Evaluation of Real-time Performance Models in Wormhole-routed On-chip Networks

Master of Science Thesis
In Electronic System Design

By

Karl-Henrik Nielsen

Stockholm, March 2005

Supervisor: Zhonghai Lu
Examiner: Axel Jantsch
Abstract

Network-on-Chip (NoC) is advocated as a communication platform to interconnect tens even hundreds of heterogeneous processors, IP cores etc. on a single chip for future System-on-Chip (SoC) design. One important application domain is real-time systems such as multimedia applications.

For real-time applications, satisfying the timing constraints of messages is essential for performance and design quality. However, mapping messages with a timing constraint on the network imposes a challenge for NoC design since the contentions in the network introduce nondeterministic behavior, resulting in possibly the violation of the messages' timing constraints. It is therefore important to conduct a feasibility test in order to determine whether messages meet their timing bounds or not.

The feasibility of real-time messages is analyzed on the basis of a worst-case performance model. For wormhole-switched networks, three worst-case performance models and their corresponding feasibility test algorithms have been proposed in the literature, namely, lumped link (LL), blocking dependency graph (BDG), and contention tree (CT) model. All the three models assume priority-based arbitration and deterministic routing.

This project is aimed to compare and evaluate the three feasibility tests in terms of network utilization and pass ratio of feasible messages. To this end, a network simulator is designed and programmed in the OMNeT++, which is a discrete-event simulation environment. In addition, the three feasibility tests are implemented in C++ (partially based on previous work). The results of the feasibility tests are evaluated against the simulation results using the network simulator developed.

Two types of traffic are used for the evaluation. One is uniformly distributed random message streams, the other is for an M-JPEG encoder. Simulation results show that the CT is the optimal choice for a real-time feasibility test, the BDG the second followed by the LL.
Acknowledgements

This master thesis was carried out at the Department of Microelectronics and Information Technology’s Laboratory of Electronics and Computer Systems (IMIT/LECS). It is a part of the System-On-Chip for Mobile Internet master program at the Royal Institute of Technology (KTH). It will lead to the Degree of Master of Science with specialisation in System-on-Chip design.

The work on this master thesis has been carried out by Karl-Henrik Nielsen in the time period from July 2004 until March 2005.

My supervisor during this master thesis was Zhonghai Lu, and I would like to thank him for all the guidance, discussions and support he has given me throughout the master thesis. His door was always open, and there was never a question he could not answer. I have learned a lot from him, and he has my deepest gratitude. I would also like to thank my examiner Professor Axel Jantsch and my opponent Siegfried Albel.

At last I would like to give thanks to the OMNeT++ community.

Table of Contents

1 Introduction ............................................................................................................. 1
   1.1. Background and Problem Statement ............................................................... 1
   1.2. The Thesis Work Process ............................................................................... 2
   1.3. Outline of the Thesis ....................................................................................... 3

2 Background on Network-on-Chip ........................................................................ 5
   2.1. Buses ................................................................................................................. 5
   2.2. Network-on-Chip .............................................................................................. 6
   2.3. Topologies ........................................................................................................ 7
       2.3.1. Fully Connected Network ........................................................................ 7
       2.3.2. Fat Tree .................................................................................................. 7
       2.3.3. 2-D Meshes and Tori ............................................................................. 8
   2.4. Routing Algorithm .......................................................................................... 9
   2.5. The Switching Technique ............................................................................... 10
       2.5.1. Messages, Packets and Flits ................................................................... 10
       2.5.2. Store and Forward Packet Switching ................................................... 11
       2.5.3. Virtual Cut Through Packet Switching .................................................. 11
       2.5.4. Wormhole Switching ............................................................................. 12
       2.5.5. Circuit Switching ................................................................................... 14
   2.6. The Switch Design .......................................................................................... 15
       2.6.1. Buffers and Crossbar ............................................................................. 15
       2.6.2. Arbitration Algorithm ........................................................................... 16
       2.6.3. Flow Control .......................................................................................... 16

3 The Feasibility Tests ............................................................................................ 19
   3.1. General Presentation ....................................................................................... 19
   3.2. The Lumped Link Model ............................................................................... 23
   3.3. The Blocking Dependency Graph ................................................................... 27
   3.4. The Contention Tree Model .......................................................................... 30

4 The Network Simulation Tool ............................................................................ 33
   4.1. NED and C++ ................................................................................................. 34
   4.2. Simple Modules .............................................................................................. 36
   4.3. The Messages .................................................................................................. 36
   4.4. The Graphical User Interfaces ....................................................................... 36
   4.5. The Simulation Files and Execution ............................................................... 37
5 The Network Simulator Developed ................................................................................. 39
   5.1 Internal Switch Processes and Lane States ............................................................ 40
   5.2 The XY Routing Algorithm .................................................................................. 42
   5.3 The Switch Arbitration and Lane Allocation Algorithm ...................................... 43
   5.4 Credit-Based Flow Control .................................................................................. 43
   5.5 Simulator Structure ............................................................................................ 46
      5.5.1 The Topology ................................................................................................. 47
      5.5.2 The Switch .................................................................................................... 47
      5.5.3 The Link and Gates ....................................................................................... 48
      5.5.4 The Generator .............................................................................................. 48
      5.5.5 The Upload Logic .......................................................................................... 48
      5.5.6 The Lane Allocator ....................................................................................... 51
      5.5.7 The Forward Lane Allocator Array ............................................................... 52
      5.5.8 The Switch Allocator and Crossbar ............................................................... 52
      5.5.9 The Input Gates ............................................................................................ 52
      5.5.10 The Receiver ............................................................................................... 54
   5.6 Self Triggered Modules and Priority ...................................................................... 54

6 The Simulation Results ............................................................................................... 55
   6.1 Definition of Simulation Result Units ..................................................................... 55
   6.2 The NoC Simulator Validation .............................................................................. 56
      6.2.1 Latency vs. Link Utilization ......................................................................... 58
      6.2.2 Throughput .................................................................................................. 58
      6.2.3 Latency vs. Link Utilization Based on Priority Ranges .................................. 59
      6.2.4 Latency vs. Number of Packets Received ..................................................... 60
   6.3 The Real-Time Feasibility Random Simulations ..................................................... 61
      6.3.1 Random Message Set ................................................................................... 61
      6.3.2 NoC Simulation Process ............................................................................... 62
      6.3.3 Simulation Time ......................................................................................... 63
      6.3.4 Head of Line Blocking ................................................................................. 63
      6.3.5 Simulation Settings ....................................................................................... 64
      6.3.6 The Feasibility Models ................................................................................. 65
      6.3.7 The Feasibility Tests Performed .................................................................... 65
      6.3.8 Feasibility Tests with Routing Delay Equal to 1 .......................................... 67
      6.3.9 Feasibility Tests with Routing Delay Equal to 2 .......................................... 76
   6.4 An M-JPEG Implementation Real-Case Study ......................................................... 80
      6.4.1 M-JPEG ....................................................................................................... 80
      6.4.2 Parallel Pipelined Encoder Architecture ..................................................... 80
      6.4.3 Module Descriptions .................................................................................... 81
      6.4.4 Simulation Implementation .......................................................................... 82
      6.4.5 Simulations .................................................................................................. 86

7 Conclusion .................................................................................................................. 89

Abbreviations ................................................................................................................. 91

Reference List ............................................................................................................... 93
List of Tables

Table 1: Attributes of contending message example ..................................................... 22
Table 2: The features of the feasibility models .............................................................. 23
Table 3: The adjacency matrix of the contending BDG example ................................... 28
Table 4: Message attributes for the network simulator validation ................................... 57
Table 5: Simulation configurations for the network simulator validation ....................... 57
Table 6: Simulation settings for random message set simulations .................................. 64
Table 7: Feasibility model attributes ............................................................................ 65
Table 8: Medium message simulation attributes ........................................................... 65
Table 9: Variation in the number of virtual channel simulation attributes ....................... 68
Table 10: Statistics information for variation in VC for $U_{OE\, LL}$, $U_{OE\, BDG}$ and $U_{OE\, CT}$ .................. 71
Table 11: The equal storage space simulation attributes .............................................. 71
Table 12: Statistics information for equal storage size for $U_{OE\, LL}$ ............................ 74
Table 13: Statistics information for equal storage size for $U_{OE\, BDG}$ ......................... 74
Table 14: Statistics information for equal storage size for $U_{OE\, CT}$ ............................ 74
Table 15: Variation in the number of cells per VC simulation attributes ......................... 75
Table 16: Statistics information for variation of number of cells per VC for $U_{OE\, LL}$, $U_{OE\, BDG}$ and $U_{OE\, CT}$ ........................................................................................................... 76
Table 17: Variation in the number of VCs with routing delay 2 simulation attributes ....... 77
Table 18: Message stream attributes for M-JPEG real-case ......................................... 84
Table 19: The first 2 pipelined parallel M-JPEG message stream attributes ................. 86
List of Figures

Figure 1: A 16-terminal SPIN fat tree topology ................................................................. 8
Figure 2: A 2-D mesh topology with bi-directional links ..................................................... 9
Figure 3: A message, packet and flits ................................................................................ 11
Figure 4: A header flit, payload flit and tail flit ................................................................. 13
Figure 5: A deadlock in a wormhole routed network ....................................................... 13
Figure 6: A simple input buffered switch design ............................................................ 16
Figure 7: Priority inversion example .............................................................................. 21
Figure 8: Example of contending messages ..................................................................... 22
Figure 9: The first contention graph example for the lumped link model ....................... 25
Figure 10: The timing diagram for the first lumped link model example ......................... 26
Figure 11: The second contention graph example for the lumped link model ............... 26
Figure 12: The timing diagram for the second lumped link model example ................. 26
Figure 13: The contention graph example for the blocking dependency graph model .. 27
Figure 14: Initial timing diagram of message 0 from BDG example .............................. 28
Figure 15: Initial timing diagram of message 1 from BDG example .............................. 28
Figure 16: Initial timing diagram of message 3 from BDG example .............................. 29
Figure 17: Improved timing diagram of message 0 from BDG example ....................... 30
Figure 18: Improved timing diagram of message 1 from BDG example ....................... 30
Figure 19: Final timing diagram of message 3 from BDG example ............................... 30
Figure 20: Contention Tree example for the CT model .................................................... 31
Figure 21: Timing diagram of message 4 for the CT example ........................................ 32
Figure 22: Timing diagram of message 3 for the CT example ........................................ 32
Figure 23: Final timing diagram for message 5 in the CT example ............................... 32
Figure 24: Concrete example of nested modules ........................................................... 35
Figure 25: Example of a compound module which contains simple modules .......... 35
Figure 26: The files, compilers and linker needed for an OMNeT++ simulation ............ 37
Figure 27: A screenshot from OMNeT++ that shows us the switch developed ............ 39
Figure 54: Missed messages in percent vs. LBU for variation of number of cells per VC for $U_{OE\, LL}, U_{OE\, BDG}$ and $U_{OE\, CT}$ ................................................................. 76
Figure 55: Generated traffic vs. link bandwidth utilization for $U_{LL}, U_{BDG}$ and $U_{CT}$ with header routing delay equal to 2 ...................................................................................... 77
Figure 56: Generated traffic vs. link bandwidth utilization for variation in VC for $U_{OE}$ with routing delay 2 .................................................................................................................. 78
Figure 57: Missed messages in percent vs. LBU for variation of number of VCs for $U_{OE\, LL}, U_{OE\, BDG}$ and $U_{OE\, CT}$ with delay 2 ................................................................. 79
Figure 58: M-JPEG encoder structure ........................................................................................................ 80
Figure 59: Parallel M-JPEG encoder architecture ............................................................................. 81
Figure 60: N pipelined parallel M-JPEG encoder architecture example ..................................... 85
Figure 61: Generated traffic vs. link bandwidth utilization for the real-case study ............ 87
Figure 62: Link bandwidth utilization vs. number of infeasible pipelines for the real-case study ................................................................................................................. 88
Chapter 1

Introduction

1.1. Background and Problem Statement

A Network-on-Chip design methodology is presented in Kumar et. al. [1]. It states that a NoC design process involves the backbone design, platform design and system design phase. With a NoC backbone we mean the topology and communication issues related to the design. This means choosing/designing the right topology, switches, channels, communication protocols and network interfaces. In the platform design phase we come to issues like design of the resource nodes, scaling of the network and system level control. In the last phase we map the application to the resources and validate the results. The application tasks will run on the resources and communicate with other resources using messages. In real-time systems it is crucial that the message streams between the resources will meet their deadlines. A feasibility test in the system design phase will make it possible to determine the latency and feasibility of these message streams. The designer can with the help of this test determine the practicability of the NoC mapping. This will lead to a reduction in design time and therefore also the cost.

For wormhole routed networks is it difficult to guarantee that a message will reach its deadline. Feasibility testing for real-time communication on wormhole routed networks has therefore been an important issue in multiprocessor systems over the recent years. The lumped link [2, 3] and blocking dependency graph models [4] are two feasibility tests that have been presented for multiprocessor system design. As wormhole based networks is becoming more and more accepted by NoC designers [5, 6, 7, 8], there is a need for real-time feasibility tests also in the NoC design phase. A contention tree feasibility test [9] was therefore introduced for these purposes.
The real-time feasibility tests for parallel computing can also be used in the design of a NoC. The complexity of such a test is of little importance to the NoC developer as long as it is performed offline. The lumped link, blocking dependency graph and contention tree feasibility tests need to be compared and evaluated on their deliverable network utilization and pass ratio for use on NoCs. We also need to verify the accuracy of these feasibility tests.

1.2. The Thesis Work Process

The work was divided into five phases.

The five phases were:
- Literature study and OMNeT++ set-up
- Conceptual phase of network simulator
- Code and validation of network simulator
- Implementation of the three feasibility tests
- Evaluation and summary of results

In the first phase, background material on NoCs was studied. In addition OMNeT++ was installed and set-up. In the conceptual phase, there was developed a concept for the wormhole routed network simulator. All the aspects of a wormhole switch were considered. The network simulator was then coded and validated for correctness. Tests on the code were also done throughout the coding phase. Several minor errors in the code were discovered in the validation phase for the simulator. Errors were hard to detect, and led to that the validation consumed a great deal of time.

