Adaptive Network on Chip Routing using the Turn Model

BY

Jonathan W. Brown

B.S., University of New Hampshire (2011)

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

May, 2013

This thesis has been examined and approved.

Thesis director, Michel Charpentier
Associate Professor of Computer Science

Radim Bartos
Associate Professor of Computer Science

Qiaoyan Yu
Assistant Professor of Electrical and Computer Engineering

Phil Hatcher
Professor of Computer Science

Date

ACKNOWLEDGMENTS

It is with immense gratitude that I acknowledge the support and help of my thesis advisers, Professor Michel Charpentier, who always found time to provide constructive feedback to my thoughts and who was patient with my writing style, Professor Radim Bartos, who taught me how to explain my thoughts and present them clearly in writing, Professor Qiaoyan Yu, who answered my detailed oriented questions and helped me progress, and Professor Phil Hatcher who never let any detail slip pass me.

I would also like to thank all my friends in W236 who helped me on the days where nothing seemed to be working and gave me the push to approach the problem from a different angle and the Computer Science Department that let me experience the joy of programming.

Lastly I would like to thank my parents who always gave me encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Adaptive Network on Chip Routing using the Turn Model

by

Jonathan W. Brown

University of New Hampshire, May, 2013

To create a viable network on chip, many technical challenges need to be solved. One of the aspects of solutions is the routing algorithm: how to route packets from one component (e.g., core CPU) to another without deadlock or livelock while avoiding congestion or faulty routers. Routing algorithms must deal with these problems while remaining simple enough to keep the hardware cost low.

We have created a simple to implement, deadlock free, and livelock free routing algorithm that addresses these challenges. This routing algorithm, Weighted Non-Minimal OddEven (WeNMOE), gathers information on the state of the network (congestion/faults) from surrounding routers. The algorithm then uses this information to estimate a routing cost and routes down the path with the lowest estimated cost.

A simulator was developed and used to study the performance and to compare the new routing algorithm against other state of the art routing algorithms. This simulator emulates bit reverse, complement, transpose, hotspots, and uniform random traffic patterns and measures the average latency of delivered packets.

The results of the simulations showed that WeNMOE outperformed most routing algorithms. The only exception was the XY routing algorithm on uniform random and complement traffic. In these traffic patterns, the traffic load is uniformly distributed, limiting the opportunity for an improved route selection by WeNMOE.

CHAPTER 1

Network on Chip Description and Model

1.1 History

A network on chip (NoC) uses a regular layout of routers connected by wires. This concept has existed for many years and has been implemented successfully [1]. The most common layout of the routers is a 2D mesh (a grid) where each router has 4 neighbors. Because of this regular layout, many assumptions that do not apply to TCP/IP networks can be applied to NoC. These assumptions simplify routing. For example, shortest path calculation is trivial on a 2D mesh.

In addition, a regular layout allows a router and its neighbors to be closer. This closeness keeps the connecting wires short and is important because when transferring data through the network, wire delays now account for most of the latency. This is because the speed of the CPUs has increased, but the speed of the wires has not [2, 3]. Therefore, the closer the routers, the faster the chip.

A router's memory footprint also directly affects the length of the wires. The larger the footprint, the longer the wires are to connect routers [4]. Consequently, several NoC routing algorithms do not have routing tables or have small routing tables to keep the memory footprint small.

Even with a regular topology, NoC still requires a routing algorithm, mainly to avoid deadlock and livelock. Most of the routing algorithms are based on a 2D mesh NoC because routing algorithms can be designed easily for this layout. Intel is one of the manufacturers of a 2D mesh NoC with their Single-Chip Cloud Computer and Intel Teraflops Research Chip [5, 6, 7]. The Teraflops Research Chip implements a 2D mesh network with 80 cores

and the their cloud computer contains technology to have upwards of 100 cores on a single chip.

1.2 Network on Chip

The network on chip is a regular layout of routers connected by wires. The rest of this section describes the model of a NoC used.

The layout of the routers is a 2D mesh where each router has four neighbors (north, east, south, and west) is shown in Figure 1-1. The wires connecting the routers are unidirectional, so a connection between neighboring routers uses two wires.

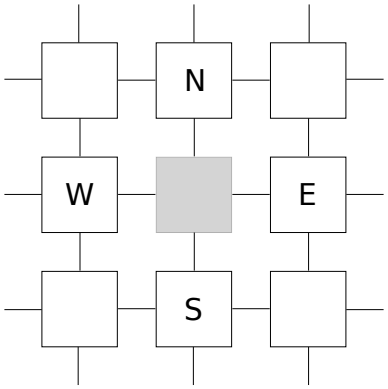


Figure 1-1: Neighbors of a router

1.2.1 NoC Router

The model of a router is comprised of five input queues, five output queues, and a component. The component of a router is a hardware device that uses/needs access to the network (e.g., core or memory). The queues in a router are shown in Figure 1-2. Each direction (neighbor) has a dedicated input and output queue to service the wires. The wires transfer data from the output port of a router to an input port of a neighboring router. The last input and output queues are the connections to the component the router services. The input queues in a router are usually small and the output queues have a size of one. The

component is usually a processing core.

A router also contains logic to decide where to route data in the network using information from surrounding routers; at the core of this is a routing algorithm. A routing algorithm is considered *local* if routing decisions are based on local information (e.g., the current node or the current node and some surrounding nodes). An algorithm is considered *optimal* or *minimal* if the route chosen for a packet always uses a shortest path from source to destination. An algorithm is considered *non-minimal* if there is at least one route used that is longer than the shortest path. An algorithm is *adaptive* if packets with the same source and destinations can traverse different minimal and/or non-minimal paths. Note that grid topologies usually contain multiple shortest paths for a given source and destination pair.

The component itself also has a queue. This is the queue of generated packets that have not been injected into the network. Generally, the component will function at a faster clock speed than the network.

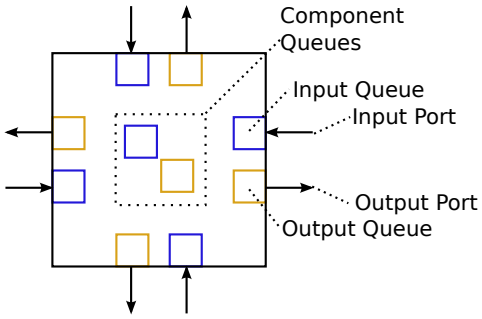


Figure 1-2: Model of a NoC router

1.2.2 Packet

The information/data that travels through the network is divided into packets. Each packet is made of flow control digits (flits). The queues in a router store flits and the wires transfer flits.

The first flit in a packet is called the head flit. This flit contains information for routing

the packet to its destination. Usually this is just the destination address, but depending on the NoC, the head flit could contain more information that is used during routing. The next several flits in the packet are data flits, they contain the actual data that is to be transmitted over the network. The last flit of the packet is called the tail flit. This flit signals the routers that the end of the packet has been reached. Figure 1-3 show the structure of a packet.

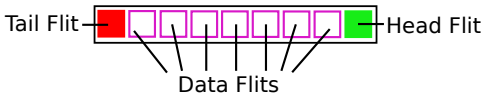


Figure 1-3: Structure of packet

1.3 NoC Operation

The operation of a network is split into cycles, and during each cycle the same events occur. The cycle is constructed in such a way to allow a flit that is not blocked to move across one router each cycle. This allows quick transfer of an entire packet if there is no contention for ports.

There are three major events that occur each cycle: packet generation, routing, and wire transfer. At the start of each cycle, the routers check for any new packets that a component may have generated. If there is a new packet in the component's queue, the first several flits are moved to the router's input queue.

After the check for newly generated packets, a router will use the routing algorithm and route the packets. First an arbiter decides on an order (a priority) to route the flits residing in the input ports of the router. Next the routing algorithm looks at the first flit in the queues (in the order specified) and if the flit is a head flit, routes the flit.

Routing a flit consists of determining the correct output port to send the packet using the routing algorithm. If the routing algorithm uses network state to help determine an output port, the state of the network at the cycle the packet arrived at the router is used. Once an output port has been chosen, if the output port is not being used, an internal

connection is made from the input port the head flit resides in to the specified output port. If the output port already has an internal connection to another input port, the decision is saved and the head flit waits until the next cycle. This decision also creates a dependency: the input port waits for the output port to change state.

Determining the correct output port may take several cycles if the routing algorithm cannot process head flits as fast as the network operates. This means that if the first flit in an input queue is a head flit, it may stay as the first flit in the queue for multiple cycles while the routing algorithm determines the correct output port. During this time, other flits are processed inside the router.

Once all the internal connections have been made, for each internal connection, the router moves the first flit in the input queue to the connected output queue. If the flit moved is a tail flit, the internal connection is removed so the output port can be used by a different packet next cycle.

The wire transfer moves flits from the output queue of one router to the input queue of a neighboring router. All the wires transfer at the same time. If the input queue is full or the output port is empty, the wire does not transfer a flit. The wire transfer may also transfer state information of the routers to the router's neighbors.

A packet is marked to exit the network when the packet reaches its destination. This occurs when the routing algorithm determines the correct output port is the output port connected to the component. The connection functions like other internal connections. However, the component transfers the packet from the router instead of a wire.

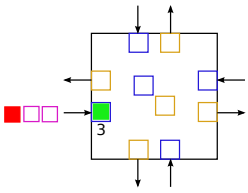
1.4 NoC Operation Examples

The following examples show the the concepts described previously in Section 1.3 at the cycle level (each figure shows the state of the router at each cycle, not after each event).

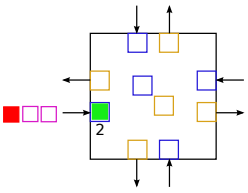
The first example, in Figure 1-4, shows how a single packet travels through a router. At cycle 1, the head flit of the packet arrives at the router. The number 3 by the head flit is how many cycles the routing algorithm will need to determine the correct output

port for the flit. Different routing algorithms need different amounts of time based on their complexity. At cycle 2, the count is reduced to 2. The head flit (and therefore the rest of the packet) is waiting. At cycle 3 the count is 1. At cycle 4, the routing algorithm has determined the desired output port and the router makes the internal connection (the solid line). The head flit is moved to the output port and the wire transfers the head flit to the next router. At cycles 5 and 6, the second and third flits follow the head flit. At cycle 7, the tail flit travels through the router and the router removes the internal connection.

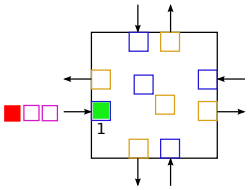
The second example (Figure 1-5) shows how a packet can wait if the requested output port is already being used. At cycle 1, the router has an internal connection from the south input port to the east output port. The packet that is using this connection (Packet A) has already transferred two flits. There is an incoming packet (Packet B) in the west input port of the router. At cycle 2 the third flit of Packet A is transferred, and the routing algorithm still needs two more cycles to determine the output port for Packet B. At cycle 3 the fourth flit of Packet A is transferred and the routing algorithm needs one more cycle for Packet B. At cycle 4, the fifth flit of Packet A is transferred and the routing algorithm finishes operating on the head flit of Packet B. Since the requested output port, east, is being used, the router remembers this decision/intent (the dotted line) and the packet is blocked. At cycle 5, the sixth flit of Packet A is transferred and Packet B is still waiting. At cycle 6, the tail flit of Packet A is transferred and the internal connection is removed. At cycle 7, the intent is changed to be a connection and the head flit of Packet B is moved.



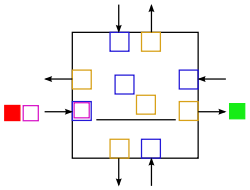
(a) Head flit arrives and routing starts



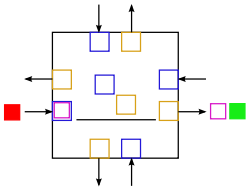
(b) Routing continues, two more cycles



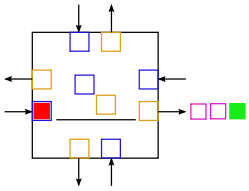
(c) Routing continues, one more cycle



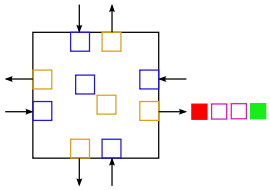
(d) Head flit is routed



(e) Second flit follows head flit

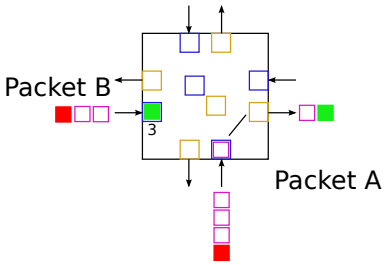


(f) Third flit follows head flit

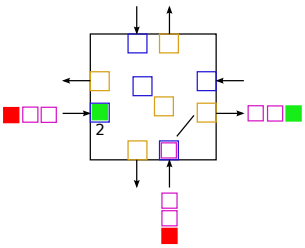


(g) Tail flit follows third flit, releases channel

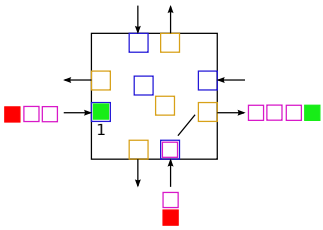
Figure 1-4: Routing taking several cycles to process a head flit



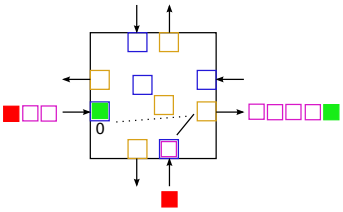
(a) Packet B arrives, routing is started, packet A continues to be transferred



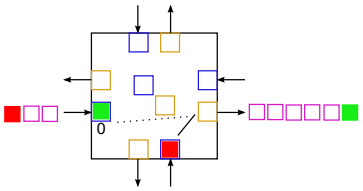
(b) Routing continues on packet B, packet A continues to be transferred



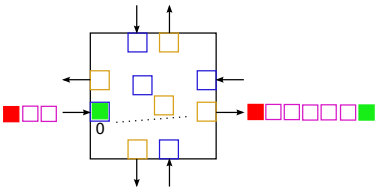
(c) Routing continues on packet B, packet A continues to be transferred



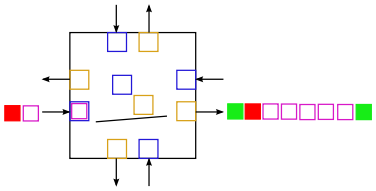
(d) Routing finishes, packet A has intent, packet A continues to be transferred



(e) Packet A waits, packet A continues to be transferred



(f) Packet A waits, packet A finishes transfer, releases channel



(g) Packet A starts being transferred

Figure 1-5: Routing taking several cycles to process a head flit with the desired output port already reserved

1.5 Challenges in a NoC

There are two types of problems in a NoC, problems the NoC could prevent and problems the NoC could not prevent. The problems the NoC could prevent are generally routing problems, mainly livelock and deadlock. Problems the NoC could not prevent are called faults.

