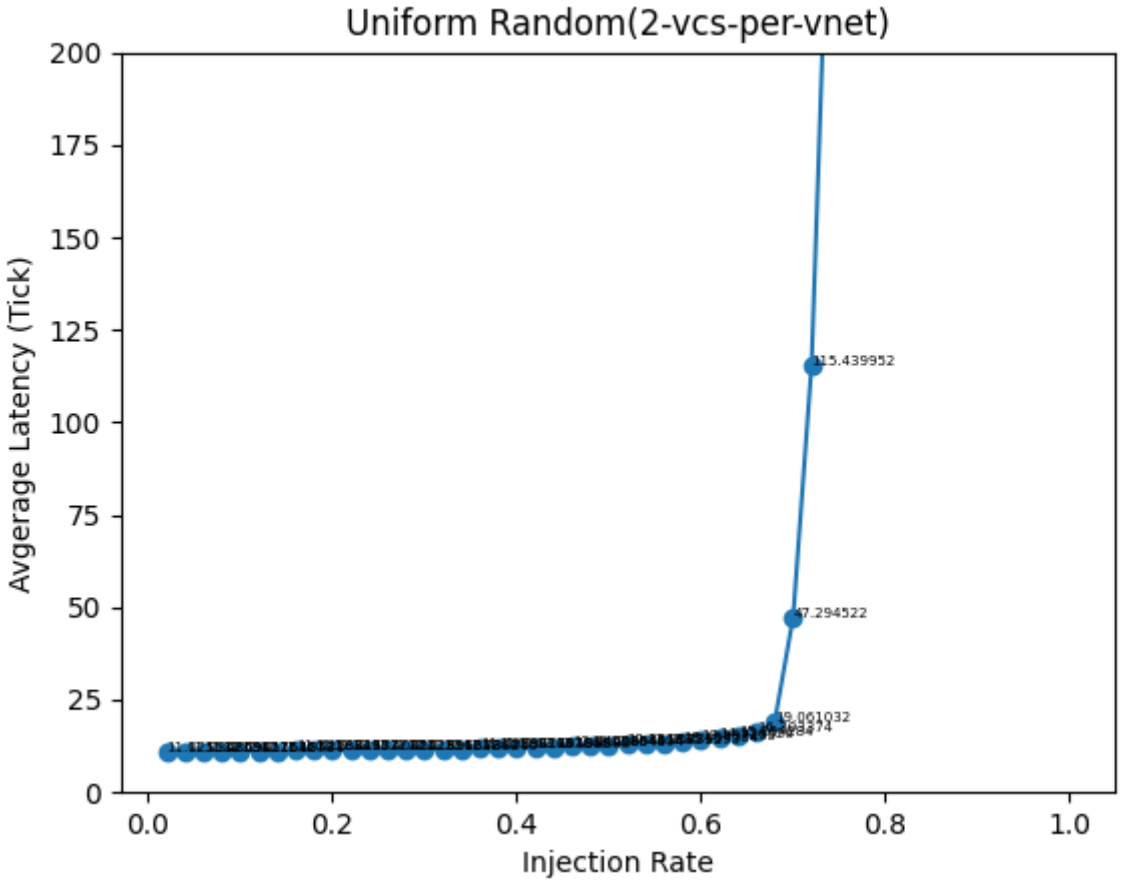

Shuffle(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
43.118397
30.383416

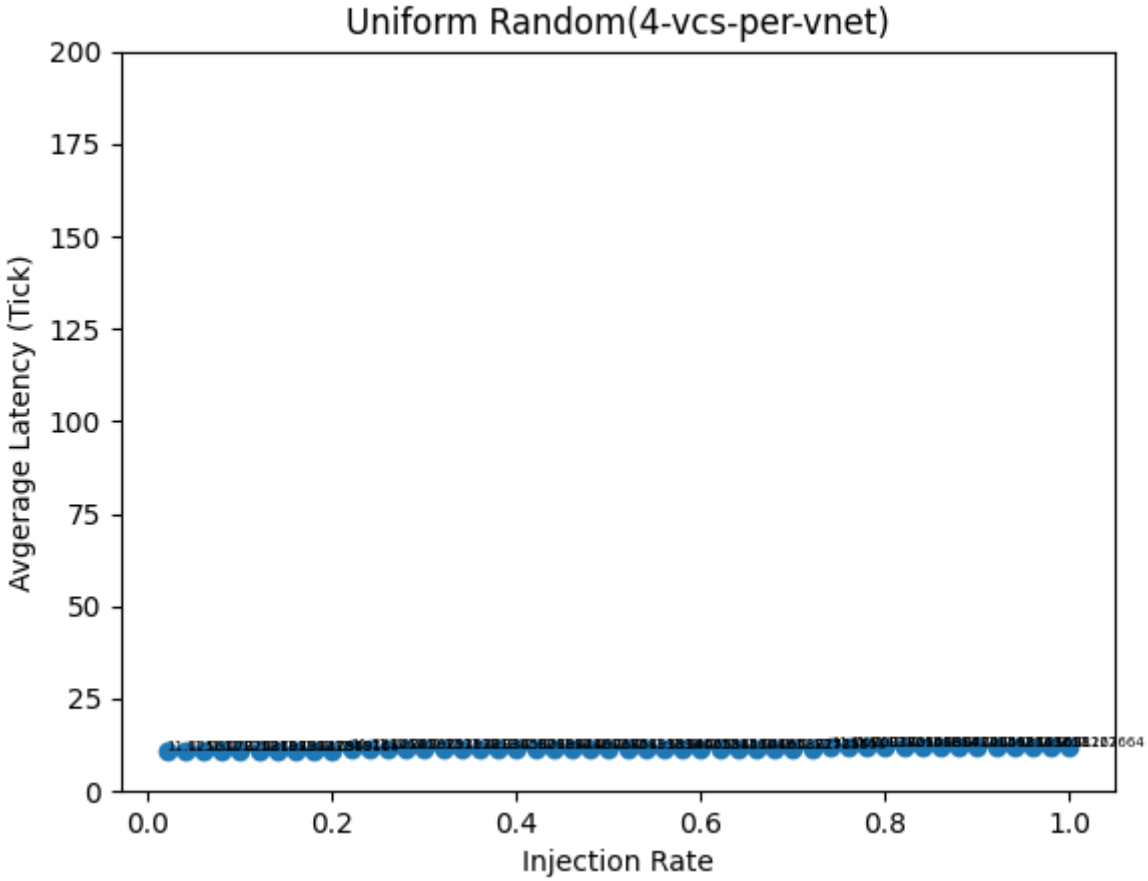

Tornado(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
200
175
150
125
100
75
50
25
0
0.0
0.2
0.4
0.6
0.8
1.0
144.8458
41.781785
17.279569


Tornado(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
200
175
150
125
100
75
50
25
0
0.0
0.2
0.4
0.6
0.8
1.0


Transpose(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
218.754699
126.39582
0.529915
38.257171
15.877337
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0


Transpose(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0


Uniform Random(2-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0
115.439952
47.294522
19.061032


Uniform Random(4-vcs-per-vnet)
Avgerage Latency (Tick)
Injection Rate
0
25
50
75
100
125
150
175
200
0.0
0.2
0.4
0.6
0.8
1.0

We simply introduce some information of Mesh_v3 in this lab. (since it's simple, we) In fact, we only add all the diagonals to the original Mesh. And make the algorithm to be walk along the diagonal first until they have the same x or the same y. We solve the deadlock problem by the virtual channels.