The next phase was to implement the lumped link, blocking dependency graph and contention tree model in C++ code. Code for the lumped link and contention tree was provided by my supervisor Zhonghai Lu. This code was tested and a few changes were made. The blocking dependency graph model however needed to be coded from scratch.

In the last phase, an evaluation was done for the feasibility tests and a rapport of the work was written.
1.3. Outline of the Thesis

- **Chapter 2:**
  Chapter 2 gives a description of Network-on-Chip. It presents the issues related to a NoC design such as topology, routing algorithm, switching technique, arbitration algorithm, flow control and channel buffers. The chapter also relates to the design issues of NoC architectures that have or are being developed.

- **Chapter 3:**
  In chapter 3 we present the three feasibility tests. In order to give a thorough understanding of the differences between the feasibility tests, the chapter gives several examples of the functionality of the tests.

- **Chapter 4:**
  This chapter describes the Objective Modular Network Testbed in C++ (OMNeT++). OMNeT++ is a discrete-event simulation tool. The chapter only gives a brief description of the background material needed in order to understand the build up of the network simulator.

- **Chapter 5:**
  Here we find a detailed presentation of the network simulator implementation with descriptions of both OMNeT++ and networking issues.

- **Chapter 6:**
  In this chapter we show how the simulations were executed and confirm correctness of the network simulator. We also evaluate and compare the three feasibility tests and present the results we found.
Chapter 2

Background on Network-on-Chip

At the end of the decade we will see further technology scaling that will allow building of System-on-Chips (SoCs) with several hundreds of components and 4 billion transistors on the chip. It will be a great challenge to design and implement a low power, high bandwidth and reusable way to communicate between the different IP-cores on the chip [10]. How will the communication between the tens or hundreds of heterogeneous processors, coprocessors and other IP cores be implemented? The main bottleneck of such a design will be in the communication between the cores. Many researchers in the academic world and the industry have therefore realized that there is a need for a better way to implement the communication on the chip than by using buses.

2.1. Buses

So why can’t we simply use buses in the future for communication between the IP-cores as we do now. There are several limitations that will make the buses less useable. The continuing increase of different IP-cores on the chip will make it impossible to use a bus configuration with a high bandwidth. Buses do not scale well with the system size because the bandwidth of the bus is shared between all cores attached to it. Buses are a good choice if the number of IP-cores is small, but as we know the number of IP-cores will increase. The operating frequency will also degrade with the increasing number of cores attached to the bus, because of the capacitive loading in the wires. As the wire lengths increases, so will also the power consumption of the chip. And as we know there are many efforts in the SoC area today to keep the power consumption of the chip as low as possible.
2.2. Network-on-Chip

One solution to the routing problem above called Network-on-Chip [11, 12] has been accepted by parts of the academic world, and several different NoCs are being developed. The concept of the NoC has been adopted from the parallel computers field. The main advantage that makes the NoC a superior solution is that it is scalable. The topologies used in the NoCs do not degrade the bandwidth in such a large extent as in buses. In some topologies, the bandwidth is not degraded at all. The NoCs therefore offer high bandwidth communication between IP cores. With NoCs the issue of reuse comes to mind. Will it be possible to reuse the networks from time to time? The answer is yes. All the IP-cores are connected to the network via a standard common interface. It will therefore be possible to reuse both the network and the IP-cores. With the possibility of reuse we get a small time to market and reduced engineering costs.

A SoC is the integration of several heterogeneous components such as CPUs, DSPs, memories, FPGAs and ASICs on a single chip [13]. These resources are connected to a bus or a network interconnection such as the NoC. In a NoC the nodes in the network communicate through switches by passing messages. The nodes are attached to the network with the help of network interfaces (NI).

The NoC is physically built up by means of switches, channels and network interfaces in an ordered way. A NoC can be described by its topology, routing algorithm, switching technique and arbitration algorithm. All these topics will be discussed in this chapter. Output scheduling, flow control and channel buffering are closely related to switch design and they will also be examined. Network interfaces and links will not be presented in the master thesis. For information about network interfaces, I refer to the OCP [14] and the VCI [15] standards which are two commonly used interfaces for NoCs.

The performance of an interconnection network can be characterized by throughput and latency. The throughput of a system is the average number of flits received at a node per cycle. Latency is the time it takes for a message to be transmitted from the source to the receiver. Latency includes switch delay, routing delay, link delay and blocking. Switch delay is the time for a flit to pass through a switch, while with routing delay we mean the time it takes for a header flit to pass through. Link delay is the time it takes over a
transmission line, while blocking means the total time the message is blocked from acquiring system resources by other messages in the network.

2.3. Topologies

The nodes in the NoC are interconnected through links or channels. A network can either be direct or indirect [16]. In a direct network a node has a point-to-point connection to a number of other nodes in the network. A switch in the indirect network may not have a node attached to it. Messages may need to pass several nodes in order for it to reach another switch with a resource connected to it. The topology defines how these nodes are interconnected with each other in a structured way. There are several defined topologies used for NoCs. Fully connected networks, fat trees, meshes and tori are to name a few.

2.3.1. Fully Connected Network

In a fully connected network all the nodes in the network are connected with every other node. This topology is similar to a bus approach and makes scaling difficult. A fully connected network is an expensive choice and is therefore only used for networks with a small number of resources. This network approach is consequently not so popular for NoCs.

2.3.2. Fat Tree

A fat tree has an indirect topology structure and is a better alternative than a fully connected network when it comes to size. The fat tree is one of the best topologies from a connectivity point of view. The Bandwidth does not get degraded with increasing number of cores. On the other hand, as the number of cores increases so must the size of the network. In order to sustain the bandwidth, the network size increases drastically with the number of IP-cores. The NoC will therefore take a substantial amount of the chip size. It should therefore only be used when we have a small number of IP-cores. The SPIN architecture [5, 17] uses a fat tree topology. In the SPIN fat tree every node has four sons. The father is replicated four times at any level of the tree. So four fathers offers four equivalent paths, where the shortest path is through the nearest ancestor. So as I explained before this fat tree is best on a connectivity point of view, but very complex when seen from the wiring point of view. It will therefore consume a large silicon
area and is therefore only good for small systems. The SPIN would be very well suited for a 16-terminal configuration [18] that we can see an example of in Figure 1.

![Figure 1: A 16-terminal SPIN fat tree topology](image)

### 2.3.3. 2-D Meshes and Tori

2-D meshes and tori are the most popular choices for NoCs. The mesh is the topology that most resembles the layout of a chip. These two topologies are direct multi-hop networks, were a message needs to traverse one or more nodes in order to reach its destination. A 2-D mesh scales well and does not develop the large area and complex wiring as for example a fat tree when there are a large number of nodes. The switch is also smaller in a 2-D mesh compared to a fat tree. In addition routing is easy, and the local interconnects are independent of the size of the network. As the cost of area on the chip gets increasingly more expensive year for year, it is important to make the NoC as small as possible. When several tens of cores need to be connected on chip, then a 2-D mesh would be the only cost efficient topology to use. Most NoCs today use a 2-D mesh topology. The Xpipes [6], Nostrum [12, 19], SoCBUS [20, 21, 22] and SoCIN architectures [7] all use or recommend the use of a 2-D mesh. Higher order meshes are not used because there is no need for them. The applications to be used with the NoCs communicate with a high degree of locality, and it will therefore be no need for higher order meshes. Furthermore the 2-D mesh is an ideal choice for wormhole routed networks, because the negative effects of the large internode distances are minimized [16]. A 4*4 2-D mesh topology is shown in Figure 2. The 2-D mesh torus is much similar to the normal mesh, but it has a wire connecting the first and last switches in the mesh. In a Network-on-Chip a message may pass several switches. A hop is defined as the communication action from switch to switch. So the average number of hops will be
reduced for torus networks. We will receive a lower latency, but an increase in the cost and complexity.

![A 2-D mesh topology with bi-directional links](image)

Figure 2: A 2-D mesh topology with bi-directional links

### 2.4. Routing Algorithm

The routing algorithm determines the path a message should travel. There are several different routing algorithms to choose from. The main ones are deterministic and adaptive routing.

In deterministic routing the path is chosen before the message is sent into the network. The message is not allowed to deviate from this path. This results in lightweight switching implementations. The switches become easier to develop, and deterministic routing makes it easier to detect future deadlocks in the system. It does not however utilize the network to its full potential when we have a heavy load. Each message must also carry the complete routing information. This will therefore increase the message size. XY routing is a simple example of deterministic routing. Messages going to a receiver must first travel in the X direction, and then when it reaches its column for its destination it must follow the Y direction. The use of XY routing will give us a deadlock free NoC at the price of available bandwidth. In wormhole routing we often encounter deadlocks, so this is a cheap way to get rid of them. The SoCIN [7] is an example of a
NoC architecture that uses XY routing because of its simplicity and deadlock free behaviour.

In adaptive routing a message may select an alternative route if for instance the channel is congested. Messages can choose different paths through the network each time, all depending on the state of the network. This approach increases the efficiency of the network when we have heavy traffic. The negative side of an adaptive routing strategy is that the switch gets more complicated and will therefore consume more area on the chip. Deadlocks are also harder to avoid in adaptive routing compared to deterministic. In addition a message may encounter a livelock. With livelock we mean a message that is routed in the network indefinitely, never reaching its destination. The SPIN [5, 17] architecture implements a wormhole switched adaptive network.

2.5. The Switching Technique

Packet switching and circuit switching are the two main switching techniques used in interconnection networks. Packet switching divides a message into a number of smaller units called packets. Examples of well know packet switching techniques are store and forward, virtual cut through and wormhole packet switching. In circuit switching, a connection must be set up between the source and destination before a message can be transmitted.

2.5.1. Messages, Packets and Flits

So far we have only used messages to describe the data that is sent or received in the network. There are however some distinctions of the data that is routed. A message is the complete data that needs to be transmitted, while a packet is only a part of this data. In packet switching, a message gets divided into packets. A packet contains a header, payload and a tail like we see in Figure 3. The header contains routing information for the packet, while the tail could for instance contain an error checksum (FCS) for the payload and a flag to signal the end of the packet. The total message size gets larger, but we get an increase in the network utilization. In cut through packet switching, even the packets get divided into smaller units called flits. The flit size in a NoC is often equal to the physical channel width. The channel width could for example be 8-bit or 32-bit wide. A flit would then have the same size. In this study we will not look at the effects of
dividing a message into packets. Packets and messages are therefore used interchangeably when describing the data in the network.

![Diagram of a message, packet and flits]

**Figure 3:** A message, packet and flits

### 2.5.2. Store and Forward Packet Switching

In store and forward packet switching, the switch must receive the complete packet before it can begin transmitting the packet to the next switch [23]. This means that we get a delay at each switch as the whole packet must be received before a new transaction can begin. We get however, in comparison to circuit switching, lower latency and higher network throughput. For NoCs this approach is especially expensive as it requires an enlarged die size due to the size of each buffer.

### 2.5.3. Virtual Cut Through Packet Switching

For virtual cut through packet switching we do not need to wait for the arrival of the complete packet before initiating the transaction to the next switch. This reduces the latency experienced at each switch for store and forward packet switching. Buffering is only employed, if the packet gets blocked due to contention. The buffer size in this packet switching approach is the size of a packet, so for an extreme case, where a message gets blocked at each switch, the packet switching approach gets reduced to store and forward packet switching.
2.5.4. Wormhole Switching

When we hear the name wormhole switching most of us associate it with the television series Star Trek and research into wormholes by Stephen Hawking. According to Stephen Hawking [24] a wormhole is a “thin tube of space-time connecting distant regions of the Universe.” This wormhole connects regions of space for a brief amount of time. In parallel computing and NoC the wormhole routing technique was first introduced by Dally and Seitz [25]. The function of the wormhole routing technique is similar to the definition of a wormhole from Hawking. A channel between to nodes in a network is opened for a short amount of time. In this time period a packet can reach its destination before the channel is torn down.

This is however not entirely correct, as the channel acts more like a worm. It worms its way through the network, and the head can go faster then rest of the worm. The worm will then extend itself, and the maximum size would be from the source to its destination. This is however only possible when the tail or parts of the payload gets blocked due to contention. The maximum length of the worm is also possible if the message size in flits is larger then the number of hops.

Wormhole switching [26, 27] is a cut through packet switching approach and the favourite switching technique for NoCs. The switching technique is used for the network simulator that was developed in this master thesis. Consequently it will be explained in much more detail than the other switching techniques. Wormhole switching produces reduced buffer sizes and low latency. On the negative side we get a higher congestion in the network and a larger risk for deadlocks.

A packet is transmitted in units of flits. In wormhole switching three different kinds of flits are used. The header flit contains routing information, virtual channel identifier (VCID) and priority. The payload of the packet is divided into payload flits that include a VCID but no other routing information. A tail flit could include a small Cyclic Redundancy Check (CRC) and a tail flag.
The header is used to set up the routing path for the packet. When a header enters a switch it will have a buffer associated with it. This buffer is occupied by the packet until the tail enters the buffer and releases it. Packets can therefore not be interleaved in the buffers of the switch. A flit advances to the next switch as soon as it arrives at the switch. If a header flit gets blocked then all the trailing flits will advance forward until all the cells of the virtual channels have been occupied. All the flits will then remain in the buffers they occupy in a pipeline fashion. The packet that was blocked will only advance when the system resources that it needed becomes available.