1.5.1 Livelock and Deadlock

Deadlock and livelock are related to how a packet is routed through a network. Deadlock occurs when there is a cycle of dependencies in the network. Said in another way, there is a cycle that can be formed by following the internal connections and intents in a router and the wires that connect the routers. The flits that are part of this cycle can only move if the flit that is blocking is moved, and the flit that is blocking can only move if the original flit can move.

A NoC that uses a router queue size of one with an algorithm that is not deadlock free will see deadlock occur faster than using a queue of size 2. The same thing happens with queues of size 3 and 4, etc. If the queues are infinite, deadlock cannot occur because eventually, all packets will be able to move. However, infinite queues are not practical.

Livelock occurs when a flit is moving around the network and never reaches the intended destination. This can only happen if non-minimal paths are used and if the packet travels in a loop around the destination. If there is no loop possible, livelock cannot occur because the packet will eventually reach a destination, but the destination may not be correct.

1.5.2 Hardware Faults

There are two major types of hardware faults, permanent and transient. Permanent are usually manufacturing faults and never disappear. Transient faults exist only for a certain number of cycles then disappear.

An example of a permanent fault is a missing wire. If a wire is missing, the routing algorithm needs to be able to route around the broken wire or the NoC will not work. This

is because the routing algorithm will detect the broken wire and not move the packet or will send the packet over the missing wire, but actually the packet just disappears.

An example of a transient fault is a wire that does not transfer flits for a few cycles due to an event in the environment. If this happens and the router can sense the fault, a packet can be delayed or rerouted.

A good routing algorithm should detect faults so no packet is lost and reroute packets around faults such that delays are minimized, while making sure that livelock and deadlock do not occur.

CHAPTER 2

Routing Algorithms in a 2D Mesh

There are several routing algorithms that can be used on a 2D mesh. This chapter describes the ones that are used for comparison against the proposed routing algorithm. For simplicity, these routing algorithms do not use virtual channels. Virtual channels allow routing algorithms to time-multiplex physical channels. This has the effect of mimicking more connections between routers than there physically are. Virtual channels are not considered in this thesis because of added complexity in the routing algorithm and extra hardware required.

2.1 The Turn Model

In a 2D mesh, a packet can travel in one of four directions, north, east, south, and west. A turn is defined as a packet changing the direction of travel. For example, a packet could turn from east to north ($_{\rightarrow}{}^{\uparrow}$) or south to west ($_{\leftarrow}{}^{\downarrow}$). With this definition, there are 8 possible turns as shown in Figure 2-1.

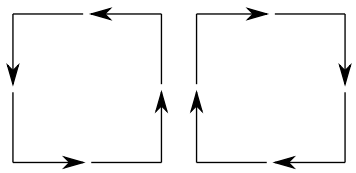


Figure 2-1: The eight turns in the turn model

In [8], the turn model is introduced and it is shown that deadlock can be avoided if at least one turn from each cycle is disallowed in the routing algorithm. In addition, algorithms

based on the turn model are livelock free. This is because, to have livelock, a packet must travel in a loop and since one turn from each cycle is removed, loops cannot occur. The algorithms based on the turn model may use either minimal paths or non-minimal paths.

2.2 XY

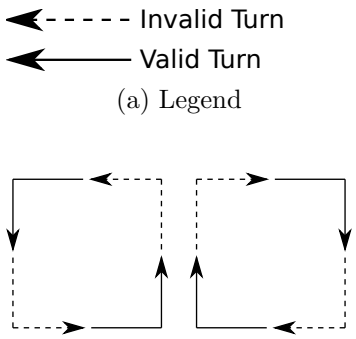
XY is the classic routing algorithm for network on chip [8]. It is simple, deadlock free, minimal, not adaptive, and based on the turn model. XY first routes a packet in the X direction then in the Y direction. Figure 2-2(b) shows the valid and invalid turns in XY. Dashed lines represent invalid turns and solid lines are valid turns. Two turns are eliminated from each cycle. This algorithm is also called *X first* or *static XY* to differentiate it from *dynamic XY*. Algorithm 1 shows the pseudo code for this routing algorithm.

Algorithm 1 XY

```
1  if at destination then
2     arrive
3  else if destination is west then
4     go WEST
5  else if destination is east then
6     go EAST
7  else if destination is north then
8     go NORTH
9  else if destination is south then
10    go SOUTH
11 end if
```

2.3 West-First

This routing algorithm is also based on the turn model. If the destination is to the west of the current router, the packet heads west then north or south to the destination. If the destination is not west of the current router, the packet can be routed adaptively along any minimal path. Figure 2-2(c) shows the valid and invalid turns in west-first turn model.



(a) Legend

(b) Turns in XY

(c) Turns in west-first

(d) Turns in negative-first

Figure 2-2: Turns for algorithms based on the turn model

Algorithm 2 shows the pseudo code for west-first. The adaptivity of this algorithm is shown on lines 18 and 20.

2.4 Negative-First

The routing algorithm negative-first prohibits turns from a positive direction to a negative direction. Positive directions are east and south and negative directions are north and west. Figure 2-2(d) shows valid the turns in this model. Algorithm 3 shows the pseudo code for this method. Packets routed with this algorithm can be routed adaptively north east (line 16) and south west (line 22).

Algorithm 2 West-First

```
1  if at destination then
2    arrive
3  else if destination in same column then
4    if destination is north then
5      go NORTH
6    else
7      go SOUTH
8    end if
9  else if destination in same row then
10   if destination is east then
11     go EAST
12   else
13     go WEST
14   end if
15 else if destination is west then
16   go WEST
17 else if destination is north then
18   go NORTH or EAST
19 else if destination is south then
20   go SOUTH or EAST
21 end if
```

2.5 Odd-Even

The odd-even routing algorithm [9] classifies columns as either odd or even according to the column's number. The first column is zero and is even, the second column is one and is odd etc. The algorithm prohibits certain turns depending on what column the packet is in. In an even column, the packet cannot turn north or south from west (Figure 2-3(a)). In an odd column a packet cannot turn east (Figure 2-3(b)).

This algorithm is both deadlock and livelock free. The proof for this uses the idea that in any cycle, there must be a rightmost column. Whether the rightmost column is even or odd, a cycle cannot occur because of the prohibited turns. Therefore, no cycle can exist.

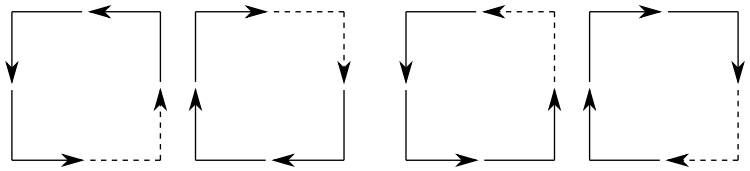
The added complication of odd and even columns allows the algorithm to be more adaptive than other turn model based routing algorithms. Algorithm 4 shows the pseudo code for this algorithm. The adaptivity for this algorithm is on line 35.

Algorithm 3 Negative-First

```
1  if at destination then
2    arrive
3  else if destination in same column then
4    if destination is north then
5      go NORTH
6    else
7      go SOUTH
8    end if
9  else if destination in same row then
10   if destination is east then
11     go EAST
12   else
13     go WEST
14   end if
15 else if destination is north east then
16   go NORTH or EAST
17 else if destination is north west then
18   go WEST
19 else if destination is south east then
20   go SOUTH
21 else if destination is south west then
22   go SOUTH or WEST
23 end if
```

Algorithm 4 Odd-Even

```
1  Valid_Directions = ∅
2  if at destination then
3     arrive
4  else if destination in same column then
5     if destination is north then
6        Valid_Directions ← (Valid_Directions ∪ NORTH)
7     else
8        Valid_Directions ← (Valid_Directions ∪ SOUTH)
9     end if
10 else if destination is east then
11    if destination in same row then
12       Valid_Directions ← (Valid_Directions ∪ EAST)
13    else
14       if this column is odd then
15          if destination is north then
16             Valid_Directions ← (Valid_Directions ∪ NORTH)
17          else
18             Valid_Directions ← (Valid_Directions ∪ SOUTH)
19          end if
20       end if
21       if destination is odd or more than one column away then
22          Valid_Directions ← (Valid_Directions ∪ EAST)
23       end if
24    end if
25 else
26    Valid_Directions ← (Valid_Directions ∪ WEST)
27    if this column is even and destination is not in the same row then
28       if destination is north then
29          Valid_Directions ← (Valid_Directions ∪ NORTH)
30       else
31          Valid_Directions ← (Valid_Directions ∪ SOUTH)
32       end if
33    end if
34 end if
35 adaptively choose a direction from Valid_Directions
```



(a) Valid and invalid turns in even columns

(b) Valid and invalid turns in odd columns

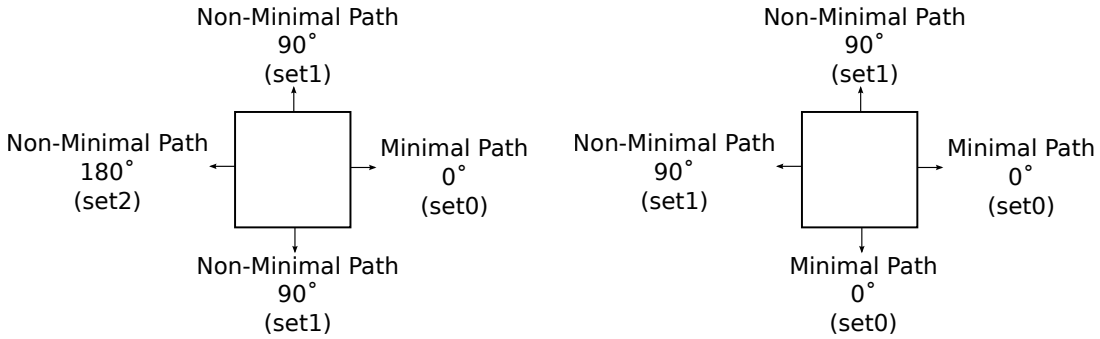
Figure 2-3: Valid and invalid turns in Odd-Even

2.6 Non-Minimal Odd-Even

The concept of odd and even columns in the (minimal) odd-even algorithm has been used to design a non-minimal odd-even algorithm (NMOE) [10]. This algorithm allows any turn at any point in the network as long as the turn itself is valid in the current column and if after the turn, the packet can still reach its destination without taking an invalid turn. The idea is that by allowing extra paths to be available, even if the paths are non-minimal, the packet can reach its destination faster.

When the routing algorithm chooses an output port, it can choose a direction from one of three sets. The first set contains directions that would have the packet follow a minimal path. The second set has the packet choose a direction that is 90° from a direction that follows a minimal path (e.g., north is 90° from east). The third set is a direction that is 180° from a direction that follows a minimal path (e.g., west is 180° from east). Figure 2-4 shows two of the possible combinations of 0°, 90° and 180° directions. If the routing algorithm only uses directions from set0, the algorithm will act like the minimal odd-even algorithm.

The routing algorithm chooses which set and direction to use by looking at the input queue size of the router in that direction. Initially, the algorithm looks only at directions from the minimal set (set0). If there is an input port that is not full along a direction from the minimal set, the routing algorithm chooses that direction. If all input ports are full using directions from the minimal set, the routing algorithm will consider the directions in the 90° set (set1). If all those directions are not available, the routing algorithm will attempt to use a direction from the 180° set (set2). If all valid directions are unavailable,



(a) Destination to the east (b) Destination to the southeast

Figure 2-4: Direction sets

the routing algorithm waits for the first valid direction that is available.

The pseudo code for populating the three sets is shown in Algorithms 5 through 11. The pseudo code for choosing a direction from the three sets is shown in Algorithm 12.

Algorithm 5 Odd-even NM

```
1  set0 = set1 = set2 = ∅
2  if at destination then
3     arrive
4  else if same column then
5     populate the sets using 'NMOE same column' (Algorithm 6)
6  else if same row then
7     populate the sets using 'NMOE same row' (Algorithm 7)
8  else if destination north east then
9     populate the sets using 'NMOE north east' (Algorithm 8)
10 else if destination south east then
11    populate the sets using 'NMOE south east' (Algorithm 9)
12 else if destination north west then
13    populate the sets using 'NMOE north west' (Algorithm 10)
14 else if destination south west then
15    populate the sets using 'NMOE south west' (Algorithm 11)
16 end if
17 adaptively choose a direction from the three sets
```

Algorithm 6 NMOE same column

```
1  if this column is odd then
2    if coming from the east then
3      set1 ← (set1 ∪ WEST)
4    end if
5    if destination is south then
6      set0 ← (set0 ∪ SOUTH)
7    else
8      set0 ← (set0 ∪ NORTH)
9    end if
10 else
11   set1 ← (set1 ∪ WEST)
12   if destination is south then
13     set0 ← (set0 ∪ SOUTH)
14     if column address is not 0 then
15       set2 ← (set2 ∪ NORTH)
16     end if
17   else
18     set0 ← (set0 ∪ NORTH)
19     if this column address is not 0 then
20       set2 ← (set2 ∪ SOUTH)
21     end if
22   end if
23 end if
```

Algorithm 7 NMOE same row

```
1  if this column is odd then
2    if destination is east then
3      set0 ← (set0 ∪ EAST)
4      if destination is more than 1 column away then
5        set1 ← (set1 ∪ NORTH ∪ SOUTH)
6      end if
7      if coming from east then
8        set2 ← (set2 ∪ WEST)
9      end if
10   else
11     set0 ← (set0 ∪ WEST)
12   end if
13 else
14   if destination is east then
15     set0 ← (set0 ∪ EAST)
16     set2 ← (set2 ∪ WEST)
17     if not coming from west then
18       set1 ← (set1 ∪ NORTH ∪ SOUTH)
19     end if
20   else
21     set0 ← (set0 ∪ WEST)
22     set1 ← (set1 ∪ NORTH ∪ SOUTH)
23   end if
24 end if
```

Algorithm 8 NMOE north east

```
1  if this column is odd then
2    set0 ← (set0 ∪ NORTH)
3    if destination is more than 1 column away then
4      set0 ← (set0 ∪ EAST)
5      set1 ← (set1 ∪ SOUTH)
6    end if
7    if coming from east then
8      set1 ← (set1 ∪ WEST)
9    end if
10 else
11   set0 ← (set0 ∪ EAST)
12   set1 ← (set1 ∪ WEST)
13   if not coming from west then
14     set0 ← (set0 ∪ NORTH)
15     set1 ← (set1 ∪ SOUTH)
16   end if
17 end if
```

Algorithm 9 NMOE south east

```
1  if this column is odd then
2    set0 ← (set0 ∪ SOUTH)
3    if destination is more than 1 column away then
4      set0 ← (set0 ∪ EAST)
5      set1 ← (set1 ∪ NORTH)
6    end if
7    if coming from east then
8      set1 ← (set1 ∪ WEST)
9    end if
10 else
11   set0 ← (set0 ∪ EAST)
12   set1 ← (set1 ∪ WEST)
13   if not coming from west then
14     set0 ← (set0 ∪ SOUTH)
15     set1 ← (set1 ∪ NORTH)
16   end if
17 end if
```

Algorithm 10 NMOE north west

```
1  if this column is odd then
2    set0 ← (set0 ∪ WEST)
3  else
4    set0 ← (set0 ∪ WEST, NORTH)
5    set1 ← (set1 ∪ SOUTH)
6  end if
```

Algorithm 11 NMOE south west

```
1  if this column is odd then
2    set0 ← (set0 ∪ WEST)
3  else
4    set0 ← (set0 ∪ WEST ∪ SOUTH)
5    set1 ← (set1 ∪ NORTH)
6  end if
```

Algorithm 12 Choice of direction for odd-even non-minimal [10]

```
1  for each set in set0, set1, set2 do
2    for each direction d in the set do
3      if the router's input queue in that direction is not full then
4        route the packet in that direction
5      end if
6    end for
7  end for
8  loop
9    wait for valid direction where the router's input queue in that direction is not full
10 end loop
```

2.7 DyXY

Dynamic XY (DyXY) [11] does not prohibit any turns from the turn model. Therefore, DyXY can deadlock, contrary to what the original paper claims. The authors of the original paper did not have deadlock in their experiments because their NoC simulator used unbounded queues in the routers. The algorithm cannot livelock because only minimal paths are used.

If more than one minimal path exists between a source and destination router, the algorithm will route adaptively by using stress values. A router's stress value is the current number of flits in all of the router's queues. So for each routing decision where two possible outcomes are valid, DyXY will choose the router with the smaller stress value for the output. The algorithm goes in the horizontal direction if the stress values are the same. Algorithm 13 shows the pseudo code for DyXY.

2.8 Adaptivity of Routing Algorithms

There are three levels of adaptivity for routing algorithms: not adaptive, partially adaptive, and fully adaptive [8]. An algorithm that is not adaptive has only one possible path for each source destination pair. An algorithm that is partially adaptive has at least one source destination pair with more than one path. For an algorithm to be considered fully adaptive, the algorithm must allow any path to be used between any source destination pair. XY is

Algorithm 13 DyXY

```
1  if at destination then
2     arrive
3  else if destination in same column then
4     if destination is north then
5        go NORTH
6     else
7        go SOUTH
8     end if
9  else if destination is in same row then
10    if destination is east then
11       go EAST
12    else
13       go WEST
14    end if
15 else
16    Dir1 ← horizontal direction of destination
17    Dir2 ← vertical direction of destination
18    if stress of Dir1 ≤ stress of Dir2 then
19       go Dir1
20    else
21       go Dir2
22    end if
23 end if
```

not adaptive; west-first, negative-first, odd-even, and DyXY are partially adaptive.

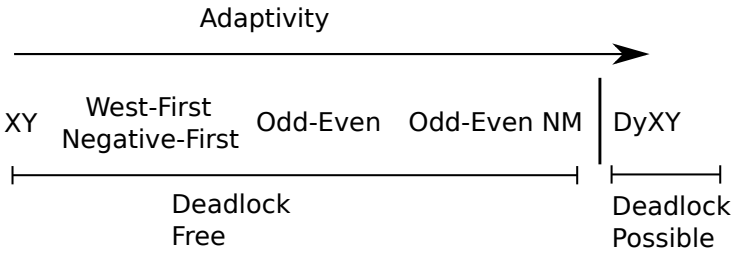


Figure 2-5: Adaptivity of Existing Routing Algorithms

Figure 2-5 puts these six routing algorithms on a scale of adaptivity with the routing algorithms on the left side less adaptive than the routing algorithms on the right. The line between odd-even NM and DyXY is the line of practicality. Routing algorithms on the right hand side of that line are not practical in a real NoC; these routing algorithms are only useful for simulations. This is because DyXY could not actually be implemented in a NoC in a deadlock free way.

Odd-even is more adaptive than both west-first and negative-first because on average odd-even has more paths between any source destination pair [9]. However, the added complication off odd and even columns makes the odd-even routing algorithm more complicated than west-first or negative-first. In general, the more adaptive a practical routing algorithm is, the more complicated it is.

No fully adaptive algorithm has been defined that is deadlock free and does not use preemption or virtual channels [12].

CHAPTER 3

Weighted Non-Minimal OddEven

Using the NMOE algorithm as the starting point, we have defined a new routing algorithm. The conclusion of the NMOE paper is that better decisions could be made when choosing a nonoptimal output port. We propose a new algorithm, called Weighted Non-Minimal OddEven (WeNMOE), which relies on five parameters to fine tune the behavior of the algorithm that calculates a routing cost for each output port. By combining and choosing the correct values for the parameters, the algorithm can choose a suitable output port when a packet is routed.

3.1 Routing Cost

The routing cost for a direction is based on the stress value for the neighboring router in that direction. This cost is modified by how full the corresponding input queue of the neighboring router is and whether the output port in that direction is along a non-minimal path:

$$\text{routingCost}(d, c) = \text{routerStress}(\text{neighbor}(d), c) \cdot \text{queuePenalty}(d) \cdot \text{dirPenalty}(d) \quad (3.1)$$

3.2 Stress Values

Stress values are used to guide the routing algorithm toward a route around congestion or faulty routers. The equation to calculate a router's current stress at cycle c is:

$$\text{currentStress}(c) = \alpha \cdot \text{queueStress}(c) + (1 - \alpha) \cdot \text{neighborStresses}(c) \tag{3.2}$$

The current stress is a combination of how many flits are in the router and the stress values of neighboring routers. The neighbor weight parameter α, $0 < \alpha \leq 1$, determines the relative weight of these values. By including the stress value of neighbors in the calculation, the congestion of a router can be propagated to other parts of the network. For α close to 1, the state of a router's neighbors are not propagated as much as when α is close to 0.

The queue stress of a router is the total number of flits currently in the router divided by the maximum number of flits the router can contain:

$$\text{queueStress}(c) = \frac{\sum_{queues} \text{queueSize}(c)}{\text{maxPossibleFlitsInQueues}} \tag{3.3}$$

The *neighborStresses* for a router is a function of all the stress values of the neighboring routers. For our algorithm, we use the average stress value of all the neighboring routers:

$$\text{neighborStresses}(c) = \frac{\sum_{neighbors} \text{routerStress}(\text{neighbor}, c - 1)}{\text{numNeighbors}} \tag{3.4}$$

The neighboring routers stress value is one cycle old. This is because the NoC needs one cycle to transfer the stress value of a router to its neighbors and makes the definition not circular.

The stress value that a router sends to its neighbors is calculated by by combining the current stress of the router and the stress value of the router last cycle:

$$\text{routerStress}(c) = \beta \cdot \text{routerStress}(c - 1) + (1 - \beta) \cdot \text{currentStress}(c) \tag{3.5}$$

The higher this value, the more congested the router is. The history weight parameter β ($0 \leq \beta < 1$) determines the relative weight of the current stress value and the past stress value. By including past stress values, the stress value of a router does not change quickly and reflects the stress values of neighbors. This is because a router's stress value increases and decreases with respect to the total number of flits in the router. When flits leave the current router, the flits most likely will have traveled to neighboring routers, which increases the stress value in the neighboring routers and decreases the stress value in the current router.

3.3 Queue Penalty

For a specific router, the same stress value is sent to all of the router's neighbors. This tells the neighbors the overall state of the router, but not the details of the input port that directly affects the current router. The queuing penalty is used for this purpose:

$$\text{queuePenalty}(d) = 1 + \omega \cdot \frac{\text{queueSize}(d)}{\text{maxQueueSize}} \tag{3.6}$$

The *queuePenalty* describes how full the input port of the neighboring router is. For a given direction d, it is the queue occupancy in this direction, scaled such that $1 \leq \text{queuePenalty} \leq 1 + \omega$. ω is called the queue penalty scale.

3.4 Direction Penalty

To influence the decision to take non-minimal paths, a function to penalize directions along a non-minimal path is used:

$$dirPenalty(d) = \begin{cases} 1 & \text{if minimal path} \\ 1 + \gamma & \text{if } 90^\circ \text{ from minimal path} \\ 1 + \delta & \text{if } 180^\circ \text{ from minimal path} \end{cases} \tag{3.7}$$

Th direction penalty function returns a value greater than or equal to 1 that describes how bad the direction d is relative to the minimal path. If d would have the packet follow a minimal path, the value is 1. If d is 90° from a minimal path (e.g., north is 90° from east) then this function returns $1 + \gamma$ and 'punishes' the direction for not being along a minimal path. This function does the same calculation for a direction that is 180° from a minimal path. The parameters γ and δ, $0 \leq \gamma \leq \delta$, determine how severe the $dirPenalty$ is when not using a minimal path.

3.5 Weighted Non-Minimal OddEven

The algorithm to choose a direction from the three sets in WeNMOE is:

Algorithm 14 Routing algorithm Weighted Non-Minimal OddEven

```
Populate the sets set0, set1, and set2 as is done in Algorithm 5
triple_set = ∅
for each set number in 0, 1, 2 do
   for each direction d in the set do
      triple_set ← (triple_set ∪ (routingCost(d), d, set number))
   end for
end for
triple = Min_≺ triple_set {compare triples using (3.8)}
route the packet in direction triple.d
```

First, the algorithm populates a triple set for each direction in each set. Then the lowest cost triple is found using:

$$(c_1, s_1, d_1) \prec (c_2, s_2, d_2) \stackrel{\triangle}{=} c_1 < c_2 \vee (c_1 = c_2 \wedge s_1 < s_2) \vee (c_1 = c_2 \wedge s_1 = s_2 \wedge d_1 < d_2) \quad (3.8)$$

First the triple with the smallest routing cost is chosen. If two triples have the same routing cost, then the triple with the lowest set number (closest to the minimal path) is

chosen. If two triples have the same set number and routing cost, then the direction with the highest priority is chosen. The priority of directions is (from highest to lowest) north, east, south, west and ensures that the algorithm remains deterministic.

3.5.1 Differences between WeNMOE and NMOE

One difference between the NMOE and WeNMOE routing algorithms is that WeNMOE always chooses an output port at the end of routing while NMOE may not choose an output port if all input queues are full. This difference is required because the decision NMOE makes to route a packet can be done in one cycle, where the decision WeNMOE makes may take multiple cycles. Therefore, if WeNMOE ended routing without making a decision, several cycles would have to be spent to route the packet again.

The second difference is that WeNMOE may choose a minimal path over a non-minimal path, even if the input queue along the minimal path is full. This is because according to the routing cost, waiting some cycles for the minimal path to be available may let a packet be delivered faster than taking a non-minimal path.

3.5.2 Configuring WeNMOE as NMOE

The proposed algorithm can be configured to be similar to NMOE in most cases. The only case where the algorithms would act differently is when the original algorithm would not choose an output port and the proposed algorithm would.