Wormhole routing networks are frequently open to deadlocks. When a message gets blocked due to contention, a deadlock may occur. Figure 5 presents an example where four messages in the wormhole routed mesh network block each other. Here each message requests a buffer hold by another message. None of the messages in this mesh will be able to advance. For deterministic routing a deadlock can be avoided with the use of XY routing.
As wormhole switched networks are often susceptible to deadlocks and frequent blocking of messages, J. Dally [28] comprised a way to get rid of deadlocks and to reduce the effect of blocking. Virtual channels (VC) were introduced. A virtual channel comprise of a buffer and its associated state information. Instead of having only a single buffer associated with the input gate, we can have multiple virtual channel buffers connected to each of the input gates. The VCID is used to distinguish each virtual channel. Every packet in the network will be able to obtain a VC in the next router. A routing arbitration (RA) algorithm decides which virtual channel is allowed to be scheduled on the output. VCs can be used to avoid deadlocks. In order to avoid the deadlocks we need as many VCs for an input gate as the maximum number of messages passing the link to that input gate. For NoCs, VCs are only used in order to reduce the effects of blocking. Using VCs to avoid deadlocks would be too expensive.

When there is only one buffer per input gate, only a header flit may be blocked. All the proceeding flits get consequently blocked in place. With the use of multiple buffers, a payload or tail flit may also experience contention and blocking. When blocked, all the flits following behind it will remain in the buffers they occupy. The flits found before the blocked flit may advance as long as there are system resources available.

The switch is the most important building block in the network. The size of the switch is very important when it comes to the size of the network. It is therefore important that the buffer sizes in the switches are small so that we get a light weighted switch. A network using wormhole flow control makes it possible to have switches with very small buffer sizes. Wormhole routing is a very common feature for NoCs. SoCIN [7], Proteo [8, 29], Xpipes [6] and SPIN [5, 17] NoC architectures all implement wormhole routing.

2.5.5. Circuit Switching

Most NoCs today use packet switching but there are some researchers that have found it better to use circuit switching. Two of these researchers are the developers of the SoCBUS [20, 21, 22]. In circuit switching a route is locked between the sender and receiver. This route will be locked the whole time as the transfer goes on. When the transaction is finished the route must be cancelled and the resources can then be made available to others. The resources that would be made available would be the buffers and the links that were reserved for the session. Circuit switching can guarantee latency
and available bandwidth. But the bandwidth and latency can only be guaranteed for the transmission lines that have been set up. Other data that needs to be sent have to wait for the lines and the buffers to be freed.

In order to avoid deadlocks most packet switches takes use of buffers or virtual channels. The switches in packet switching therefore become more complex than in circuit switching. Deadlock issues can easily be avoided in circuit switching. This is because when you set up a transaction you can either succeed in setting up the line or fail to set up the line. No deadlocks can therefore be found in the network. The latency can be guaranteed in circuit switching because the latency is only dependent on the distance it travels. In packet switching, more factors come in to play like traffic in the network. We see that circuit switching is good for long-lived connections with a high link load and therefore not suited for random traffic. The setting up and disconnecting of a line comes with a time overhead. Setting up and cancelling the connections at a high rate will give no benefit when using circuit switching. Circuit switching would only benefit hard real-time applications.

2.6. The Switch Design

The most important component in the NoC is the switch. Due to the fact that in a direct network each node is connected to a switch it is important to keep the switches as lightweight as possible. The build up of the switch is of course related to the topology, routing algorithm and switching technique but also arbitration algorithm, flow control and channel buffers are important issues in a design [30]. A detailed description will be presented in chapter 5. This part gives an overview over switch design aspects.

2.6.1. Buffers and Crossbar

The most important building block of a switch is the crossbar, buffers and control logic as shown in Figure 6. The buffers can either be connected to the input as illustrated in the figure, it can be connected to the output, a centralized buffer pool can be applied or we can even have no buffers at all. The buffer implementation effect the amount of blocking a packet may come across. A popular buffer design for wormhole switched networks is to use virtual channels where each channel has an independent FIFO. The connectivity between the output and input ports can be implemented with a crossbar. The optimal
crossbar design is fully connected and non-blocking. This however increases the complexity of the switch and therefore also the cost. In a fully connected crossbar each input port is connected to every output port.

![Figure 6: A simple input buffered switch design]

2.6.2. Arbitration Algorithm

With an arbitration algorithm we mean the allocation of the output port for the flits ready to proceed to the next switch. When two packets compete for the output port, the conflict is resolved with the use of an arbitration algorithm. These contentions can be solved with priority, round robin, first come first serve and oldest-first scheduling. A prioritized physical channel scheduling is presented in [31]. This scheme uses a priority scheme with round robin for messages with the same priorities. Another scheme is a prioritized maximal matching algorithm [32]. A bipartite graph is constructed in order to find the maximal matching of the messages to each output gate. Network utilization would be increased with the use of this algorithm. The switch complexity on the other hand would increase.

2.6.3. Flow Control

The flow control deals with the allocation of system resources. In link-level flow control the aim is to provide a mechanism to control the flow from an output port over a link to an input port. At the input port we do not know if the switch is able to receive the flit. The input port buffer may already be filled up. The destination switch needs to give a feedback to the source of the status of the input buffers. A common approach in asynchronous designs is to use request and acknowledge signals between the source
and destination. Separate links for flow control signals are needed. The SoCIN [7] has implemented this flow control scheme. A related approach is to use credit-based flow control. Credits are passed between the source and destination so that the source is able to keep control of the free buffer space at the destination.
Chapter 3
The Feasibility Tests

3.1. General Presentation

Wormhole routing has become a popular switching technique for NoCs because it produces reduced buffer sizes and low latency. However due to the fact that messages in wormhole routed networks are frequently subjected to contention, it is hard to guarantee timely delivery of messages. Several feasibility tests have been proposed for wormhole switched networks in the recent years.

The on-chip network communication can be related to predictable events [33]. In a SoC design the developer will port a specific application to the NoC resources. The traffic pattern of this application can easily be predicted as the application is known to the designer. Information will most likely be sent with a certain period and length. This traffic information can be used to calculate the feasibility of a NoC mapping. Three feasibility tests that can be used for this purpose is the lumped link, blocking dependency graph and the contention tree models. All the three feasibility tests need this form of static behaviour in order for them to predict if a message stream is feasible.

In order for the computation time of a message to be predicted we need a deterministic routing scheme. An adaptive routing approach would introduce to much unpredictability for the feasibility tests. In addition, finding the communication contention in the network would be complex and intricate for an adaptive routing approach. Deadlocks must also be avoided by using for example X-Y routing for meshes.

A priority scheme must be decided upon. The most common priority scheme in uni-processor systems is Rate Monotonic Scheduling (RMS) [34]. In RMS the Rate Monotonic Algorithm (RMA) gives a message a priority according to its period. The
message with the shortest period will get the highest priority. It is not the task of this study to find and compare different priority schemes for the system developed. RMS however proved to be a mediocre choice for the mesh network. In a NoC a message passes several routers before reaching its destination. The Kumar et. al. [2] paper evaluated different priority schemes. Simulations showed that a priority scheme that was based on the amount of blocking a message can withstand and the number of hops was the best to use.

The H3 algorithm:

\[ H3 = \frac{(\text{deadline} - \text{size})}{\ln(e + \text{hops} - 1)}. \]

The message with the lowest H3 will get appointed the highest priority. This priority scheme proved much more suitable than RMS.

Situations can occur in wormhole routed networks where a lower order priority message can block a higher priority message. This priority inversion can lead to a higher latency than expected. This situation can occur if the number of VCs on an input gate is lower than the number of different contending message streams that use this input gate. Assume that all the switches in Figure 7 only have a single VC per input gate. The message with the lowest message number has the highest priority. We can further assume that the message size of M_9 is much smaller than the message size of M_12. All the resources start to transmit at time zero. After M_9 has finished its transmission, M_12 will advance onto switch 0 only to find that the VC is occupied by M_15. M_12 can not advance until M_15 frees up the VC. In simple wormhole routed networks we do not have flit-level pre-emption. A NoC must be uncomplicated in order to minimize the die size. A flit-level pre-emption technique would be too complicated and expensive for it to be implemented on the chip. All the feasibility tests presented in this master thesis assume as many VCs per link as messages per link. So a feasibility test for Figure 7 would assume that we have at least two VCs per link. As a result all the feasibility tests ignore priority inversion.
After a mapping in the system design phase of a NoC implementation, the designer will know the communication needs for each node. All the message streams in the network will form a message stream set for the complete system ($S_S$). All the messages in the message stream ($M_i$) will be defined with the same routing path, priority of importance ($i$), message size in flits ($f$), period ($p$), deadline ($d$) and end to end transmission time ($T$). With end to end transmission time we mean the response time or computation time. The time it takes for a message to travel from the source to the receiver without any contention in the network. The routing path defines only the source ($s$) and receiver node ($r$). The messages in the $M_i$ are sent into the network continuously with the defined period. A message stream is defined with a seven-tuple $M_i = \{s, r, i, f, p, d, T\}$. The definitions message and message stream are used interchangeable in this master thesis.

In order to estimate the latency of a message we need to associate a priority with each message. Each message is given a priority according to a given priority scheme. In the $S_S$ the messages are sorted in descending priority order. The first message in this set ($M_0$) will be given highest priority. According to priority each message will form contending messages sets ($S_C$). $M_i$ contends with another message in the $S_S$ if the other message delays the message by using some of the same system resources. The message will only contend with messages higher up in $S_S$. A message ($M_0$) that delays the message in this way will directly block the $M_i$. We must however also consider indirect contention messages ($M_\text{II}$). In a first level indirect contention, $M_\text{II}$ will block $M_0$, so that $M_i$ is indirectly blocked by the message $M_\text{II}$. $M_\text{II}$ does not share a link with $M_i$ but
indirectly blocks through $M_{Di}$. In the same manner we can have several levels of indirect contention.

As an example we consider that an 8*8 Mesh has $n$ messages: $S_S = \{M_0, M_1, M_2, M_3, M_4, \ldots M_n\}$. Figure 8 shows an excerpt of this mesh. In this section of the mesh we find 6 diverse messages. Table 1 shows the characteristics of each message present. The message sets are $S_{C0} = \{\emptyset\}$, $S_{C1} = \{MD_0\}$, $S_{C2} = \{\emptyset\}$, $S_{C3} = \{MD_1, MI_0\}$, $S_{C4} = \{MD_2\}$ and $S_5 = \{MD_3, MD_4, MI_2, MI_1, MI_0\}$.

![Figure 8: Example of contending messages](image)

<table>
<thead>
<tr>
<th>Message</th>
<th>Priority</th>
<th>Source</th>
<th>Receiver</th>
<th>Size</th>
<th>Period</th>
<th>Deadline</th>
<th>Computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>29</td>
<td>29</td>
<td>30</td>
<td>5</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>13</td>
<td>31</td>
<td>4</td>
<td>4</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>15</td>
<td>23</td>
<td>3</td>
<td>3</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>23</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>40</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1: Attributes of contending message example
Table 2 summarizes the properties of the three feasibility tests that will be presented next. We see that the distinction between the three tests lay in the consideration of different contention. The next sub-chapters will present the models and explain the differences.

<table>
<thead>
<tr>
<th>Model</th>
<th>Priority Inversion</th>
<th>Direct Contention</th>
<th>Indirect Contention</th>
<th>Disjoint Concurrent Contention</th>
<th>Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lumped Link</td>
<td>Ignored</td>
<td>Implemented</td>
<td>Not considered</td>
<td>Not considered</td>
<td>Never Occurs</td>
</tr>
<tr>
<td>Blocking</td>
<td>Ignored</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Not considered</td>
<td>Never Occurs</td>
</tr>
<tr>
<td>Dependency</td>
<td>Ignored</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Never Occurs</td>
</tr>
<tr>
<td>Contention Tree</td>
<td>Ignored</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Never Occurs</td>
</tr>
</tbody>
</table>

Table 2: The features of the feasibility models

3.2. The Lumped Link Model

Harry and Özgüner [2, 3] have presented an offline feasibility test for real-time wormhole routed communication where the feasibility of a message stream is found after all contending messages are lumped into one path.

In the lumped link model, all indirect and direct messages that contend with one another are lumped onto the same link. The indirect messages will therefore be treated in the same way as direct messages. Since the lumped link model does not make any distinction between direct and indirect messages after they have been lumped together, we can assume that the model is pessimistic. Simulation results in chapter 6 will confirm this assumption.

Since the lumped link model is pessimistic of nature we only include direct contention and one level of indirect contention in the message set of contending messages. A message $M$ with message number $n$ can only be blocked by messages from 0 to $(n - 1)$. The message $M_n$ will contend with all higher priority messages that contend with the message either directly or indirectly at level 1.
In order find the worst case latency $L(t)$ of a message $M_i$ we implement the equation

$$t = L(t) = T_i + \sum_{j=1}^{i-1} T_j \cdot \left\lfloor \frac{t}{p_j} \right\rfloor.$$

The sum $L(t)$ is the computation time of $M_i$ in addition to all the higher priority messages that contend with the message that can occur up to time $t$. If the $L$ of $M_i$ is smaller than the deadline of that message then the message is feasible. $T_j$ is described as the end to end transmission time required by a contending message and $p_j$ is the period of that message. A message with a higher priority and a lower period can block the message $M_i$ several times.

In order to solve the equation we need to find the $t$. $t$ is the latency of the last iteration. The equation converges when the latency of the current iteration is equal to the latency of the previous iteration. A fixed-point iteration technique can help us in finding the $t$ where the equation converges. Yen and Wolf [35] cover this fixed-point iteration technique in detail.

The first thing we carry out in a fixed point iteration technique is to calculate the sum of transmission time divided by the period for all the elements in the contending message set. This is to make sure that this sum is below one. If this test does not hold up then we have exceeded the capability of the link to schedule the last message onto the link.

If the link capacity is not breached then we need some more calculations. We start by setting the first $t$ in the iteration equal to the computation time of the message being tested divided by 1 minus the sum we got previously. Next we get the computation time and period of the first message, and then we collect the sum of the first $t$ we found divided with the period of the first message in $S_{ci}$. After that we multiply this with the computation time of the first message. This is done for all the messages in the set and all the results are added together. At last we add the latency of the message being tested and we check if $t$ is smaller than this sum. If it is smaller then we need to run one more time until $t$ is larger or equal to the sum $L(t)$. For the next runs run we set the $t$
equal to the $L(t)$. The last $L(t)$ we found is the worst case response time of the message being tested. The feasibility is established by comparing $L(t)$ with the deadline of the message.