To configure WeNMOE to act like NMOE, the *queuePenalty* function would need to be changed to:

$$\text{queuePenalty}(d) = \left\lfloor \frac{\text{queueSize}(d)}{\text{maxQueueSize}} \right\rfloor \tag{3.9}$$

This is because no ω value can make the original function have binary outputs. In addition, *queueStress* function would be set to a constant value of 1:

$$\text{queueStress}(c) = 1 \tag{3.10}$$

The other parameters would be set as follows:

- $\beta = 0$
- $\delta = 0$
- $\gamma = 0$

The history weight value is 0 to ignore past values and there are no penalties to choosing non-minimal paths. These changes are required to ignore all values except *queuePenalty* when determining an output port.

The routing values and stress values used in this algorithm can be used in other 2D mesh routing algorithms. If the routing algorithm uses non-minimal paths, no changes are required. If the routing algorithm uses only minimal paths, the *dirPenalty* function should be removed from the routing cost equation and (3.1) replaced by (3.11):

$$\text{routingCost}(d) = \text{stress}(\text{neighbor}(d), c-1) \cdot \text{queuePenalty}(d) \tag{3.11}$$

Alternatively, γ and δ may be set to 0.

3.6 Implementation on a NoC

A NoC that implements Weighted Non-Minimal OddEven needs to send the stress value of a router to its neighbors every cycle. In addition, a credit based system is needed between the output and input ports as in NMOE. A credit system allows an input port and an output port to communicate so the output port will not send flits over the wire if there are no free spots for flits (credits). Finally, floating point arithmetic needs to be converted to integer arithmetic.

CHAPTER 4

Experimental Framework

A simulator was implemented based on the NoC model described in Chapter 1 and can perform the measurements described in this chapter. In addition, this simulator implements deadlock detection, basic fault generation, and basic verification of results.

4.1 Experiments

Each experiment starts with an empty NoC and is done using one packet injection rate, one traffic pattern, and one routing algorithm.

The packet injection rate represents the probability each router has to generate a new packet every cycle. For example, a packet injection rate of 2% means every cycle each router has a 2% chance to generate a packet.

Traffic patterns are functions that take the source for a packet and produce a destination. Table 4-1 shows the traffic patterns used and their definitions. In this table, i and j represent the row and column coordinates of a source router.

Bit reverse traffic takes the source row and columns bits, reverses the bits of them, and transposes the reversed results. Hotspot traffic sends traffic to a router chosen from a small set of destination routers using a uniform distribution. Complement traffic takes the row and column coordinates and subtracts then from the total number of rows minus 1 and the total number of columns minus 1 respectively to produce the destination coordinates. Transpose traffic uses the source's row address as the destination's column address and the source's column address as the destination's row address. Uniform random traffic randomly chooses a destination router according to a uniform distribution.

Bit Reverse
Src: $(i, j) \rightarrow$ Dest: (rev(j), rev(i))
Take the source row and column bits, reverse them, and transpose them

Hotspot (D)
Src: $(i, j) \rightarrow$ Dest: random router from set D
Send traffic to a router uniformly chosen from a set of possible destination routers

Complement
Src $(i, j) \rightarrow$ Dest: $(ROWS - 1 - i, COLS - 1 - j)$
Subtract source row and column from total number of rows and columns

Transpose
Src: $(i, j) \rightarrow$ Dest: (j, i)
Source row is the destination column and source column is destination row

Uniform Random
Src: $(i, j) \rightarrow$ (RAND, RAND)
Uniformly choose a random destination row and column

Table 4-1: Traffic Patterns

4.2 Measures

Many measurements can be used to evaluate the results of an experiment. These measurements include calculating the packet arrival rate, the number of misroutes during network operation, the number of different paths used between a source and destination, the total number of flits in the network, and the average packet latency. The two measures used here are the average number of flits in the network and the average packet latency, although other measures were used to validate the experiments.

4.2.1 Average Number of Flits in the Network

The average number of flits in the network describes the stability of the routing algorithm for a specific packet injection rate. If the average number of flits is stable throughout the experiment, the routing algorithm is able to route packets at least as fast as packets are injected into the network. This can be shown by plotting the number of flits in the network

at each cycle and fitting a line to the plot. (Figure 4-1 is a plot of this and is from an experiment using the XY routing algorithm and uniform random traffic. If the average number of flits in the network is not increasing, the thick line across the plot will have a slope close to 0. The y-intercept of that line is close to the actual average number of packets in the network.

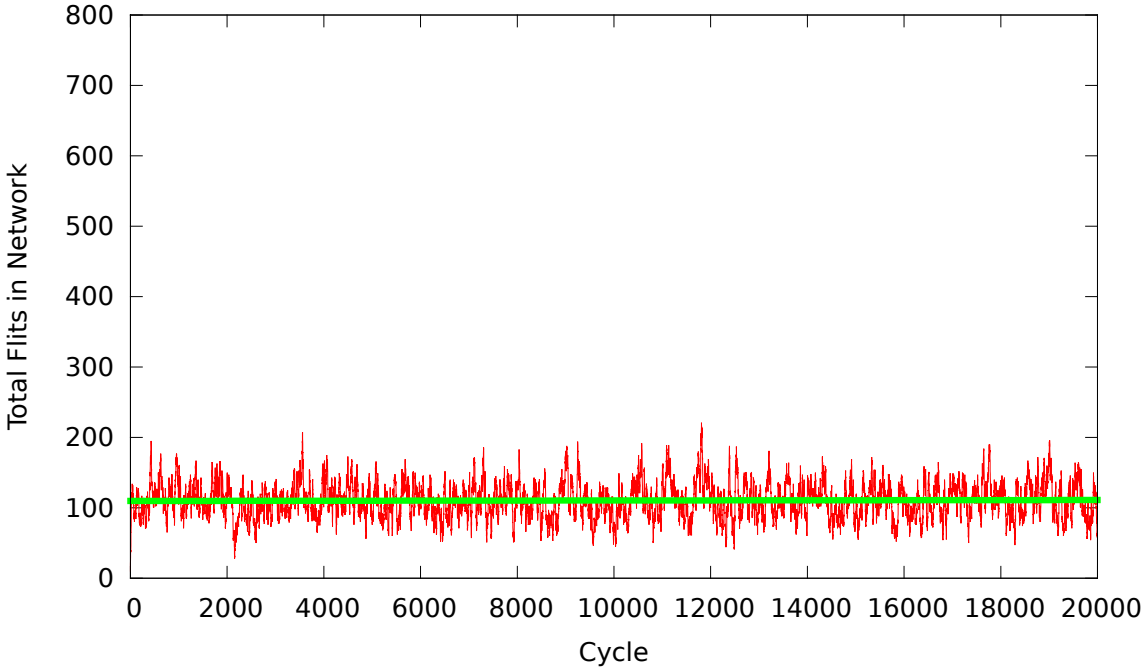


Figure 4-1: Total number of flits in the network for packet injection rate 1%

As the packet injection rate increases, the fit line will have a larger y-intercept because the average number of packets in the network also increases. The slope of the line also stays close to 0 as long as the routing algorithm can route packets fast enough. However, if the routing algorithm cannot route packets fast enough, the slope increases (Figure 4-2). Because the slope is not close the 0, the average number of packets in the network grows to infinity over time.

The performance of a routing algorithm cannot be measured if the slope of the fit line is not close to 0. This is because the routing algorithm has broken down and cannot route

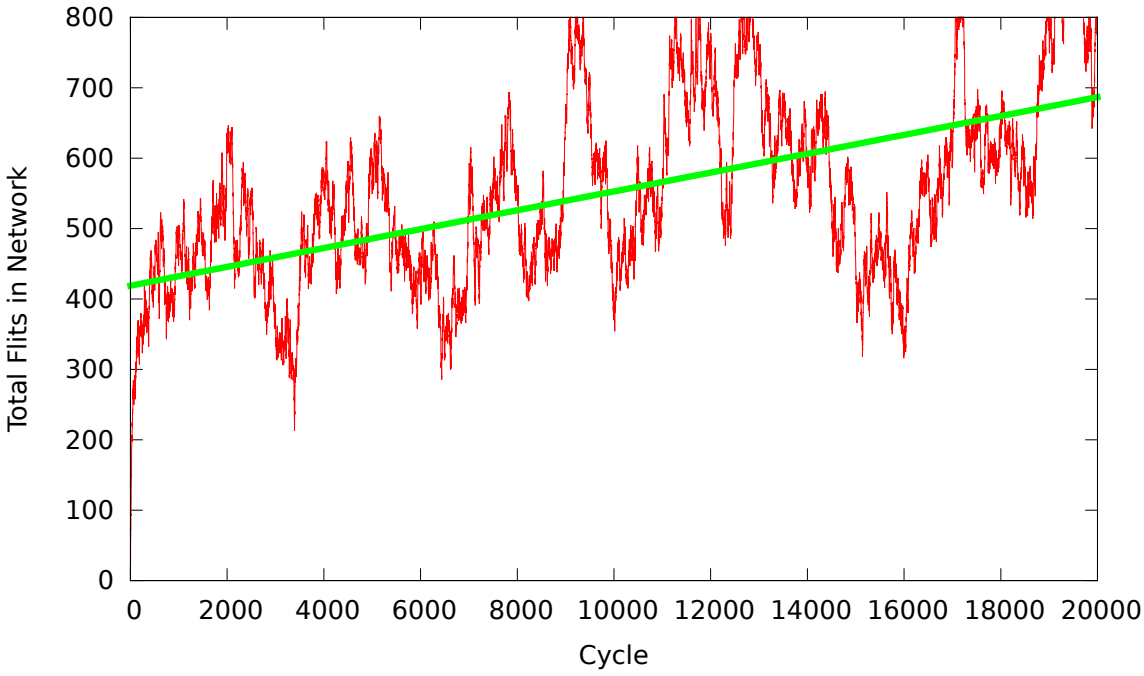


Figure 4-2: Total number of flits in the network for packet injection rate 2%

packets faster than they are generated. At this point, the network is saturated.

4.2.2 Packet Latency

Packet latency is the time it takes for a packet to be delivered, measured from the packet's generation to the delivery of the tail flit. The formula to calculate the latency d for each packet is:

$$d = d_i + \sum_{r \in path} (d_h + d_q(r) + 1) + d_d \tag{4.1}$$

d_i is number of cycles the packet is waiting in the core's queue after generation. Injection of a packet occurs when the head flit is added to a router's queue. It is possible that a packet is generated and immediately injected into the router ($d_i = 0$) because the router's queue has free space.

The sum represents the total latency of events that occur at each router along the packets path. These events are routing, queuing, and wire transfer. Routing takes d_h cycles

and is dependent on the routing algorithm used. $d_q(r)$ is the queuing wait, the time spent waiting in queues (not including routing) for router r. The more congestion on a network, the higher d_{qr}. The last term is constant and represent the time it takes for flits to travel across a wire.

d_d is the number of cycles the network needs to finish delivery of the packet. It is equal to the length of the packet minus two. This is because when a flit is being delivered, there is no wire transfer (minus 1 to offset the sum) and a flit can be delivered the last cycle of routing (minus 1 to not double count cycles).

4.2.3 Packet Latency at Many Injection Rates

To easily see when a routing algorithm breaks down, the average packet latency can be plotted over several packet injection rates (Figure 4-3).

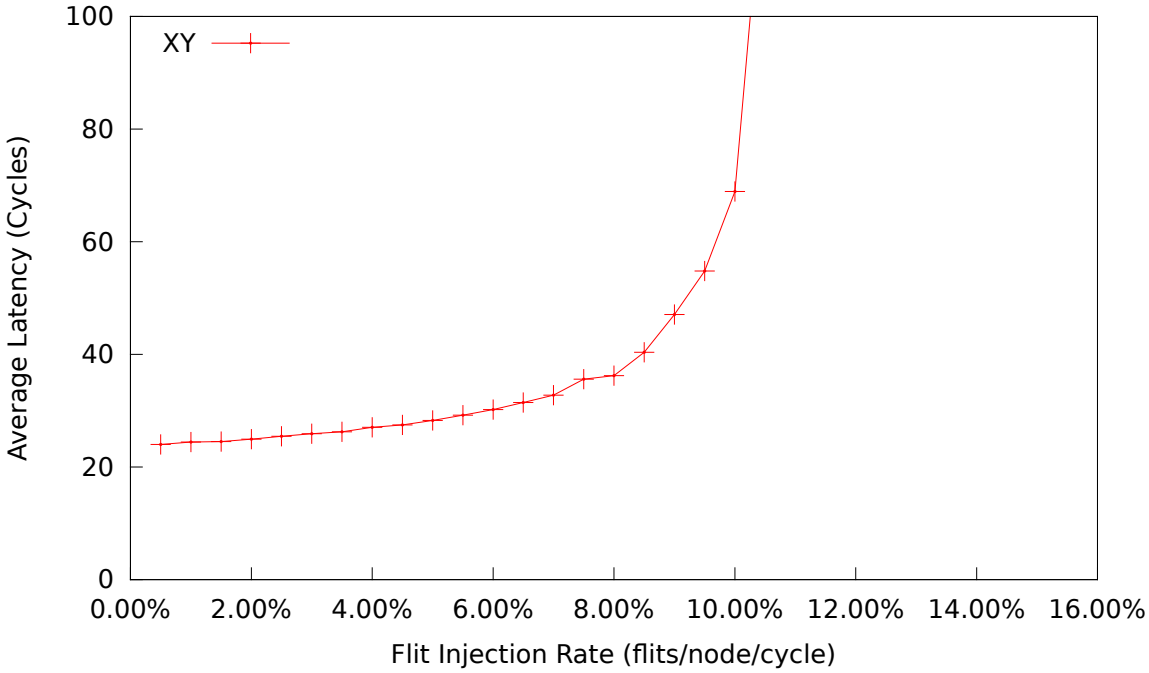


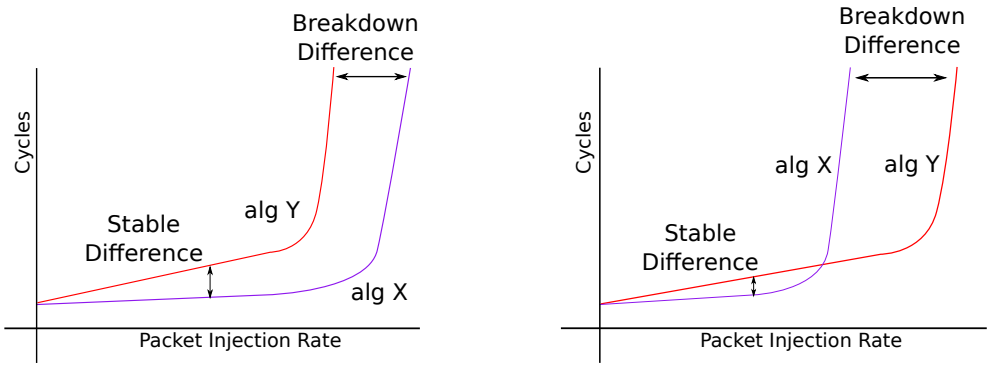
Figure 4-3: Average latency for XY in a 9×9 mesh

There are three parts to a plot that show how well a routing algorithm functions. The first part is at low packet injection rates, when the routing algorithm can route packets

faster than the core can generate packets (0.5% to about 7% in Figure 4-3). The second part is when the cores are generating packets at packet injection rates near the limit that routing algorithm can route packets (7.5% to about 8%in Figure 4-3). The final part is when the routing algorithm cannot keep up with the number of packets being injected, the network is saturated, and the average number of flits in the network is not stable.

The most significant part of this plot is the second part, where the limits of the routing algorithms plotted are reached. This part determines the highest injection rate the routing algorithm can handle.

4.2.4 Comparing Multiple Algorithms



(a) Algorithms do not cross

(b) Algorithms do cross

Figure 4-4: Important differences when comparing multiple algorithms

To compare multiple algorithms, each algorithm's average packet latency is plotted against the packet injection rate on the same plot. There are two differences to look for on these plots: the stable difference and the breakdown difference (Figure 4-4).

The stable difference shows how much better one algorithm is than another when the average number of flits in the network is stable. In both subfigures, algorithm X is better.

The breakdown difference describes how much better a routing algorithm is at packet injection rates that cause break down. In Figure 4-4(a), algorithm X is better than the algorithm Y, however in Figure 4-4(b) algorithm Y is better.

PIR	Actual Avg Time	Actual Avg Pkts	Calculated Avg Time	Calculated Avg Pkts	Ratio	% Error
0.1	5.951	0.556	5.556	0.595	1.071	7.1
0.2	5.949	1.112	5.560	1.190	1.070	7.0
0.3	5.911	1.670	5.566	1.773	1.062	6.2
0.4	5.936	2.243	5.607	2.374	1.059	5.9
0.5	5.945	2.806	5.612	2.973	1.059	5.9
0.6	5.946	3.361	5.601	3.567	1.061	6.1
0.7	5.927	3.905	5.578	4.149	1.063	6.3
0.8	5.951	4.488	5.610	4.761	1.061	6.1
0.9	5.951	5.062	5.625	5.356	1.058	5.8
1.0	5.962	5.630	5.630	5.962	1.059	5.9
1.1	5.971	6.201	5.637	6.568	1.059	5.9
1.2	5.943	6.734	5.612	7.131	1.059	5.9
1.3	5.956	7.300	5.616	7.742	1.061	6.1
1.4	5.964	7.875	5.625	8.350	1.060	6.0
1.5	5.961	8.440	5.626	8.942	1.060	6.0
1.6	5.965	9.011	5.632	9.545	1.059	5.9
1.7	5.967	9.566	5.627	10.144	1.060	6.0
1.8	5.956	10.124	5.624	10.721	1.059	5.9
1.9	5.958	10.694	5.628	11.320	1.059	5.9

Table 4-2: Little's Law results

Figure 4-4(b) shows a tradeoff between algorithm X and Y. For smaller packet injection rates, algorithm Y is worse, but can support higher injection rates before breakdown.

4.3 Little's Law

Little's law states, "the average number of customers in a queuing system is equal to the average arrival rate of customers to that system, times the average time spent in that system"[13]. In an equation representing NoC, this is $N = W\lambda$ where N is the average number of packets in the network, λ is the packet injection rate, and W is the average latency for packets.

For Little's Law to be applicable, the system has to be stable. In NoC this means that the slope of the line in the number of flits in a network must be close to 0.

Experiments were run and results are gathered in Table 4-2. These experiments used

the XY routing algorithm. The packet injection rate was varied from 0.1% to 1.9% and the packet length was one. The first three columns are the results from the experiment, the next two columns are the calculated results, and the last two columns are the ratios of the calculated results to the actual results and the percent error. The ratios are close to 1 and the percent error is small which helps show that this NoC model represents a valid queuing system.

CHAPTER 5

Experimental Evaluation

Parameters for WeNMOE were chosen by running experiments with bit reverse traffic with queue length of 1 and changing the values until no more improvement was found. The parameters used in WeNMOE are shown in Table 5-1.

The following experiments compare WeNMOE, using the set values from Table 5-1, against other routing algorithms. These experiments use different traffic patterns and injection rate. All experiments have a queue size of 1 and a packet size of 5. The network sizes used are 8×8 and 9×9. Using different sizes of the network allow the effects of having even or odd columns on the right most edge to be seen when using the odd-even based routing algorithms.

5.1 Bit Reverse

Experiments with bit reverse traffic were generated for flit injection rates between 0.5% and 11.5% at 0.25% increments for the 8×8 network.

The source and destination pairs that bit reverse traffic generates is shown in Figure 5-2. As is shown in the figure, this type of traffic generates small hotspots throughout the network.

Symbol	Value	Name
α	0.01	Neighbor Weight
β	0.3	History Weight
γ	1.25	Direction Penalty 90°
δ	2	Direction Penalty 180°
ω	2	Queue Penalty Scale

Table 5-1: Parameter values for WeNMOE

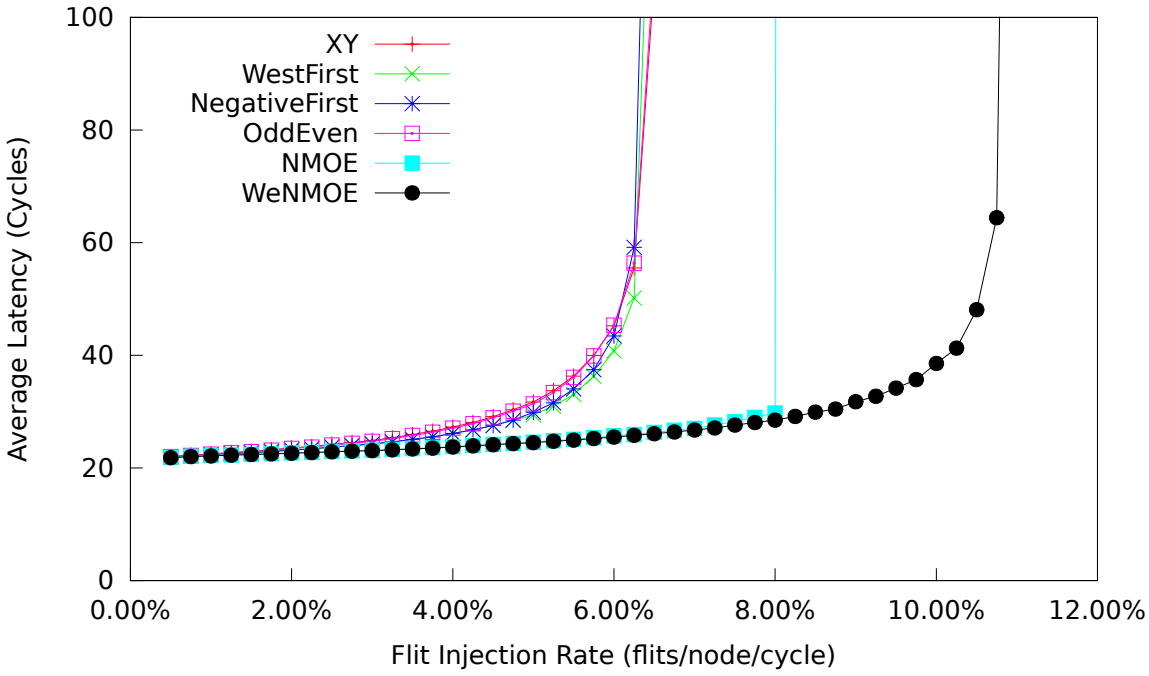


Figure 5-1: Bit reverse traffic on a 8×8 mesh

WeNMOE out performs all other algorithms on this type of traffic. This is because WeNMOE can route around the hotspots generated by bit reverse traffic.

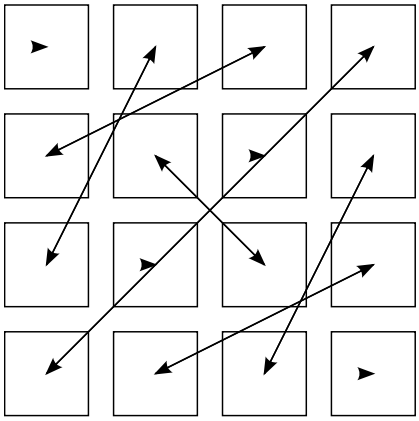
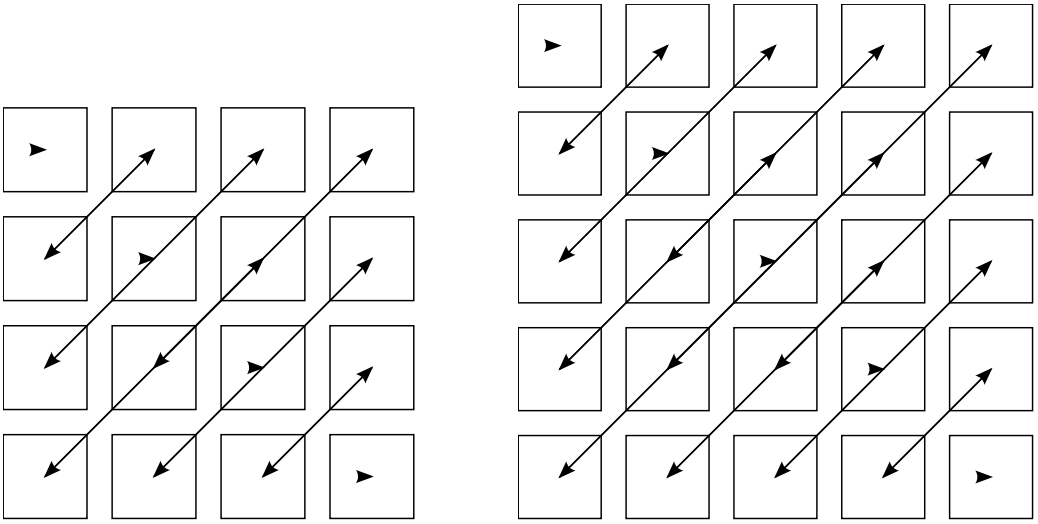


Figure 5-2: Source and destination pairs for bit reverse traffic on a 4×4 network



(a) Transpose traffic on a 4×4 network (b) Transpose traffic on a 5×5 network

Figure 5-3: Source and destination pairs for transpose traffic

5.2 Transpose

Experiments with transpose traffic were generated for flit injection rates between 0.5% and 12% at 0.25% increments for 8×8 and between 0.1% and 2% at 0.05% increments for 9×9. The source destination pairs for transpose traffic are shown in Figure 5-3.