**Lumped link model algorithm for message $M_i$**

1. For each $M_j$ in $S_{ci}$
2.   $st = st + T_j/p_j$
3.  iterationNumber = 0
4. if ($st < 1$)
5.   {
6.     $t = T_i/(1-st)$
7.     do {
8.       if (iterationNumber != 0) $t = Lt$
9.       for each $M_j$ in $S_{ci}$
10.          $stold = stold + T_j^*(t/p_j)$
11.         $Lt = T_i + stold$
12.     } while ($t < Lt$)
13.    if ($Lt < p_i$) $M_i$ = Feasible
14.    else $M_i$ = Infeasible
15. } else $M_i$ = Infeasible

As an example for the lumped link model we can look at the message $M_5$ and the contending messages $M_4$ and $M_2$ in Figure 9.

The messages are represented with $M_2 = \{13, 15, 3, 15, 15, 15, 4\}$, $M_4 = \{14, 7, 1, 5, 30, 30, 6\}$ and $M_5 = \{31, 7, 0, 3, 40, 40, 5\}$. This example is taken from figure 8 assuming $M_3$ is infeasible and therefore not in contention with $M_5$. Implementing the fixed point iteration technique on $M_5$ shows us that the $M_5$ is feasible. The initial $t$ has a value of 10. The fixed point iteration technique only takes a single iteration. The $stold$ values are 6 and 10. Adding $T_5$ to the last $stold$ produces an $L(t)$ of 15 which is larger than $t$. The maximum latency of $M_5$ is thus 15. This can be confirmed by applying messages $M_2$, $M_4$ and $M_5$ into a timing diagram with the number of slots equal to the deadline for $M_5$. The

![Figure 9: The first contention graph example for the lumped link model](image)
timing diagram reveals $M_5$ fitting in between slots 9 and 15 and therefore having a latency of 15.

As a further example we look into the feasibility of $M_3 = \{30, 23, 2, 3, 25, 25, 4\}$ using the lumped link model. The $S_{C3}$ contains messages $M_0 = \{29, 30, 5, 4, 10, 10, 4\}$ and $M_1 = \{29, 31, 4, 4, 15, 15, 4\}$. Employing the lumped link model on $M_3$ reveals that the message is infeasible.

The outcome of the fixed point iteration technique gave us a maximum latency of 26. We need two iterations where the $L(t)$ values are 22 and 26 respectively. We can again confirm this result by adding $M_0$, $M_1$ and $M_3$ into a timing diagram. We see that $M_3$ can only be scheduled in slots 9, 14 and 24. The last flit is scheduled outside the deadline at slot number 26. The deadline of $M_3$ is 25 and we need 26 cycles in order to schedule the message. The message stream is infeasible.
3.3. The Blocking Dependency Graph

The algorithm of the blocking dependency graph model [4] consists of generating a direct initial timing diagram and then modifying this diagram by deleting all the indirect time slots that might not contend. The BDG algorithm can free up indirect time slots that do not block and therefore increase the total number of feasible messages for an application that was mapped to system resources. If we go back to look at the feasibility of M₃ = {30, 23, 2, 3, 25, 25, 4} from Figure 8 we see that this message is actually feasible. This is due to the freed indirect time slots that do not contend with M₃.

In the BDG model a graph is built that shows the direct and indirect level 1 blocking of a message Mi. The SCᵢ is found and the graph gets built based on the direct and indirect contending messages in the SCᵢ. In the algorithm the blocking dependency graph of SCᵢ is implemented by an adjacency matrix. In this study we use a modified adjacency matrix that includes distinctions of direct and indirect contention. We again study the blocking effect of M₀ = {29, 30, 5, 4, 10, 10, 4} and M₁ = {29, 31, 4, 4, 15, 15, 5} on M₃.

![Figure 13: The contention graph example for the blocking dependency graph model](image)

The modified adjacency matrix that was implemented in the simulation model to evaluate the feasibility models is illustrated in Table 3. In the matrix an indirect contention is marked by the value 2. A direct contending message is set to 1. An important point regarding the adjacency matrix is that if a message is both contending directly and indirectly then the message will be marked according to the direct contention. We see that M₃ has two contending messages, M₀ and M₁. M₁ is marked as direct while M₀ is marked as indirect. M₁ has a direct message contention with M₀ as a father.
The next step of the BDG algorithm is by generating initial timing diagrams based on the adjacency matrix. The slots in the timing diagram can be set to either allocated (A), waiting (W), busy (B) or free (F). We start with the message with the highest priority and we build its initial timing diagram. For $M_0$ we have no contention so the diagram contains the scheduled slots of $M_0$ only up to the deadline of $M_3$. All the utilized slots are marked allocated while the slots that are not used are marked free.

![Figure 14: Initial timing diagram of message 0 from BDG example](image1)

An initial timing diagram for $M_1$ must also be constructed. This timing diagram will be derived from the schedules of $M_1$ and $M_0$.

![Figure 15: Initial timing diagram of message 1 from BDG example](image2)
M₁ is obstructed by M₀ in the first four time slots and the slots are therefore marked as waiting. All the slots that are used by M₀, but that do not interfere with the schedule of M₁, are marked busy.

Finally we can look at the result of M₃. We observe that this timing diagram is the same as the one we found for the lumped link model. Now we need to consider the indirect contention before we can look at the final timing diagram.

At this time we need to go back to the initial timing diagrams and check every indirect message. The allocated and waiting time slots of the indirect messages are checked against the time slots for all of its children. That is, an indirect message is checked against all its direct children. If we have an allocated or waiting indirect time slot and this time slot is busy or free for all its children then we can mark it as free.

M₀ is contending indirectly so we need to consider all the allocated and waiting time slots in this diagram. These time slots are checked against the time slots in its child M₁. First we start at slot 0 of M₀ and check if we have a slot marked allocated or waiting. This slot is checked alongside slot 0 of M₁. We see that it is marked waiting so we are not able to free it. There are no changes made for slots 0 to 3. The next allocated slot is slot number 10 in M₀. Slot 10 in M₁ is marked as busy so we can set the slot number 10 to free in both diagrams. This is repeated for slots 11-13 and 20-23. In the case of finding a waiting slot in the indirect diagram that can be freed, then we need to schedule the slots in the direct diagram again to conform to the indirect diagram.
Figure 17: Improved timing diagram of message 0 from BDG example

Figure 18: Improved timing diagram of message 1 from BDG example

In Figure 17 and 18 we can see the updated and final diagrams of M₀ and M₁. Finally we can update the diagram for M₃ and compare the total number of free slots against the computation time of M₃. The diagram contains 11 free slots which is larger than T₃. We have proven that the message M₃ is feasible when we consider indirect contention.

Figure 19: Final timing diagram of message 3 from BDG example

3.4. The Contention Tree Model

A NoC based feasibility test presented in [9] uses a contention tree in order to analyse the feasibility of a message. It considers not only direct and indirect contention but also disjoint concurrent contention. Only the real-time feasibility analysis will be a subject of evaluation in this master thesis.
After attaining all the contention in the network, a contention tree is developed that reflects the entire contention in the network. In contrast to the BDG model where a graph was implemented for each message, the CT model only develops one graph for the complete system. There are no boundaries on the levels of indirect contention. As a result feasibility tests on the messages would include direct and all the levels of indirect contention. This causes the model to be more pessimistic.

The contention tree of Figure 8 is shown in Figure 20 if we assume that M₀-M₅ are the only messages in the system. Each node contains a list \( L_i \) of its parents. The \( L_5 \) of M₅ contains both M₄ and M₃. \( L_3 \) contains M₁ and the parent of M₁ is M₀. In this way we can easily form the contention tree with a global indirect message level. Meaning there are no limits on the amount of indirect contentions for a message Mᵢ.

![Contestion Tree example for the CT model](image)

If we were to implement a lumped link or blocking dependency graph feasibility test on M₅ we would conclude that the message was infeasible. These tests do not however look at the disjoint concurrent contention. The two branches M₂-M₄ and M₀-M₃ can be scheduled concurrently. The contention tree model consists of a model that also takes disjoint concurrent messages into account. A timing diagram of M₄ would contain M₂ while the timing diagram of M₃ contains M₀ and M₁. In the CT model, indirect contention is evaluated together with the direct contention. This is done by setting all the slots that are left until a new period begins free after all the computation time of the message being tested has been scheduled. In Figure 21 we see the timing diagram of M₄ that is needed for M₅. M₅ has a deadline at time 40 so we only need to investigate the first 40
slots. M4 must wait for M2 in order to be allowed to schedule. The non-contending indirect message contention has already been removed from the timing diagrams.

![Figure 21: Timing diagram of message 4 for the CT example](image)

Figure 22 shows the 40 first slots of the timing diagram of message M3. M0 and M1 only contend with M3 at the beginning. In the second schedule of M3 there is no contention by M0 and M1.

![Figure 22: Timing diagram of message 3 for the CT example](image)

We know that M4 and M3 are disjoint so M5 can schedule the slots of M4 and M3 on top of each other. In Figure 23 we see that a single slot can be scheduled by two messages if they are disjoint. The timing diagram shows that slots 0-9 can be scheduled disjoint concurrently. So in comparison with the blocking dependency graph, 10 extra slots have been freed. This makes it possible for M5 to be scheduled in the time slots 13-17. Note that all the slots after the scheduling of M5 are actually free if another message would have M5 as a father.

![Figure 23: Final timing diagram for message 5 in the CT example](image)
Chapter 4

The Network Simulation Tool

Optimized Network Engineering Tools (OPNET) and Objective Modular Network Testbed in C++ (OMNeT++) are two discrete-event simulation tools used for simulating networks. OPNET is a commercial tool and the market leader capable of simulating large communication networks with detailed protocol modelling and performance analysis [36]. While OPNET is the market leader, OMNeT++ [37] is becoming a popular choice for a non-commercial networking simulation tool. The main advantage of OPNET over OMNeT++ is the large comprehensive library of detailed protocol and application models and network devices. OMNeT++ is not as powerful as OPNET, but it benefits by being public-source and free for academic and non-commercial purposes. The wormhole switch-based network simulator was constructed using OMNeT++ as it is public-source, flexible and debugging friendly.

OMNeT++ is an object-oriented modular discrete event simulator. The development of OMNeT++ started in 1992 by Ákos Kun and András Varga. András Varga is still the main contributor to OMNeT++ and he has developed most of the OMNeT++ simulator. This chapter will give a brief overview over the most important topics of the OMNeT++ simulator that the reader needs to know for the following chapters. It is not intended to be used as a reference guide or manual to OMNeT++. The OMNeT++ [37] user manual is comprehensive and simple to read. The interested reader is therefore advised to look at the manual and the OMNeT++ webpage [38] in order to get better acquainted with OMNeT++. This master thesis used OMNeT++ 2.3. A newer version has now been released and some of the issues explained below may be different in the new version.

OMNeT++ is a discrete-event simulation tool which means a system whose dynamics is characterized by the occurrences at discrete points in time. In order for the simulator to know which event it should schedule next, it needs to hold the future events in a Future
Event Set (FES). This list will hold all pending events. With pending events we mean an event generated but not yet applied on the simulation [39]. Each event in the FES contains a timestamp that lets the simulator know when the event should be invoked on the system.

4.1. NED and C++

The simulation model that you develop in OMNeT++ is built up by means of modules. OMNeT++ utilizes a textual topology language (NED Language) when specifying the modular description of the network. This NED Language is employed by the developer to describe the network. The language consists of module and channel descriptions. The channels are used to connect two modules together in order for them to communicate. Like in a network, modules communicate with each other by sending messages through channels. The NED language describes the layout of the modules, while C++-files describe the inner workings of them. All the functionality of the network is described in C++ code. The developer then only needs to learn the NED language in order to construct the simulator, given that he is familiar with C++.

As a basic example we could look at a simple NoC. A node in this network topology would be described with a module and the link would be described by a channel. We can therefore easily construct a network topology like a mesh by connecting modules with other modules with the use of channels. The network topology can be assembled with the NED language. A network can be comprised of compound modules and simple modules. A simple module does not have any other sub-modules and the function or process of the simple module must be described by C++-files. A compound module can contain other nested compound modules or simple modules. So the designer determines the level of the module structure.

Another concrete example of nested modules can be seen in Figure 24. We start by describing the topology in the figure using the NED language. In this example we have four modules to illustrate four switches connected by bi-directional links in a 2*2 mesh. If we go down one level and look inside the compound module of switch 2, we can see the inner workings of the switch. Switch 2 contains 4 compound modules or sub-modules.
These 4 compound modules describe in more detail the functionality of the switch. The modules still communicate by sending messages to each other.

At the bottom level inside the input buffers compound module in Figure 25, we see a receiver and three virtual channels. These modules are simple modules and are also described by the NED language. The NED language does not however describe how the simple modules functions. C++-files are needed to describe the functionality of the receiver and the virtual channels. As an example, we could think of what kind of buffer we should use for the virtual channels? The C++-file could for example describe a FIFO.
4.2. Simple Modules

The simple modules describe the behaviour of the network. In order for this to happen, you have to overwrite different virtual functions. The virtual functions used in the wormhole switch-based network simulator are: initialize, handleMessage and finish. The initialize function is used to setup all the modules. It is called by OMNeT++ at start up. Finish is called at the end of a simulation run and writes out statistic information. The functionality of the module is described in the handleMessage function. This function is called each time the module receives an input at one of its gates. Self-messages can be sent to the module in order to trigger periodic events. As an example we could assume that a module needs to activate itself each cycle. A new self-message can then be scheduled each time the function is run. The module would then send a message to itself each cycle and thereby activating itself.