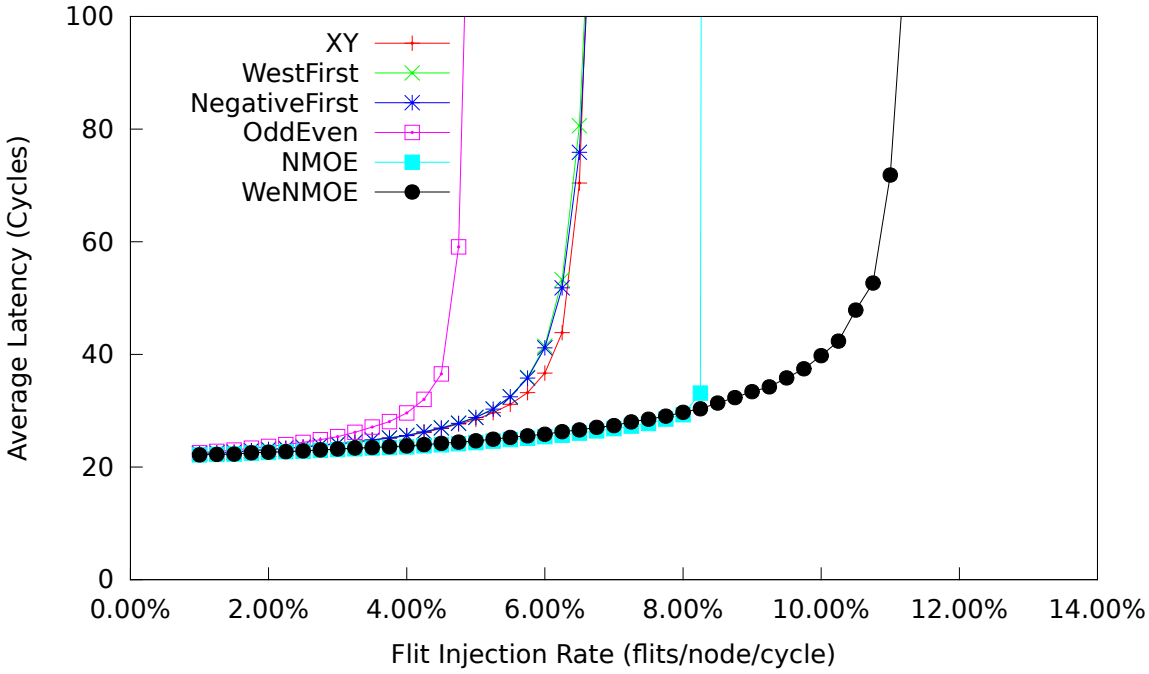


Figure 5-4: Transpose traffic on a 8×8 mesh

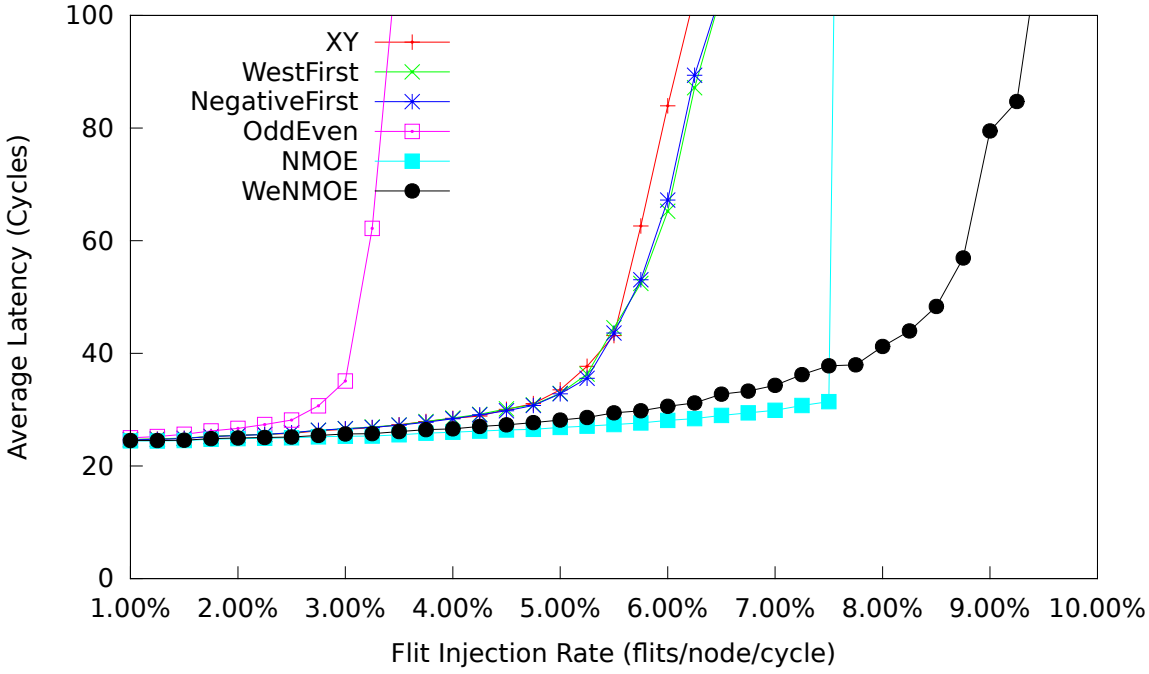
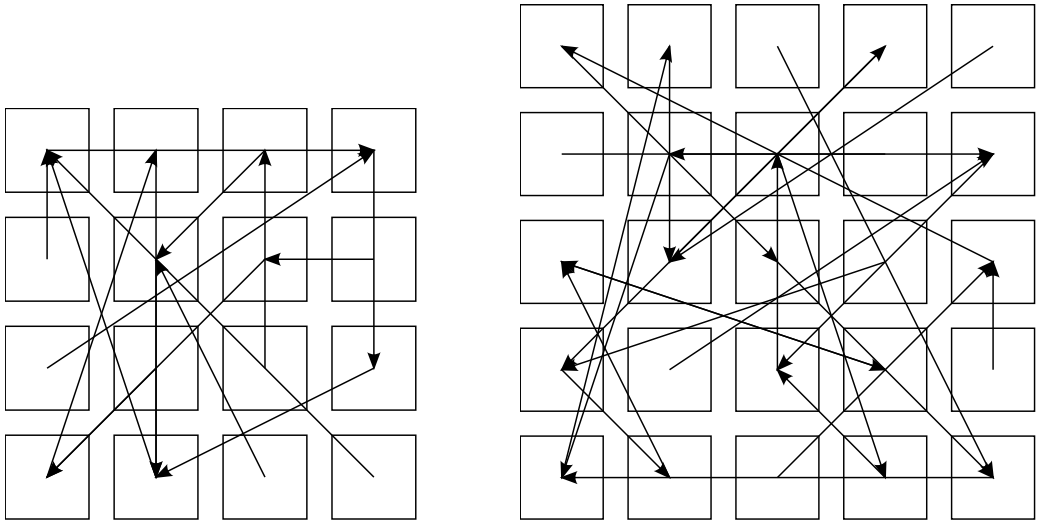


Figure 5-5: Transpose traffic on a 9×9 mesh

5.3 Uniform Random



(a) Uniform random traffic on a 4×4 network (b) Uniform random traffic 5×5 network

Figure 5-6: Example of source and destination pairs for uniform random traffic

Experiments with uniform random traffic were generated for flit injection rates between 3% and 12% at 0.25% increments. At 12%, all routing algorithms being tested have broken down. There is no large difference between results on a 9×9 or 10×10 network size.

A possible uniform traffic distribution is shown in Figure 5-6.

XY outperforms all other routing algorithms for uniform random traffic and WeNMOE performs second best. XY performs better because WeNMOE attempts to make smart choices on where to route packets, but the smart choices are not needed because the traffic is already balanced (the nature of uniform random) and these choices cause more congestion.

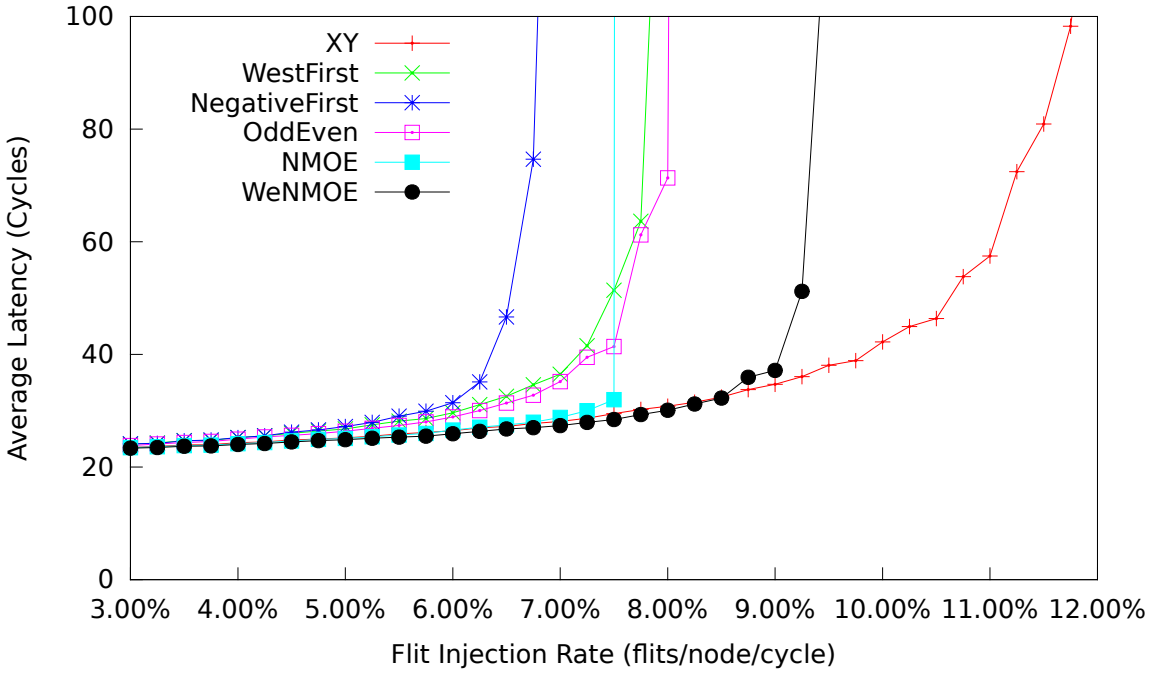


Figure 5-7: Uniform random traffic on a 8×8 mesh

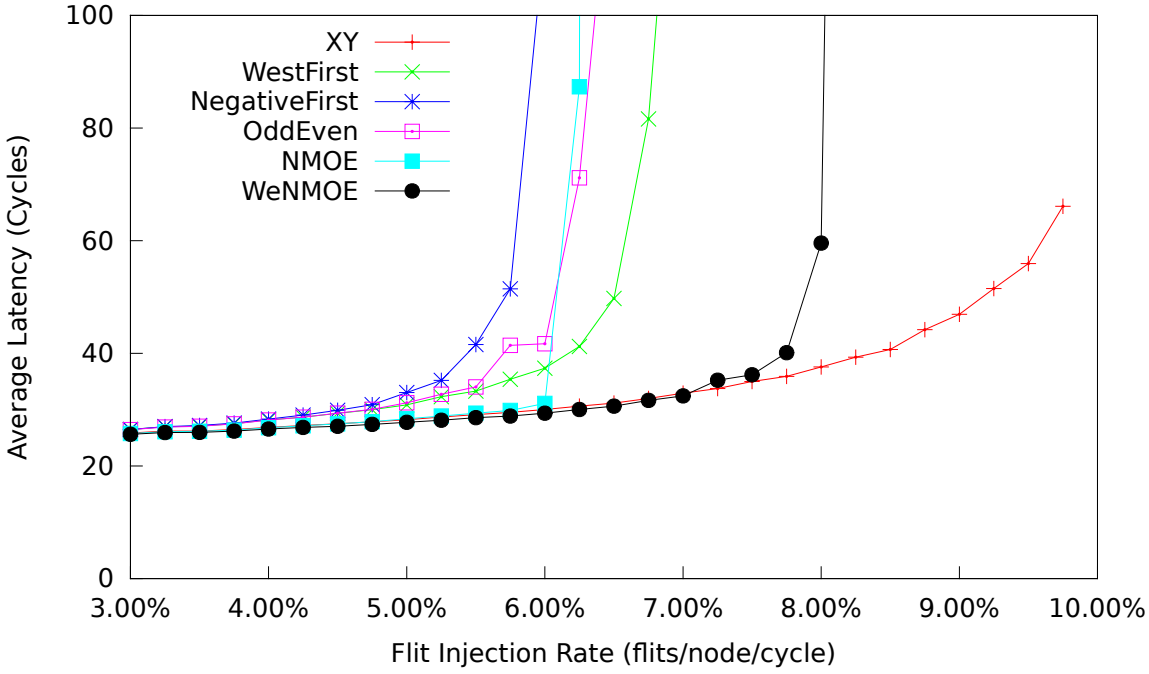
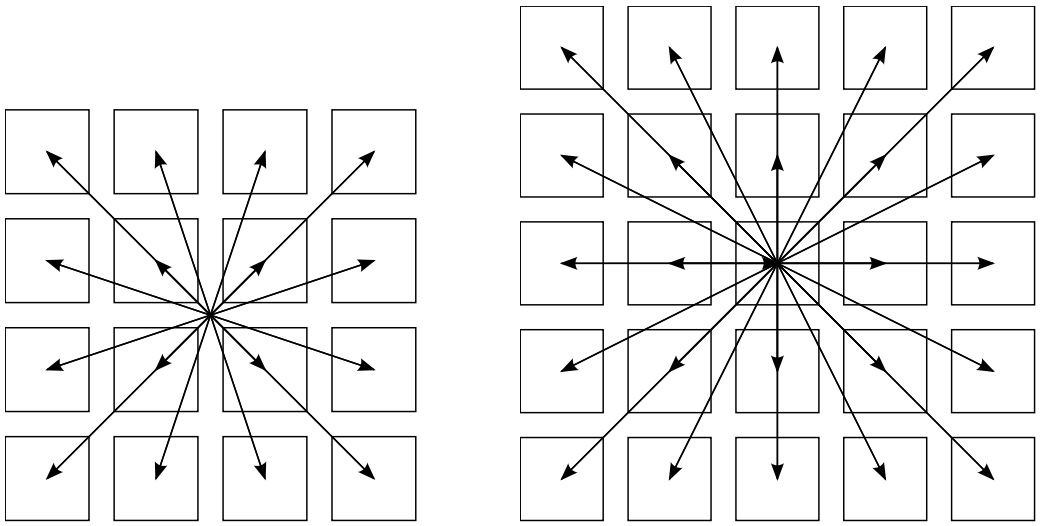


Figure 5-8: Uniform random traffic on a 9×9 mesh

5.4 Complement



(a) Complement traffic on a 4×4 network (b) Complement on a 5×5 network

Figure 5-9: Source and destination pairs for complement traffic

Experiments with complement traffic were generated for flit injection rates between 0.05% and 8% for 9×9 and between 0.3% and 8%. Both increase at 0.25% increments. At 8%, all routing algorithms being tested have broken down.

The source and destination pairs generated by complement are shown in Figure 5-9. This type of traffic generates a large hotspot in the center.

As is the same with uniform random, XY performs the best for this type of traffic, WeN-MOE performs second best. XY performs better because it naturally avoids the hotspots in the center of the network.

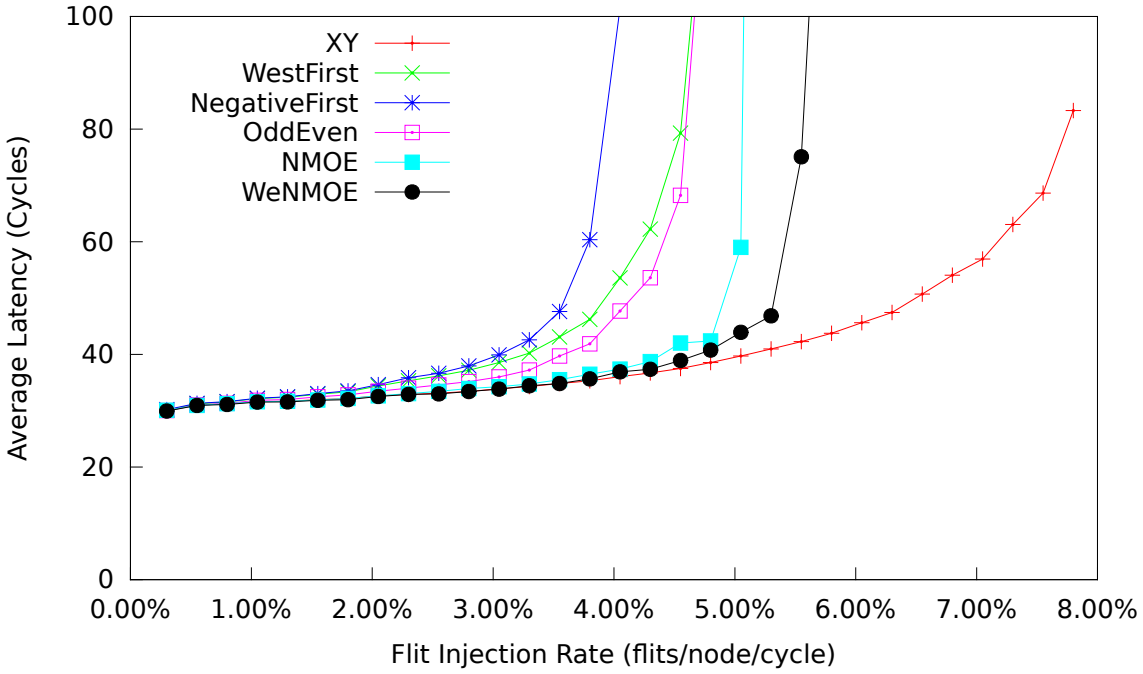


Figure 5-10: Complement traffic on a 8×8 mesh

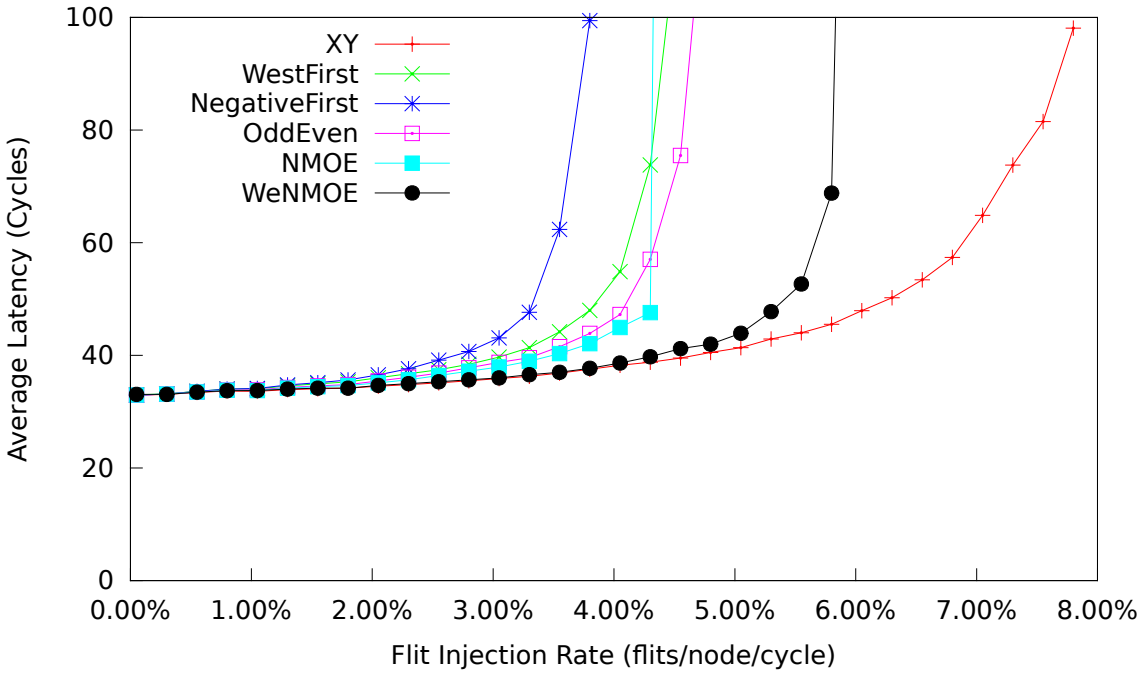
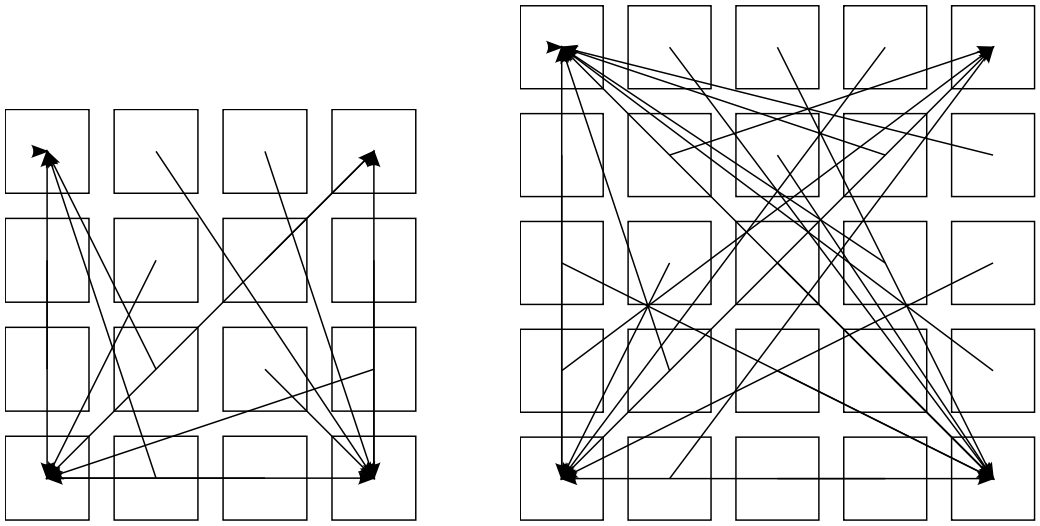


Figure 5-11: Complement traffic on a 9×9 mesh

5.5 Hotspot

Two types of hotspot traffic were generated, where the destinations are the four corners of the network (Figure 5-12) and where the destinations are the four corners of the center square (Figure 5-13).

On the 8×8 networks, WeNMOE and NMOE have the same performance and outperform all other algorithms as expected. These routing algorithms can route around heavily congested spots. However, on the 9×9 networks, WeNMOE outperforms NMOE. This difference may be attributed to the same difference between NMOE and WeNMOE that affected the transpose traffic.

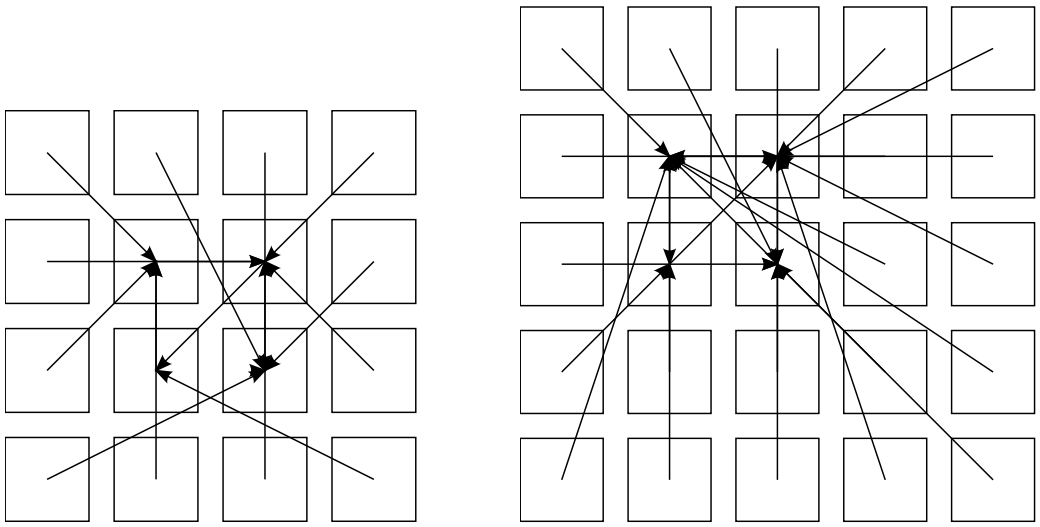


(a) Hotspot traffic on a 4×4 network (b) Hotspot traffic on a 5×5 network

Figure 5-12: Example of source and destination pairs for hotspot traffic to corners of a network

5.6 WeNMOE and NMOE

The reason WeNMOE outperforms NMOE consistently (the breakdown difference), is because WeNMOE always chooses an output port where NMOE does not. When the packet injection rate gets to large enough values, NMOE waits for the first open input port, which



(a) Hotspot traffic on a 4×4 network (b) Hotspot traffic on a 5×5 network

Figure 5-13: Example of source and destination pairs for hotspot traffic to the center of a network

has a chance to be along a non-minimal path where WeNMOE will choose a minimal path when all routers have a high stress value.