4.3. The Messages

Information is sent and modules are activated by the use of messages. Messages are defined in its own *.msg file. A message has several useful properties defined by OMNeT++. Name, length, message kind, and time stamp are some important attributes. In addition the programmer can add their own properties to the message. An important attribute is length. The length keyword defines the length of the message and is later used by OMNeT++ to compute transmission delay over a link. The user can also define different message types and distinguish them by setting the message kind attribute. A module can then activate different functions depending on the message kind on the arrival of a message.

4.4. The Graphical User Interfaces

Two user interfaces are defined for the OMNeT++ simulator. Tkenv is a graphical user interface, while the Cmdenv is a command line user interface. Tracing, debugging, and simulation execution can be carried out in Tkenv. Cmdenv is faster than Tkenv and therefore used when we want output data from several simulations that have a long simulation time. The implementation of either Cmdenv or Tkenv into the executable can be done easily by changing a single line in the make file.
4.5. The Simulation Files and Execution

As can be seen in Figure 26 there are several files that are linked together to make up the final simulation program. The user of OMNeT++ only needs to write the *.ned files and the *.cc files. A NED compiler transforms the NED-files into C++ code. The user defined NED-files and C++-files are linked together with a simulation kernel library and a user interface library. An executable is built that is controlled by a configuration file. Simulation data can be written out to data files during or after a simulation run.

Figure 26: The files, compilers and linker needed for an OMNeT++ simulation
Chapter 5

The Network Simulator Developed

This chapter builds upon the definitions presented in chapter 2. It presents the network simulator that was developed for the study of the real-time performance models. We have previously introduced the OMNeT++ simulation system. The network simulator was built on top of OMNeT++.

The most time consuming task in this master thesis was the development of a wormhole switched network simulator that was able to evaluate real-time messages. The switch developed in the interconnection network contains a lane allocator, switch allocator and crossbar, forward allocator, input gates, receiver, generator and upload logic which is illustrated in Figure 27.

Figure 27: A screenshot from OMNeT++ that shows us the switch developed
The switch design in the simulator is based upon Z. Lu’s [40, 41] flit ejection and flit admission papers. The basis of this study is to evaluate the real-time performance models. Not to build a switch that is optimal for cost and complexity. Flit admission and flit ejection is consequently built based upon the most ideal solutions.

5.1 Internal Switch Processes and Lane States

The internal working of the switch is quite equal to Peh and Dally’s [42] presentation of an accurate modelled pipelined router. A simulated second in the OmNeT++ simulator is set equal to one cycle for the simulations of the wormhole-routed NoC. The simulation second is then divided into smaller time units that each initiates the beginning of a pipelined stage. The simulator can be set to be a one-cycled switch even though the inner workings of the switch work in pipeline stages. As we know a module can be initiated with the transmission of a self message. Modules in the network will send messages to themselves at defined time units and mark the beginning of a new pipeline stage. The switch is divided into a routing and message generation stage, lane allocation stage, forward lane allocator stage and a switch allocation and arbitration stage.

![Figure 28: The internal stages of the switch](image)

A packet that enters the switch developed in this study must proceed through a certain amount of states before proceeding to the next switch. The states defined in the OmNeT++ switch are: routing, lane allocation, lane allocation 2, scheduling, switch arbitration and reception.

![Figure 29: The internal states a message must pass through in the switch](image)
A simulated second begins with the routing and message generation stage. On the reception of a header flit on the input gate, the flit is sent to a virtual channel lane corresponding to the VCID in its header. Each input gate can receive one flit per cycle. At the same time a message might be generated and sent to a flit uploading buffer if one is available. All the new header flits that entered into a lane at this time will get their routing information interpreted. During this time the state of the lane will be set to a routing state. After the routing has finished, the lanes receive the lane allocation state. Flits that do not need to be scheduled on the output ports will be set to a reception state. 0.40 simulated seconds later the lane allocator sends a self message to itself and the simulator enters the lane allocation stage. In this stage the lane allocator looks at the state of each lane. Lanes in the lane allocation state are eligible for a lane association. Requests from all the lanes that are in the lane allocation state gets sent to the forward lane allocator. The state for these messages changes to a temporary lane allocation state. The first reception of a request initiates the forward lane allocator. The lane allocator has sent all the requests in prioritized order. So the first request has the highest order and a lane association is therefore attempted on this request. If the request succeeds then the state goes into scheduling. A failure leads to the state going back into lane allocation. This is then attempted for the next prioritized messages until all the requests have been processed. At simulation time plus 0.60 simulation seconds, a self message initiates the forward lane allocator and we are in the forward lane allocator stage. Now all the scheduling states are found and the buffer spaces of the lanes in the next switches are checked. If a lane is in the scheduling state and the associated next buffer is able to receive a flit then the lane enters into the switch arbitration state. The last stage, switch allocation and arbitration stage, begins 0.40 seconds later just before the beginning of the routing and generation stage. The switch allocator gets run and looks for lanes that are in the switch arbitration state. The ones with the highest priority get scheduled on the outputs.

If a lane is granted switch arbitration, and it has remaining flits of the same package in the lane, then the state will go back to the scheduling stat after the flit has been sent. If the lane is empty, it will go into an idle state.
5.2 The XY Routing Algorithm

The routing algorithm that was chosen had to be simple and avoid deadlocks. A simple and cheap routing scheme is the dimension-ordered XY routing. A packet travelling must first travel in the X direction before going in the Y direction. A packet is then only allowed to turn one time from the X to Y direction. In this way we avoid deadlocks but pay for it in bandwidth. The header flit must contain XY routing information. The routing header is constructed at the source switch before it is scheduled on the output gate. The XY algorithm calculates the X and Y direction the packet must travel in depending on the position of the switch. One bit will be used to define if the flit needs to go in the west or east direction (xDir). Another bit is implemented for the north or south direction (yDir). Then we need two fields to define the number of hops the packet needs to travel in each direction (xMod and yMod). The number of links the packet needs to pass is counted in each direction. So the size of these two fields is dependent on the size of the mesh.

As an example we assume we have a packet going from switch 0 to switch 6 in Figure 31. The xDir routing fields would be the east direction and the yDir would be the south direction. The xMod and yMod would be 2 and 1 respectively. The header flit would begin to travel in the east direction. When it passes the first link its xMod counter gets decremented to 1. For the second pass the xMod field is set to zero. Switch 2 notices that we have an xMod of zero and it schedules the message in the Y direction. In this case the Y direction is south. The flit travels south and its yMod gets decremented to zero. The last switch notices that both the xMod and yMod are zero and sends the flit to the receiver.
5.3 The Switch Arbitration and Lane Allocation Algorithm

A lane is associated to another lane on a packet by packet level. The packet retains the associated lane until the tail flit enters the FIFO. A contention for the same lane is solved by priority. The one with the highest priority will get the lane association. The switch arbitration is based on a flit by flit basis and contentions are solved with a combination of priority and oldest-first scheduling. In the case of two contenders having the same priority, the one with the oldest timestamp will be chosen. This creates a fair switch arbitration scheme that also preserves the order of the packets.

5.4 Credit-Based Flow Control

The switch uses a credit based flow control. Credits are passed on separate links in order to keep track of the availability of scheduling a flit on an output gate. A forward allocator keeps track of the number of free buffer space and lanes. A switch will only need to keep track of the virtual channel lanes in the next routers that can be associated with its lanes.

The number of channels a switch needs to store information for is given by:

\[ TC = VC \times G \]
where VC is the number of virtual channels and G is the total number of active output gates.

Figure 32 shows a screenshot from the forward allocator array during a simulation and an illustration of the lanes in the next router that we need information for. There is only one virtual channel and all the gates are active. The switch only needs to keep a control on four virtual channel lanes, one in each switch. In the example illustrated the east and north switch have been occupied, and the north switch has only space for one more flit.

![Diagram of network switch and virtual channel lanes](image)

**Figure 32: A screenshot from the forward allocator with illustrated associated buffers**

At the departure of a flit from a virtual channel lane or flit uploading buffer, credits are sent to the forward lane arrays that need to be updated. Credits can be sent to the local forward lane array, the previous forward lane array or both. This depends on where the flit is going and where it came from. At start up time all the virtual channel lanes are empty. Consequently all the forward lane arrays get initialized to free and to the number of cells of a virtual channel. The number of free cells in the VC is then decremented or
incremented by the passing of messages. The decrementing credits get sent to the local fwd lane array while the incrementing credits get sent to the previous fwd allocator.

We have four possibilities:

- Flit leaves upload flit buffer:
  - Send credit and decrement counter of the local forward lane array

- Flit leaves input gate and is scheduled on output gate:
  - Send credit and increment counter of the previous forward lane array
  - Send credit and decrement counter of the local forward lane array

- Flit leaves input gate and is transmitted to the receiver:
  - Send credit and increment counter of the previous forward lane array

Figure 33 illustrates a more detailed picture of the connections implemented for credit passing. Here the forward lane array in the east switch needs to have control on the number of cells in the VCs free in its neighbours. When a flit is sent out on the output gate a credit gets sent to the local forward lane array in the east switch decrementing its counter for the east gate. A second credit is sent back to the west switch saying that a cell has been freed. If a flit gets sent from the upload buffer then only the local forward lane array needs to be implemented.

As discussed earlier a virtual channel lane gets another lane associated with it in a neighbouring switch. The forward lane array also keeps a control on the number of free lanes in the next switches. On lane associations the forward lane array gets updated and the corresponding lanes are set to occupied. The lanes can only be freed by credit
passing. When a flit leaves a virtual channel lane it will send a credit back to the previous switch. In the case that the flit is a tail flit, the credit will also contain information that states that the lane should be freed.

5.5 Simulator Structure

The structure of the simulator in OMNeT++ is made up to closely resemble the build up of a real NoC switch. The simulator is built up hierarchically. Each part of the switch is presented with an individual compound or simple module. The internal signalling of the switch is implemented in the simulator with message passing. The user of the simulator can easily understand the inner workings of the switch by running a simulation. The complete module build up of the OMNeT++ simulator can be seen in Figure 34. The rest of this chapter will go through the most important modules presenting them with an explanation of the OMNeT++ implementation. All of the modules are either compound modules or simple modules. The compound modules consist of simple modules defined in its own Ned file and C++ file. The C++-file describes the functionality of the simple module.

Figure 34: Network simulator module hierarchy with a * noting modules used multiple times
5.5.1 The Topology
The topology that was implemented is a k*k mesh presented in Figure 35. A k*k mesh has proven to be a well suited alternative for NoCs. The width and height of the mesh is defined by the user in the configuration file. The topology is a compound module that is described with the help of the NED language. In the NED file the sub-modules defined are a switch module and a statistics module. The statistic module is defined in a separate NED file and is used for statistics gathering and feasibility testing. Each switch in the topology is based on the same module description. The topology NED file only describes the structure of the topology of how switches and channels are connected. The switch and channel used are presented in other NED files.

![Figure 35: A screenshot of a 4*4 mesh topology](image)

5.5.2 The Switch
The switch architecture of the NoC was presented in Figure 27. The switch has an own compound module that defines the structure and connectivity of a lane allocator, switch allocator and crossbar, forward allocator, input gate, receiver, generator and upload logic.
5.5.3 The Link and Gates

In the channel NED file we describe the link used to connect the switches. Two links are used to implement a bidirectional link. The mesh has two bidirectional links in each direction. One is used for flit transfer while the second is used for credit-based flow control. All the switches in the mesh are constructed like Figure 27. The k*k mesh is not a torus mesh so all the input and output gates may not be connected depending on the position of the switch in the mesh. In switch 0 only the north and west input gates are connected, and the south and east are the only output gates connected. Each gate is defined with a name depending on the direction of the link. North, south, east and west are the names of the gates. An output gate going south is therefore connected to the south input gate of the next switch. The routing algorithm makes sure that no flits get scheduled to a gate that is not connected.

5.5.4 The Generator

The generator can generate messages on a random basis, or it can generate messages based on the message set defined by the statistic module. This is dependent on the outcome of the simulation the user wants. If the simulator performance needs to be tested then the user will define that the generator should generate its own random messages. The statistics simple module found in the topology compound module is the place where we write out statistics, run feasibility tests and define messages. At start-up time a random message set is generated and a feasibility test may be applied to it. The user can choose to make the generator generate messages based on this message set. The generator will at the start up of the simulator go into the statistic module and copy out all the information for the messages that needs to be sent from the generator. These messages are then generated at periodic intervals depending on the period of the message.

5.5.5 The Upload Logic

The “pcktbuffer” is a compound module shown in Figure 36, and it defines the upload logic. It is based on Z. Lu’s [41] ideal flit admission approach. In an ideal approach all the uploading buffers are connected to every output gate.
The Packet Buffer

The generator sends all the packets it generates to the packet buffer in the upload logic noted as “fifopkts” in Figure 36. This is an infinite FIFO buffer were all the packets are stored before going into the network. Packet storage delay is not considered in this study so the flits will get their time stamp when leaving the packet FIFO. The time stamp is set to the current simulation time.

Upload Flit Buffers

Four flit buffers are connected to the packet buffer marked as “fifo3[0-3]” in the figure. The size of a flit buffer is equal to the maximum size of one packet. The flit buffers may not all be in use. The number of active flit buffers depends on the number of output gates. At start up time, the initialize function counts the number of active gates. An array is situated in the FIFO packet buffer that contains information about the status of the flit buffers. A flit buffer must be free and active in order to be used. A self message is sent to the FIFO packet buffer at each cycle. The buffer checks for messages in its buffer and for free flit lanes. If a lane is free and active then the packet buffer sends a packet to the free flit buffer and sets the buffer to “occupied” in the status array for the flit buffers. No messages are interleaved in the flit buffer. So when a tail leaves the flit buffer, it triggers
the transmission of a message that is sent back to the FIFO packet buffer. This message contains information that the lane should be set free in the status array of the flit buffers.

On the arrival of a header flit to the flit buffer, the buffer copies this flit and sends it up for routing. If a flit buffer is granted access to use one of the output gates then we get a grant flit on the input of the queue. The grant flit also contains the new routing information for the flit. This is updated before the flit leaves the FIFO. When the flit leaves the router we need to send up the next flit information to routing. The flit information is only used to set the state to scheduling. We also need to send credit messages when a flit leaves the queue. If the flit is a tail flit then we need to free up the lane associated with this packet queue. The last thing we do is to send a message back to the packet buffer to say that the last flit has left the packet queue and that it can be used again.

**Routing Logic and State Controller**

The module named “route 2” in Figure 36 contains routing logic and a state controller. The state controller contains all the information needed for a flit buffer. Figure 37.a illustrates the array that is associated with each flit upload lane and virtual channel lane. The array gets updated when a header has been sent to routing. The priority of the message, time of creation, state, old VCID and old gate gets updated. A flit buffer is identified by the number of the upload flit buffer or VCID and the gate or core where the lane is found. After routing the XY routing information gets updated and new gate is set to the gate the packet needs to schedule. The state gets continuously updated by the routing logic, lane allocator, forward lane allocator and switch allocator. The new VCID is the VCID of the new lane that gets associated with the flit buffer. The New VCID is set by the forward lane allocator. Figure 37.b gives an example of a possible status of a lane. We can see that the lane is found in the uploading buffer (Core). The lane has got a lane associated with it at the eastern switch with the virtual channel lane number two. The lane is in the switch arbitration state and may contend for a schedule on the eastern output gate.
The routing logic updates the XY routing header and state controller. On an arrival of a header flit the routing logic needs to update the XY routing and find the output gate information. Note that the XY routing is updated before a flit is sent on the output port. The XY routing implemented even gets updated in the uploading routing logic. After routing, the state gets set to lane allocation. If the flit is not the header then we set the state to scheduling. The state controller needs to be updated by a header arrival.

If an upload flit buffer gets a grant to use an output gate the switch allocator will send a grant message down to the routing module. The routing module then adds the new routing information found in the state controller and passes the grant on to the correct lane.

### 5.5.6 The Lane Allocator

The lane allocator has been explained earlier. One of the tasks of the lane allocator is the passing of credits to the correct receivers. All credits that come from the virtual channel lanes and upload flit buffers are sent to the lane allocator. The lane allocator passes the credits on to the correct previous forward lane array and the local forward lane array if needed. The main responsibility of the lane allocator is deciding which message will get a lane allocated to it in the next switch. The forward lane array does the actual allocation. The lane allocator is the one that decides which one of the contending lanes that should be allowed to send a lane allocation request to the forward lane array first. A lane contention is solved by priority.
5.5.7 The Forward Lane Allocator Array

The forward lane allocator associates lanes on requests from the lane allocator. The first request that is sent from the lane allocator is the first request that is processed by the forward lane allocator. It checks for free buffers and associates if a buffer is free. Another task for the forward lane allocator is to check for free cells in the lanes of the neighbouring nodes. An array containing information about number of free cells of the VC and the availability of a neighbouring lane is therefore maintained. Credits are processed on arrival and decrements or increments the counters of free cells in the table. A lane can also be set free or occupied.

5.5.8 The Switch Allocator and Crossbar

The crossbar is a fully connected crossbar that routes the flits to the correct output, while the switch allocator is used to decide switch arbitration. A priority and oldest-first scheduling arbitration algorithms have been implemented to determine which lane is granted the output port. Contentions are resolved for each output gate based on the arbitration scheme. This scheme does not find the maximal matching of the messages to each output gate, but services the messages with the highest priority. If a maximal matching were to be implemented, it might effect the outcome of the feasibility tests. The feasibility test assumes that a message with higher priority will be given passage through the router. In a maximal matching scheme, a higher priority message might get blocked by other smaller prioritized messages. The average latency in a maximal matching scheme would however be decreased.

5.5.9 The Input Gates

The switch design implements virtual channel input buffering. A compound input gate module is connected to each input. The module defines the virtual channels, routing logic and state controller. The number of virtual channels and the size of a virtual channel lane are all user defined. Figure 38 shows an input gate module that contains four virtual channels. The number of flits that can be stored in a virtual channel defines the size.
Input Controller and Lanes
The first encounter of a flit that enters an input gate is the input controller, shown as “node 1” in Figure 38. The controller sends the flit to the correct virtual channel lane depending on its defined VCID. Each lane has a VCID associated with it. This VCID is used to route the flit to its correct lane. The flit gets stored in the lane and the flit type is checked. If the flit is a header then we need to send the routing information up to the routing logic.

State controller and Routing
In the input logic we separate the routing and state controller to make the switch more clear and representative. The state controller is defined with the simple module “state” as seen in Figure 38. The module contains information for all the virtual channels as described in the uploading module. The functionality of the “state and route” modules is the same as the “route2” module in the upload compound module. The only difference is that we can now see the information exchange between the two. At the input gate module we also find direct connections to the flit receiver. A header flit that is headed for the receiver sets the state of the lane to reception. Flits that are later received will go directly to the receiver. They do not contend for any output gate or lane.

Figure 38: A screenshot of the input gate logic
5.5.10 The Receiver

This flit ejection design was based on Z. Lu's [40] ideal flit ejection model. Screenshots from the simulator are found in Figure 38 and 39. A virtual channel in Figure 38 contains connections to both the crossbar and the receiver. Upon arrival of a header destined for the resource the state of the lane gets set to reception. Every successive flit that enters the buffer will then be sent directly to the resource. As can be seen in Figure 39 every virtual channel has a flit ejection buffer connected to it. The size of the flit ejection buffer equals the maximal size of a packet. Each virtual channel buffer is connected to a packet sink module. The packet sink module is a compound module that contains buffers equal to the number of virtual channels connected to an input gate. The flits get sent to the sink after arrival of the whole packet. The sink removes the packet from the network and writes out necessary statistic information.

![Figure 39: Two screenshots of the receiver logic](image)

5.6 Self Triggered Modules and Priority

The switch allocator, statistics module, generator and packet uploading buffer are all scheduled to be triggered at the same simulation time. Each self message is appended a priority. This is used by OMNeT++ to solve the scheduling contention for the modules. The switch module need to be scheduled before the statistics module and therefore has a higher priority. The same goes for the generator and packet buffer. The packet must be sent to the packet buffer before the packet buffer can send the packet to the flit uploading buffer. A higher priority has therefore been given to the generator.
Chapter 6
The Simulation Results

6.1 Definition of Simulation Result Units

The unit of the results from the simulations are given by the definitions below:

- Link Utilization (LBU) for the OMNeT++ simulations is described by the average number of active links per cycle. For the simulations the unit is the total number of received flits divided by the total number of links and the total number of simulation cycles in the network.

- The Link Utilization (LBU) for the feasibility tests is given by:

\[ LBU = \sum_{i=1}^{n} \left( \frac{\text{Hops}_i \times (\text{Size}_i / \text{Period}_i)}{\text{Links}} \right) \]

where “Size” is the number of flits of the message, and where “Links” is the total number of links in the network which is equivalent to that for the simulation.

- The Generated Traffic is the generated LBU.

- Throughput is defined by the offered traffic versus the received traffic. The unit of the traffic is for the simulations the number of messages received per cycle per node.

- The latency is calculated, from the time when the message leaves the packet queue to the time all the flits have arrived at the receiver, in the number of cycles. A message is given a timestamp equal to the current simulation time on the departure from the packet queue as seen in Figure 40. We are able to calculate the total latency of the message on the arrival of the tail flit at the receiver.
6.2 The NoC Simulator Validation

In order to validate the OMNeT++ network simulator we write out statistics information for latency, link utilization and throughput and compare them with expected results.

A simulation is run until it reaches a steady state. With a steady state we mean that there are no significant changes in the results if the simulation time was extended. For this simulation a steady state was reached after 20,000 cycles.

Random messages are created in the generator of each node at a given cycle time. The cycle time is given in the configuration file and specified by the user. It specifies the time interval of message creation. At each cycle time a single message gets created in each node and outputted to the packet queue in the upload logic. The cycle time between message creations is varied at each run so that we can get statistical information based on the load in the network. A figure presenting the statistic information can then be created. A simulation needs 11 data points and therefore 11 runs to get a satisfactory curve. The cycle times between message creations for the runs are 200, 60, 25, 20, 15, 13, 12, 11, 10 and 8. In order to get average results we run 10 simulations. As we need 11 runs to get the result of one simulation we will need a total of 110 runs of 20,000 cycles each.
The priority and destination of the message are given by random number generators (RNG). All the other values are fixed and specified by the user before the simulation is run. No periodic messages are implemented and no deadlines are given. In this test a messages is specified with a four-tuple $M_i = \{s, r, i, f\}$ for source, receiver, priority and size. For a node $j$ a message $i$ gets created with the attributes given in Table 4.

<table>
<thead>
<tr>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 98]</td>
</tr>
<tr>
<td>Destination</td>
</tr>
<tr>
<td>[0, 63]</td>
</tr>
<tr>
<td>Size</td>
</tr>
<tr>
<td>[4]</td>
</tr>
</tbody>
</table>

Table 4: Message attributes for the network simulator validation

As each node produces a message at each cycle time, the source address is given by the node that creates the message.

Random numbers are produced using deterministic algorithms. The algorithm uses a seed value as a starting point to produce the random number and the next seed [37]. Non-overlapping series of random numbers are used to avoid correlation. We therefore need to use seed values that are well apart for the priority and destination RNGs for all 10 simulations.

The last configurations given by the user are found in Table 5. In this master thesis we have not studied the effect of packetisation. Consequently the size of a message is equal to the packet size and only a single packet will be generated. The number of lanes and VCs has been chosen so that we will get a cost effective, but still a low latency implementation.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Lane Size</th>
<th>Nr of VCs</th>
<th>Packet Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>8*8 2-D Mesh</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5: Simulation configurations for the network simulator validation

The statistics information is outputted during the network simulation to an output file. The modules in the OMNeT++ that need to output information do so through the statistics
module. All the modules call functions in the statistics module that either registers the information or writes it out into an output data file.

### 6.2.1 Latency vs. Link Utilization

A graph representing the latency versus link utilization is given in Figure 41. With link utilization we mean the average number of active links per cycle. A function in the statistics module is called each cycle by the nodes that receive one or more flits on their inputs. The statistics module registers all the calls and calculates the average link utilization. For the latency we calculate the average routing time for all the received messages. Note that the switch-delay is one cycle and that routing over a link takes zero cycles. From the figure we can read out a base latency of 12.5 cycles and we can see that the network saturates at a link utilization capacity of around 0.6. Expected saturation was between 0.4 and 0.7 [43]. As can be seen at the highest link utilization it drops back from the previous results. This is due to the increased contention in the network. The network is saturated and the LBU will be reduced.

![Figure 41: The latency vs. link utilization](image)

### 6.2.2 Throughput

The throughput of the network is given in Figure 42. The throughput is given by offered traffic versus received traffic. Offered traffic is given by number of generated messages normalized with the number of cycles and the number of nodes, while received traffic is the number of received messages normalized with the number of cycles and the number of nodes. As we can read out from the figure we get saturation in the network with 0.1
packets per cycle per node. Further examination shows that at the saturation point 98% of the offered traffic is received by the resources.

6.2.3 Latency vs. Link Utilization Based on Priority Ranges

Figure 43 shows four different graphs. The latency versus the link utilization graph from above with static priority ranging from 0 to 98 has been plotted with three other graphs that presents the latency based on priority. We have subdivided the priorities into three different priority ranges. A message can be given a priority in the high [66, 98], medium [33, 65] or low [0, 32] range.
The high priority range has as expected the lowest base latency of 9.8 which is 5.2 cycles on average better than the lowest priority range. For the medium range we see that the base latency is the same as the priority ranging from 0 to 98. At the saturation point we see that the latency of high priorities is 28.3 cycles as compared to the low priority messages with an average latency of 76 cycles. Priority inversion will be a problem in the network since we only have 4 virtual channels. The latency of the higher priorities will go down when we implement more virtual channels.

6.2.4 Latency vs. Number of Packets Received

As a last figure we present the latency versus the number of packets received. This simulation was only done for one data point and no average was made. The link bandwidth utilization for the simulation was 0.25 and the latencies for all the received messages were outputted. A total of 51,199 messages were received. A message that goes over one link will have latency equal to the message size. The smallest latency in Figure 44 is 4 cycles which is the same value as the message size. The figure also shows us that most packets get a latency of 9 or 10 cycles which means that the average number of hops in the simulation is 6 or 7 links. The expected average number of hops for the message was $2/3 \sqrt{N} = 5.33$, where $N$ is the number of nodes in the network [44]. This means that the expected average latency is 8.33 which is close to the simulated average latency.

![Figure 44: Latency vs. number of received packets](image-url)
6.3 The Real-Time Feasibility Random Simulations

6.3.1 Random Message Set

At start up of the simulator the statistic module generates a random message set. This message set is constrained by the link bandwidth utilization (LBU) on the links and the generated network capacity allowance specified by the user.

The link bandwidth utilization where \( m \) channels pass on a link \( j \) [9] is given by:

\[
B_j = \sum_{i=1}^{m} \frac{L_i}{p_i}
\]

where the link bandwidth utilization is the sum of the length (\( L \)) divided by the period (\( p \)) for all the messages passing that link.

The network utilization for \( k \) channels [9] is given by:

\[
U_{offered} = \sum_{i=1}^{k} \frac{(H_i) \cdot q_i}{C}
\]

where \( q_i \) is the length (\( L_i \)) in flits divided by the period (\( p_i \)), \( H \) is the number of hops and \( C \) is the total number of links in the network.