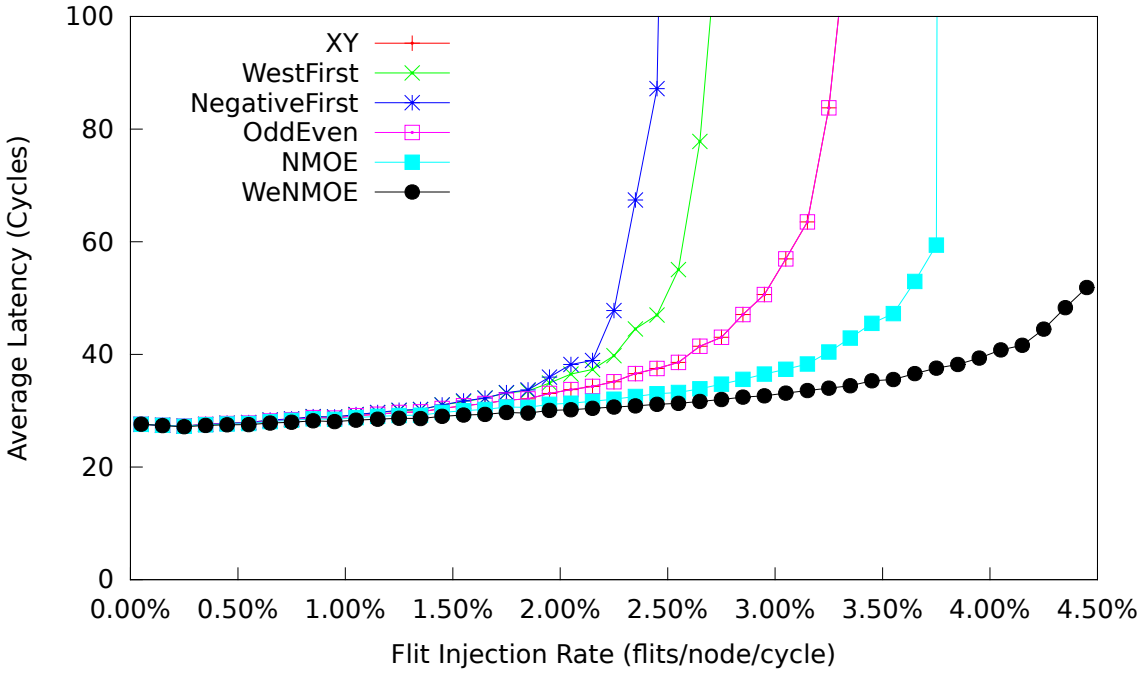


Figure 5-14: Hotspot traffic sending to the four corners on a 8×8 mesh

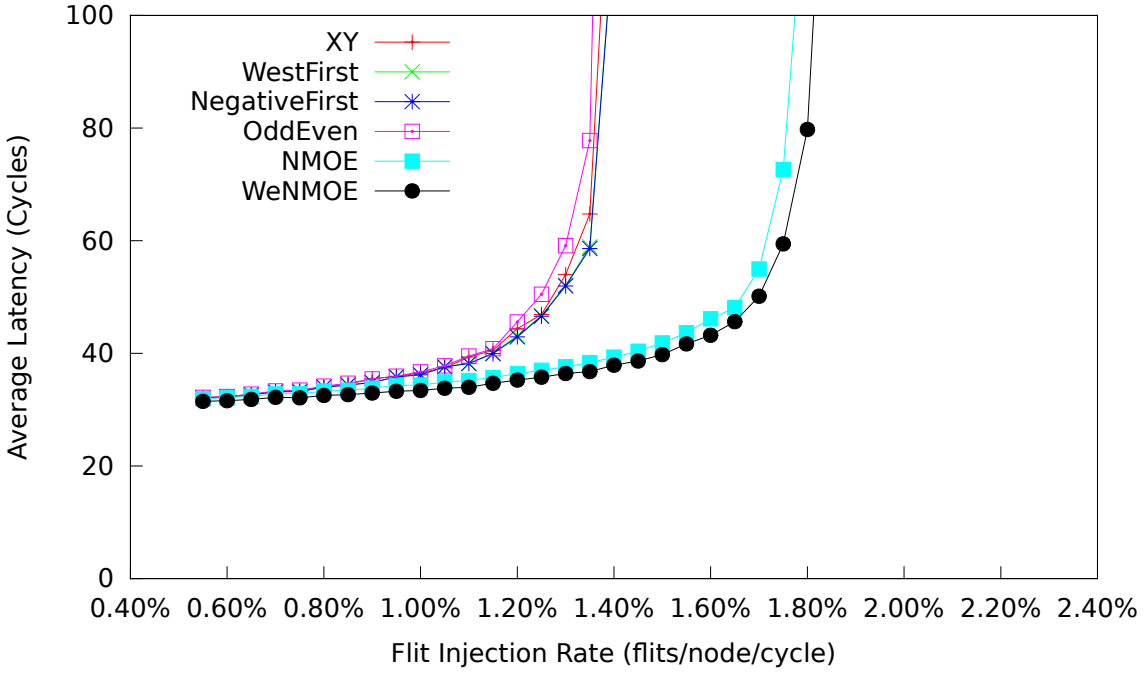


Figure 5-15: Hotspot traffic sending to the four corners on a 9×9 mesh

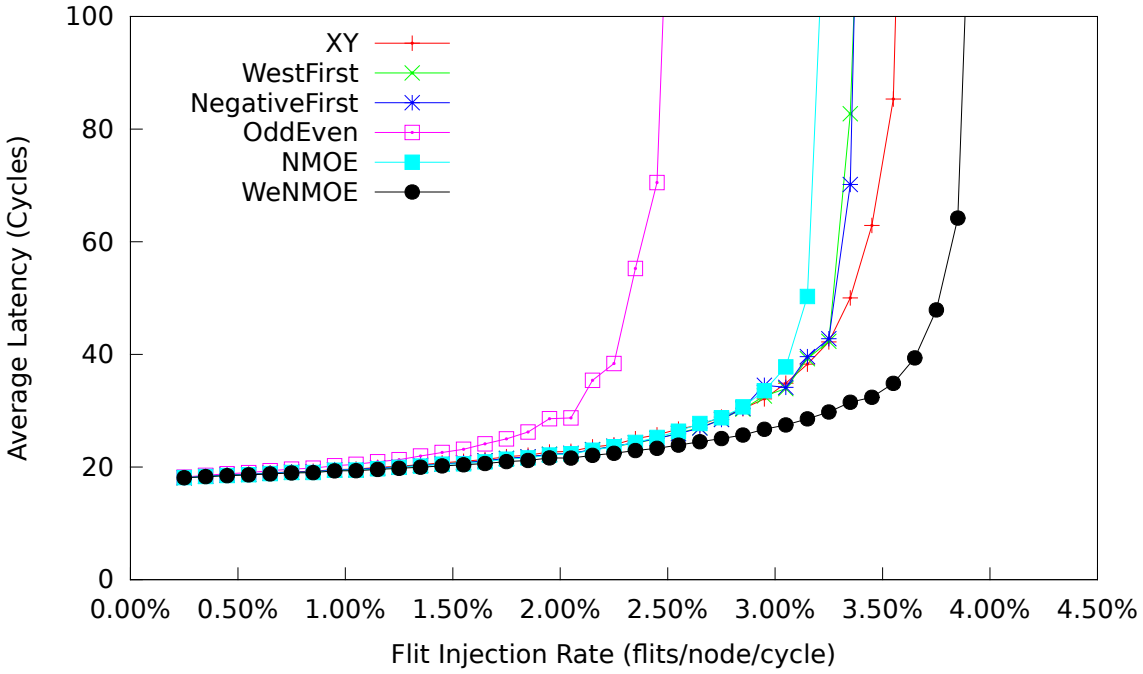


Figure 5-16: Hotspot traffic sending to the center square on a 8×8 mesh

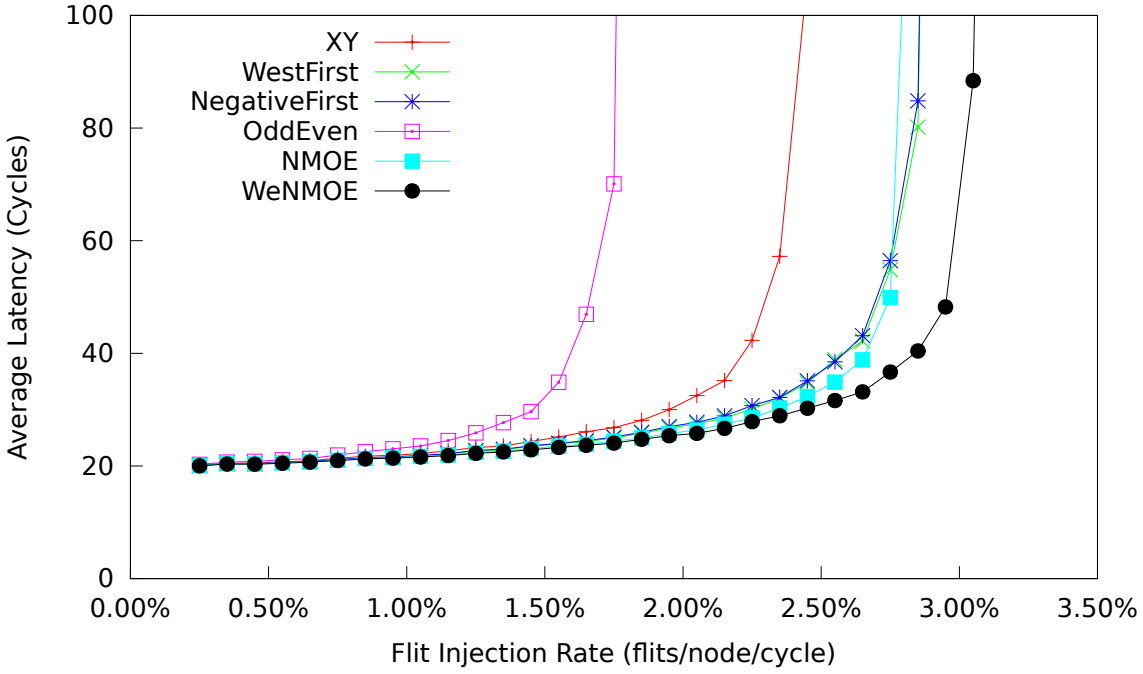


Figure 5-17: Hotspot traffic sending to the center square on a 9×9 mesh

CHAPTER 6

Conclusion

A Network on chip (NoC) moves data among components of a single chip. NoC uses a regular layout of routers and employs routing algorithms to direct packets consisting of flits through the network. NoCs face many challenges including congestion, faults, deadlock, and livelock. To avoid congestion and faults, adaptive routing algorithms route packets around congested or faulty routers. In order to prevent deadlock and livelock, routing algorithms on a 2D mesh can use the *turn model* [8] that prohibits certain turns in a NoC to avoid dependency cycles.

The non-minimal odd-even routing algorithm (NMOE) [10], adaptively chooses a direction to route a packet and is deadlock free because it is based on the turn model. This thesis shows how the performance of NMOE can be improved by using more information to make the decision to choose a direction. The proposed routing algorithm, Weighted Non-Minimal OddEven (WeNMOE), improves on NMOE by adding a routing cost estimation mechanism. The routing cost of each output port is based on the router's stress value, the state of the input port in the next router, and whether the direction is on a minimal path or not.

A router's stress value is an exponential weighted average of the current stress of the router and past stresses. The *history weight* β controls which stress value affects the router's stress more. The current stress of a router is the weighted sum of the total number of flits in the router divided by the maximum possible flits in the router and the average router stress of the four neighboring routers. The *neighbor weight* α controls how much stress values of neighboring routers affect the current router's stress.

The *queue penalty* describes how full the input port on the next router is. The *queue weight* ω controls how much the queue penalty affects the routing cost.

The *direction penalty* of an output port is a function that returns a value that depends on the possible direction relative to a minimal path. There are three possible values, 1, if the direction is on a minimal path, $1 + \gamma$ if the direction is 90° from a minimal path and $1 + \delta$ if the direction is 180° from a minimal path.

Using a simulator developed to evaluate 2D mesh NoC routing algorithms, WeNMOE was compared with the NMOE, XY, negative-first, west-first, and odd-even algorithms. The experiments were on two different network sizes, 8×8 and 9×9, and used different traffic patterns: bit reverse, hotspots, complement, transpose, and uniform random. WeNMOE outperforms all existing routing algorithms under all traffic patterns listed above with the exception of only two (complement and uniform random) where XY performed better. This is due to the uniformity of the traffic load on those patterns, which causes adaptive algorithms to create congestion that normally does not exist.

WeNMOE has a few limitations as well. Since the routing algorithm has more choices for possible directions, more hardware would be required to implement WeNMOE on a NoC. In addition, the floating point based routing cost would have to be done using integer arithmetic.

In future work, more types of traffic should be explored to evaluate WeNMOE and to better understand its benefits and drawbacks. These traffic patterns can include hotspot traffic on top of uniform random traffic and hotspots distributed randomly throughout the network. In addition, different values for the parameters of WeNMOE should be evaluated. The current values are based on WeNMOE's performance for bit reverse traffic, and may not provide the best performance for other types of traffic. Furthermore, the method used to generate the routing cost, queue penalty, and direction weight should be further explored to see if different approaches would result in better performance.

BIBLIOGRAPHY

[1] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2004.

[2] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comput. Surv.*, vol. 38, June 2006.

[3] E. Salminen, A. Kulmala, and T. D. Hamalainen, "Survey of network-on-chip proposals," March 2008.

[4] T. Mak, P. Cheung, K.-P. Lam, and W. Luk, "Adaptive routing in network-on-chips using a dynamic-programming network," *Industrial Electronics, IEEE Transactions on*, vol. 58, pp. 3701 –3716, aug. 2011.

[5] Intel, "Single-chip cloud computer." `http://techresearch.intel.com/ProjectDetails.aspx?Id=1`, 2012. [Online; accessed 27-April-2012].

[6] Intel, "Tera-scale computing research program." `http://techresearch.intel.com/ResearchAreaDetails.aspx?Id=27`, 2012. [Online; accessed 27-April-2012].

[7] Intel, "Teraflops research chip." `http://techresearch.intel.com/projectdetails.aspx?id=151`, 2012. [Online; accessed 27-April-2012].

[8] C. Glass and L. Ni, "The turn model for adaptive routing," in *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pp. 278 –287, 1992.

[9] G.-M. Chiu, "The odd-even turn model for adaptive routing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 11, no. 7, pp. 729–738, Jul.

[10] W.-C. Tsai, K.-C. Chu, Y.-H. Hu, and S.-J. Chen, "Non-minimal, turn-model based NoC routing," *Microprocessors and Microsystems*, 2012. Article in Press, doi: 10.1016/j.micpro.2012.08.002.

[11] M. Li, Q.-A. Zeng, and W.-B. Jone, "DyXY - a proximity congestion-aware deadlock-free dynamic routing method for network on chip," in *Design Automation Conference, 2006 43rd ACM/IEEE*, pp. 849 –852, July 2006.

[12] L. Ni and P. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, pp. 62–76, Feb.

[13] L. Kleinrock, *Queueing systems*. New York: Wiley, 1975.

[14] W. Jang and D. Pan, "Application-aware NoC design for efficient SDRAM access," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, pp. 1521 –1533, oct. 2011.

[15] M. A. Yazdi, M. Modarressi, and H. Sarbazi-Azad, "A load-balanced routing scheme for NoC-based systems-on-chip," in *Hardware and Software Implementation and Control of Distributed MEMS (DMEMS), 2010 First Workshop on*, pp. 72 –77, june 2010.

[16] S. Rodrigo, J. Flich, A. Roca, S. Medardoni, D. Bertozzi, J. Camacho, F. Silla, and J. Duato, "Cost-efficient on-chip routing implementations for CMP and MPSoC systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, pp. 534 –547, april 2011.

[17] S. Pasricha and Y. Zou, "NS-FTR: A fault tolerant routing scheme for networks on chip with permanent and runtime intermittent faults," in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pp. 443 –448, jan. 2011.

[18] Z. Zhang, A. Greiner, and S. Taktak, "A reconfigurable routing algorithm for a fault-tolerant 2D-mesh network-on-chip," in *Proceedings of the 45th annual Design Automation Conference*, DAC '08, (New York, NY, USA), pp. 441–446, ACM, 2008.

[19] S. Pasricha, Y. Zou, D. Connors, and H. J. Siegel, "OE+IOE: a novel turn model based fault tolerant routing scheme for networks-on-chip," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 85–94, ACM, 2010.

[20] C. Feng, Z. Lu, A. Jantsch, J. Li, and M. Zhang, "A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning for network-on-chip," in *Proceedings of the Third International Workshop on Network on Chip Architectures*, NoCArc '10, (New York, NY, USA), pp. 11–16, ACM, 2010.

[21] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-aware prioritization mechanisms for on-chip networks," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 280–291, ACM, 2009.

[22] A. K. Mishra, O. Mutlu, and C. R. Das, "An application-oriented approach for designing heterogeneous network-on-chip," Tech. Rep. CSE-11-007, Pennsylvania State University, June 2011.

[23] S. Ma, N. Enright Jerger, and Z. Wang, "DBAR: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 413–424, ACM, 2011.