A message set is created based on the flow chart found in Figure 45. The user specifies a threshold from 0.1 to 0.9. Messages are created until the network utilization of all the generated messages \( U_{Generated} \) is equal to or higher than the threshold. The link bandwidth utilization test is performed on every message. A message is only added to the message set if it does not breach the capacity of a link in its path. All the messages that passed the LBU check will form the message set.
6.3.2 NoC Simulation Process

Figure 46 shows the process of the message set creation to feasibility testing and OMNeT++ execution. The initial traffic algorithm generator and LBU check filter were both explain earlier. After the message set is created we get network utilization $U_{\text{Offered}}$. This message set is either inputted to one of the three feasibility tests or sent directly to the OMNET++ simulation. The feasibility tests will analyze the feasibility of the messages and all infeasible messages will be taken out of the message set. The percentage of all passed messages and the new reduced network utilization are found. These reduced message sets ($U_{\text{LL}}$, $U_{\text{BDG}}$ and $U_{\text{CT}}$) are then used as an input for the OMNeT++ simulator. After execution of an OMNeT++ test we will get a new pass ratio ($U_{\text{OE LL}}$, $U_{\text{OE BDG}}$ and $U_{\text{OE CT}}$). A direct input of the message set to the OMNeT++ simulator will give us the actual number of infeasible messages and network utilization ($U_{\text{OE}}$). All
messages in the message set are set feasible and sent to the OMNeT++ simulator. A message stream where a message fails its deadline will be set infeasible. The simulation will still run with this message stream in the network. The results from these executions might therefore be more pessimistic than in reality.

![Diagram of traffic model](image)

**Figure 46: The traffic model for all the simulations executed**

### 6.3.3 Simulation Time

In order to find the worst case response time of a message, we inject all the messages into the network at time zero. Subsequent messages are then injected based on their periods. A message with a period of 50 will be injected into the network at time 0, 50 and integer multiples of 50. The simulation time is trice the duration of the longest period. A message with the longest period will then be scheduled three times. As we simulate the worst case response time at time zero there is no need for a longer execution time.

### 6.3.4 Head of Line Blocking

The FIFO packet uploading queue in the source node may experience Head of Line (HoL) blocking. When contention is high all the uploading flit buffers may be occupied. The packet at the head of the uploading packet queue will then be blocked in place. In Figure 47 we see an illustration of the uploading logic input queues. The generator in
each node generates messages at the given period. The generated messages are stored in an infinite packet queue at the source node. The packets can only leave the packet queue if a flit buffer is empty. HoL blocking will therefore become more frequent as the LBU increases.

Figure 47: Illustration of the uploading logic

### 6.3.5 Simulation Settings

The message sizes are uniformly distributed between predefined values. All the tests were conducted on an 8*8 2-D mesh topology. This gives us 64 nodes and 224 links in the network. The bandwidth of the link is one flit per cycle. The priority scheme used was the Heuristic 3 presented earlier. The destination and source address is randomly chosen. Note that the same source and destination are not allowed. After the message set is created all the messages in a message stream get appointed a distinct priority. Several tests were run where the virtual channels, lane size, message size and periods were varied. The results presented are the average of ten different messages sets. A message stream is defined with the seven-tuple $M_i = \{s, r, i, f, p, d, T\}$.

For all the simulations with a message set of $n$ messages we use:

<table>
<thead>
<tr>
<th>Topology</th>
<th>Source $s$</th>
<th>Destination $r$</th>
<th>Priority $i$</th>
<th>Priority Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>8*8 2-D Mesh</td>
<td>[0, 63]</td>
<td>[0, 63]</td>
<td>[0, n-1]</td>
<td>Heuristic 3</td>
</tr>
</tbody>
</table>

Table 6: Simulation settings for random message set simulations
6.3.6 The Feasibility Models

The latency of the message can be calculated either by using computation time or size for all the messages. The BDG actually does all their calculations based on size. The number of free slots is actually based on the sizes for the contending messages. This number is then checked against the computation time of the message being tested. In order to compare the message streams we have made a change in the BDG model. The model now calculates the number of free slots based on their computation time. Also the Lumped Link has been made a little more pessimistic then necessary. For the LL model only direct messages are considered. Again we need to evaluate the models against each other. An indirect level has therefore been added to the LL model.

For the three feasibility tests we use:

<table>
<thead>
<tr>
<th>Feasibility Test</th>
<th>Indirect Levels</th>
<th>Slot Occupation Based On</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL</td>
<td>1</td>
<td>Computation Time</td>
</tr>
<tr>
<td>BDG</td>
<td>1</td>
<td>Computation Time</td>
</tr>
<tr>
<td>CT</td>
<td>Infinite</td>
<td>Computation Time</td>
</tr>
</tbody>
</table>

Table 7: Feasibility model attributes

6.3.7 The Feasibility Tests Performed

The packet size is created randomly from a set of four values presented in Table 8. Each packet will get appended a header and tail. The size therefore increases by two flits. The extra sizes of these two flits are however ignored for the LBU calculations after simulation and the feasibility tests. Each size gets appended a random period from a set of three distinct periods. The deadline is set to the same as the period.

<table>
<thead>
<tr>
<th>Packet Payload Size $f$</th>
<th>Period $p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>[50,100,150]</td>
</tr>
<tr>
<td>64</td>
<td>[100,200,300]</td>
</tr>
<tr>
<td>128</td>
<td>[200,400,600]</td>
</tr>
<tr>
<td>512</td>
<td>[800,1600,2400]</td>
</tr>
</tbody>
</table>

Table 8: Medium message simulation attributes
The first test performed on the random message sets is the implementation of the feasibility tests. In order to get the $U_{LL}$, $U_{BDG}$ and $U_{CT}$ we inject message sets with network utilization $U_{offered}$ into the three feasibility tests. For these tests we do not need to inject them into the OMNeT++ network simulator. The feasibility tests are performed in the statistics module. Consequently a run is ended at simulation time 0.1.

In order to look at the pessimistic nature of the feasibility tests $U_{OE}$, we run tests on the message sets without performing any feasibility testing on them. All the message streams are marked feasible and sent into the network for a simulation. There is however an uncertainty aspect to the LBU results. Increasing the simulation time will decrease the LBU. There will be no steady state as increased HoL blocking in the packet buffer (as explained in 6.3.4) will increase the number of infeasible messages. Messages will not be outputted at the specified time, and it will lead to missed deadlines for feasible message streams. For these tests the maximum period is 2400. We therefore use a simulation time of 7200 cycles so that each message stream will have at least three instances. The graphs of $U_{LL}$, $U_{BDG}$ and $U_{CT}$ are also added to each presented $U_{OE}$ simulation figure in order to make a comparison. Different tests are run where we vary the number of virtual channels and number of cells in a lane.

At last we perform tests on $U_{LL}$, $U_{BDG}$ and $U_{CT}$ to see how many missed messages we have when we simulate the feasible message sets using OMNeT++. For the $U_{OE\_LL}$, $U_{OE\_BDG}$ and $U_{OE\_CT}$ we input the $U_{LL}$, $U_{BDG}$ and $U_{CT}$ into the OMNeT++ network simulator and look at the pass ratio of the messages. A simulation run is executed three times longer than the duration of the longest period. We run the same tests as we did for $U_{OE}$. Presented below we have a list of the tests that will be performed in this study for feasible random message sets.

A simulation result is the average from 10 simulations. One simulation needs 9 runs. So in order to create a curve we need 90 runs. Two main simulation results are presented: One where the header experiences a delay in the switch of 1, and one where the routing delay for the header is 2.
Feasibility tests for random message sets with routing delay 1:

1. Feasibility Testing ULL, UBDG and UCT without OMNeT++ Execution
2. Variation in the Number of Virtual Channels
   a. OMNeT++ Execution UOE without Fsb Testing including ULL, UBDG and UCT
   b. OMNeT++ Execution for UOE LL, UOE BDG, UOE CT
3. Equal Storage Place
   a. OMNeT++ Execution UOE without Fsb Testing including ULL, UBDG and UCT
   b. OMNeT++ Execution for UOE LL, UOE BDG, UOE CT
4. Variation in the Number of Cells
   a. OMNeT++ Execution UOE without Fsb Testing including ULL, UBDG and UCT
   b. OMNeT++ Execution for UOE LL, UOE BDG, UOE CT

Feasibility tests for random message sets with routing delay 2:

1. Feasibility Testing ULL, UBDG and UCT without OMNeT++ Execution
2. Variation in the Number of Virtual Channels
   a. OMNeT++ Execution UOE without Fsb Testing including ULL, UBDG and UCT
   b. OMNeT++ Execution for UOE LL, UOE BDG, UOE CT

6.3.8 Feasibility Tests with Routing Delay Equal to 1

Feasibility Testing ULL, UBDG and UCT without OMNeT++ Execution

The offered traffic is inputted to the three feasibility tests. No OMNeT++ simulation is run. We see that as expected the CT feasibility test produces the highest average LBU of 0.324 and an average of 17.8 discarded message streams from an average total of 58.4. This is due to the fact that the CT takes advantage of disjoint concurrent messages. The BDG and LL are almost identical. The BDG however has an average LBU of 0.314 as compared to the LL which has an average LBU of 0.312. The average number of discarded messages is 19.3 and 19.7 respectively from the average total of 58.4. The CT has an average LBU of 72.11 percent of the offered traffic which is 2.77 percent larger than LL. The average LBU of the offered traffic from BDG is only 0.60 percent larger than LL. Even though the BDG takes advantage of indirect messages there is only a little improvement of the LBU as compared to the LL. The BDG and LL are both pessimistic feasibility tests. This is however also true for the CT. The number of indirect contentions is infinite. All the first schedules for the indirect contending
messages at cycle time zero will propagate down to the message being tested. The same thing occurs for all the slots that are not released.

![Generated Traffic vs. Link Bandwidth Utilization](image)

**Figure 48:** Generated traffic vs. link bandwidth utilization for $U_{LL}$, $U_{BDG}$ and $U_{CT}$

1. **Variation in the Number of Virtual Channels**

   For this test we perform 3 simulations where the lane size is kept constant and where we change the number of lanes like we see in Table 9.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Nr of VCs</th>
<th>Lane Size</th>
<th>Total Storage Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 9:** Variation in the number of virtual channel simulation attributes
OMNeT++ Execution UOE without Fsb Testing including ULL, UBDG and UCT

The offered traffic is inputted into the OMNeT++ simulator without a feasibility test performed. The outcome (UOE) of the test is plotted beneath. ULL, UBDG and UCT without OMNeT++ execution is added to the graphs for comparison purposes. The LBU of 5 and 4 virtual channels is 0.325 and 0.297 respectively. They are both close to the LBU of CT, BDG and LL. The choice of 5 virtual channels seems to match the CT at an offered traffic of 0.45 and it is above the LBU of all the feasibility tests. With the use of 4 virtual channels we get an LBU that is lower than the three feasibility tests at the highest offered traffic. It seems that we need at least 5 virtual channels in order to avoid priority inversion for the feasibility tests. The uncertainty of these results is however too large when the results are so close together, and a further test where we input the traffic after the feasibility tests (ULL, UBDG and UCT) is needed. The results of UOE LL, UOE BDG, UOE CT is shown next. One thing that is fairly certain however is that only 2 virtual channels will lead to too much priority inversion and therefore too many missed messages. Only 2 virtual channels is as a result not an optimal design choice.

Figure 49: Generated traffic vs. link bandwidth utilization for variation in VC for UOE
OMNeT++ Execution for $U_{OE\, LL}$, $U_{OE\, BDG}$, $U_{OE\, CT}$

For this test OMNeT++ executed simulations on the $U_{LL}$, $U_{BDG}$, $U_{CT}$ feasible message sets. The outcome of the number of missed messages is shown in Figure 50 and Table 10. Both BDG and LL are pessimistic in nature, and we experience no missed messages for 5 and 4 virtual channels. The curves are aligned with the X-axis. Even though the $U_{OE}$ simulation showed that there would be some missed messages for BDG and LL for 4 virtual channels at the highest contention in the network, we notice that after the feasibility test and following simulation of the feasible message sets there are no missed messages. The curve of the CT experienced 2 missed messages in 3 of the 10 results in the first run. There were no missed messages for the last 9 runs. The missed messages came from the same message stream, and the stream had the lowest priority associated with it. Although we had a few missed messages for the CT feasibility test there will only be 1.5 in 10,000 missed messages. Consequently, 4 virtual channels is a good choice for all of the three feasibility tests. As expected we have a higher missed message ratio for 2 virtual channels. The BDG and LL are almost identical and they experience the same missed messages. The only reason why BDG has a better result is due to the fact that BDG has more feasible message streams and therefore the percentage will be lower. The CT has a larger LBU and therefore when we decrease the number of virtual channels it is expected that the CT will get the highest number of missed messages. The BDG and LL has a maximum of 0.16 % missed message ratio while the CT is higher at a 0.5% missed message ratio. These percentages are low but will be too high for hard real-time requirements.

Figure 50: Missed messages in percent vs. LBU for variation in VC for $U_{OE\, LL}$, $U_{OE\, BDG}$ and $U_{OE\, CT}$
CHAPTER 6. THE SIMULATION RESULTS

<table>
<thead>
<tr>
<th>Model and Settings</th>
<th>LL 5 VC</th>
<th>LL 4 VC</th>
<th>LL 2 VC</th>
<th>BDG 5 VC</th>
<th>BDG 4 VC</th>
<th>BDG 2 VC</th>
<th>CT 5 VC</th>
<th>CT 4 VC</th>
<th>CT 2 VC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Missed in %</td>
<td>0</td>
<td>0</td>
<td>0,156</td>
<td>0</td>
<td>0</td>
<td>0,155</td>
<td>0,015</td>
<td>0,015</td>
<td>0,487</td>
</tr>
<tr>
<td>At Offered Traffic</td>
<td>X</td>
<td>X</td>
<td>0,236</td>
<td>X</td>
<td>X</td>
<td>0,236</td>
<td>0,328</td>
<td>0,328</td>
<td>0,328</td>
</tr>
<tr>
<td>% With Distinct Messages Missed at Max</td>
<td>0</td>
<td>0</td>
<td>0,450</td>
<td>0</td>
<td>0</td>
<td>0,444</td>
<td>0,333</td>
<td>0,333</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 10: Statistics information for variation in VC for $U_{OE, LL}$, $U_{OE, BDG}$ and $U_{OE, CT}$

2. Equal Storage Place

5 simulations are performed where the equal storage space is kept constant. The attributes are shown in Table 11. We vary both the number of virtual channels and lane sizes, but we keep the total storage space at a constant of 16 flits.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Nr of VCs</th>
<th>Lane Size</th>
<th>Total Storage Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 11: The equal storage space simulation attributes

OMNeT++ Execution $U_{OE}$ without Fsb Testing including $U_{LL}$, $U_{BDG}$ and $U_{CT}$

Figure 51 shows us the graphs from the test. We start by looking at 1 channel and 16 cells per VC. This would not be a good design choice as the priority inversion would be large due to the fact that we only have one lane. In addition we have a very large lane size. A lane can be freed and allocated to a message with high priority. A low priority message may however still be in the lane as lanes are set free when the tail enters the lane. The higher priority message therefore has to wait until the tail has left the lane before it can contend for the output. As excepted we have a maximum LBU at a generated traffic of 0.5 at 0.141. The LBU then falls down to 0.108 at generated traffic 0.9. At a high LBU there will be too much contention and priority inversion in the network, and the LBU will fall at high offered traffic rates. If we compare with 16 channels and 1 cell we see that this curve steadily increases up to a maximum of 0.162. 16 VCs and
only one cell drastically reduce the priority inversion as in contrast to 1 VC and 16 cells per VC. Nevertheless, having only 1 cell does not make it possible for flits to be pipelined in the router. Credits must be sent back to decrease the number of free cells of the VC counter, before we can send a flit to the lane. 8 lanes and 2 cells per VC have a much better LBU of 0.332 which is 37.7 percent larger of the offered traffic than having 1 cell and 16 lanes. All the feasibility results are situated beneath this curve, and all the feasibility tests should have no missed messages. 2 lanes and 8 cells per VC will also experience a heavy priority inversion as 1 lane and 16 cells per VC did. It will also not be suitable with only a LBU of 0.226. 4 lanes and 4 cells per VC however looks like a better design alternative at an LBU of 0.311. The CT however is above the curve from the point of 0.17 from generated traffic for offered LBU. Executing the feasibly messages on the OMNeT++ simulator from the CT is the only way to determine if the CT has any missed messages. The LL and BDG are both well below the 4 lanes 4 cells per VC curve and should in theory have no missed messages. The only point where there might be some missed messages would be at an offered LBU of 0.45.

![Generated traffic vs. link bandwidth utilization for equal storage place for UOE](image)

Figure 51: Generated traffic vs. link bandwidth utilization for equal storage place for $U_{OE}$
OMNeT++ Execution for $U_{OE\ LL}$, $U_{OE\ BDG}$, $U_{OE\ CT}$

After performing simulation on the $U_{LL}$, $U_{BDG}$ and $U_{CT}$ we get the curves shown in Figure 52 and Table 12-14. Again 8 virtual channels and 2 cells per VC resulted in the same results as for 5 and 4 virtual channels as described previously. The CT model still has 1.5 in 10,000 missed messages. Increasing the number of virtual channels does not improve the outcome. Priority inversion is therefore not the reason for the missed messages. HoL blocking in the packet buffer causes messages to be delayed from entering the network at the specified time. Packets may therefore have a different latency than calculated by the feasibility models. The missed message missed its deadline due to the HoL blocking. At 4 VCs and 4 cells per VC all the feasibility tests actually have the same missed message ratio. In the sixth run they all experience that a message stream misses its deadline three times for one of the 10 results. Since the BDG and LL did not have this error with only 2 cells per VC, it must come from priority inversion due to the lane size. 4 VCs and 4 cells per VC therefore also seems to be a good design choice with only 3.1 messages in 10,000 missing their deadlines. As anticipated 16 VCs 1 cell and 1 VC 16 cells per VC gives dreadful results. Due to that we are not able to pipeline the flits in the virtual channels in 16 VCs 1 cell we get approximated 55% missed messages and at the lowest 50% for all the three feasibility tests. 1 VC 16 cells per VC gives better results but still over 10% missed messages for the highest LBU. In comparison 2 VCs 8 cells per VC only has a missed message ratio of 0.16 for the BDG and LL feasibility models and 0.471 for the CT model. This is still not expectable results for hard real-time purposes. Having 6 more cells per VC per lane only leads to a slightly better improvement than having 2 cells per VC.

![Figure 52: Missed messages in percent vs. LBU for equal storage size for $U_{OE\ LL}$, $U_{OE\ BDG}$ and $U_{OE\ CT}$](image)
Table 12: Statistics information for equal storage size for U_{OE,LL}

<table>
<thead>
<tr>
<th>Model and Settings</th>
<th>LL 16/1</th>
<th>LL 8/2</th>
<th>LL 4/4</th>
<th>LL 2/8</th>
<th>LL 1/16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Missed in %</td>
<td>55.5</td>
<td>0</td>
<td>0.031</td>
<td>0.156</td>
<td>10.9</td>
</tr>
<tr>
<td>At Offered Traffic</td>
<td>0.364</td>
<td>X</td>
<td>0.236</td>
<td>0.236</td>
<td>0.450</td>
</tr>
<tr>
<td>% With Distinct Messages Missed at Max</td>
<td>34.9</td>
<td>0</td>
<td>0.450</td>
<td>0.450</td>
<td>20.2</td>
</tr>
</tbody>
</table>

Table 13: Statistics information for equal storage size for U_{OE,BDG}

<table>
<thead>
<tr>
<th>Model and Settings</th>
<th>BDG 16/1</th>
<th>BDG 8/2</th>
<th>BDG 4/4</th>
<th>BDG 2/8</th>
<th>BDG 1/16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Missed</td>
<td>55.4</td>
<td>0</td>
<td>0.031</td>
<td>0.155</td>
<td>11.3</td>
</tr>
<tr>
<td>At Offered Traffic</td>
<td>0.364</td>
<td>X</td>
<td>0.236</td>
<td>0.236</td>
<td>0.450</td>
</tr>
<tr>
<td>% With Distinct Messages Missed at Max</td>
<td>34.5</td>
<td>0</td>
<td>0.444</td>
<td>0.444</td>
<td>21.0</td>
</tr>
</tbody>
</table>

Table 14: Statistics information for equal storage size for U_{OE,CT}

<table>
<thead>
<tr>
<th>Model and Settings</th>
<th>CT 16/1</th>
<th>CT 8/2</th>
<th>CT 4/4</th>
<th>CT 2/8</th>
<th>CT 1/16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Missed</td>
<td>55.1</td>
<td>0.015</td>
<td>0.031</td>
<td>0.471</td>
<td>13.0</td>
</tr>
<tr>
<td>At Offered Traffic</td>
<td>0.364</td>
<td>0.328</td>
<td>0.236</td>
<td>0.292</td>
<td>0.450</td>
</tr>
<tr>
<td>% With Distinct Messages Missed at Max</td>
<td>34.5</td>
<td>0.333</td>
<td>0.442</td>
<td>1.80</td>
<td>24.6</td>
</tr>
</tbody>
</table>

3. Variation in the Number of Cells

Here we look at the variation in the number of cells per VC. As the average message size is large we only perform 2 simulations; one with 2 cells per VC and the other with cell size 16. The results show that there is little difference between the two. That is why we only perform these two simulations. Table 15 shows us the settings of the lanes and cells per VC for this simulation. Previously we showed that having 4 virtual channels is a good choice in order to reduce the priority inversion. We therefore choose to implement 4 virtual channels for this test.
Simulation Nr of VCs Lane Size Total Storage Place
1 4 2 8
2 4 16 64

Table 15: Variation in the number of cells per VC simulation attributes

**OMNeT++ Execution \( U_{OE} \) without Fsb Testing including \( U_{LL}, U_{BDG} \) and \( U_{CT} \)**

As stated above there is only a small variation in the results of the two tests. Only at offered message LBU of 0.42 and 0.45 we see some significant change. At offered LBU of 0.45 the difference between the two is 0.016. As these results are so close together we need to input the feasible traffic from the feasibility tests into OMNeT++ before we can say anything certain. We see however that having 16 cells per VC gives a better result than then having only 2 cells per VC in the lane.

---

**Figure 53: Generated traffic vs. link bandwidth utilization for variation of number of cells per VC for \( U_{OE} \)**
OMNeT++ Execution for $U_{OE \text{ LL}}, U_{OE \text{ BDG}}, U_{OE \text{ CT}}$

We again run the feasible messages sets for $U_{LL}$, $U_{BDG}$ and $U_{CT}$ to get the missed message ratio as we can see the results of in Figure 54 and Table 16. The results from the $U_{OE \text{ LL}}$ and $U_{OE \text{ BDG}}$ actually show no missed messages for both results. The one missed message stream from having only 4 cells per VC does now manage to reach its deadline. 4 virtual channels and 2 cells per VC actually has a better missed message ratio than having 16 cells per VC for the CT. Priority inversion will lead to a maximum average missed message ratio of 0.024 for 16 cells per VC. The inversion however only leads to an increase of 0.009%.

![Figure 54: Missed messages in percent vs. LBU for variation of number of cells per VC for $U_{OE \text{ LL}}, U_{OE \text{ BDG}}$, and $U_{OE \text{ CT}}$](image)

<table>
<thead>
<tr>
<th>Model and Settings</th>
<th>LL 2 Cells</th>
<th>LL 16 Cells</th>
<th>BDG 2 Cells</th>
<th>BDG 16 Cells</th>
<th>CT 2 Cells</th>
<th>CT 16 Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Missed in %</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0,015</td>
<td>0,024</td>
</tr>
<tr>
<td>At Offered Traffic</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0,328</td>
<td>0,292</td>
</tr>
<tr>
<td>% With Distinct Messages Missed at Max</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0,333</td>
<td>0,360</td>
</tr>
</tbody>
</table>

Table 16: Statistics information for variation of number of cells per VC for $U_{OE \text{ LL}}, U_{OE \text{ BDG}}$, and $U_{OE \text{ CT}}$

6.3.9 Feasibility Tests with Routing Delay Equal to 2

This test is similar to the variation in the number of VCs test above. However now we see if there is much change if we have a routing delay for the header equal to 2. The simulation settings are given in Table 17. The deadline has not been changed from the
previous simulation with routing delay equal to 1. This will lead to an increase in latency and therefore missed messages.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Nr of VCs</th>
<th>Lane Size</th>
<th>Total Storage Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 17: Variation in the number of VCs with routing delay 2 simulation attributes

Feasibility Testing $U_{LL}$, $U_{BDG}$ and $U_{CT}$ without OMNeT++ Execution

The following figure shows us the average LBU outputted after the three feasibility tests. No OMNeT++ simulation has been performed on the message sets.

We know from the previous simulations with routing delay 1 that there was a difference from the CT and LL of 2.77 percent for the offered traffic. Now the difference is 2.82
percent between when we compare the percentage of the offered traffic for the two of them. So there is no actual significant difference. The difference between BDG and LL was only 0.60 percent at routing delay 1 simulations for comparison of offered traffic. We see that the new simulated value with routing delay 2 is almost the half of the previous value with 0.33 percent for the offered traffic. The CT falls with an LBU of 0.016 as matched up to the previous value. The LL falls 0.17 while the BDG falls 0.18 in LBU. The differences of these results compared to the previous one are too small to make any conclusive remarks.

**OMNeT++ Execution UOE without Feasibility Testing**

Now if we look at the figure below we see that that the results are quite similar with the previous results. The simulation with 4 lanes and 2 cells per VC has dropped some when we compare the distance to the LL. The graph however is still below the other three graphs at the last offered traffic point.

![Figure 56: Generated traffic vs. link bandwidth utilization for variation in VC for UOE with routing delay 2](image-url)
OMNeT++ Execution for U_{OE LL}, U_{OE BDG}, U_{OE CT}

The results for the number of missed messages in percent for the U_{LL}, U_{BDG} and U_{CT} do not show any large changes from having a routing delay of 1. For 2 cells per VC we see a slightly improvement for the LL and BDG for the maximum missed message ratio, and a slightly worse result for the CT. The important thing that this simulation shows is that we see that the CT curve for 4 lanes and 2 cells per VC now have no missed messages like the BDG and LL models. HoL blocking does not have an impact on the results of the feasibility tests.

![Graph showing missed messages in percent vs. LBU for variation of number of VCs for U_{OE LL}, U_{OE BDG} and U_{OE CT} with delay 2](image)

**Figure 57:** Missed messages in percent vs. LBU for variation of number of VCs for U_{OE LL}, U_{OE BDG} and U_{OE CT} with delay 2
6.4 An M-JPEG Implementation Real-Case Study

In order to evaluate the feasibility models on a real-case study, a Motion JPEG (M-JPEG) encoder is implemented as multiple pipelined JPEG encoder architectures in parallel and mapped to a 3*3 mesh network. We have selected the M-JPEG encoder as it is illustrative and easy to understand.

6.4.1 M-JPEG

The M-JPEG encoder [45] is a video codec where each frame is separately compressed into a JPEG image. The encoder is decomposed into 3 independent pipelined tasks where each task is mapped to a node. The M-JPEG encoder comprises of a 2D-DCT, quantization and an encoding module as seen in Figure 58. Computations are performed on an 8 by 8 pixel block.

![M-JPEG encoder structure](image)

Each stage in the pipeline will perform operations on a block in a pipeline fashion. All the nodes in the architecture except the slowest one will waste time [46]. This is due to fact that the stages in the M-JPEG pipeline have different computational requirements. The faster stages in the pipeline will be constrained by the slowest stage, so all the faster modules will have the same output period as the most time-consuming. The M-JPEG will therefore work like a real pipeline where each module transmits a message at the same time. This is true for all the modules except the encoding module which will output at a rate that is four times slower than the rest.

6.4.2 Parallel Pipelined Encoder Architecture

In order to increase the contention and utilization in the network we apply a parallel pipelined encoder architecture [47]. The M-JPEG encoder processes frames by frames.
A single pipeline could compress a frame in parallel with other pipelines like shown in Figure 59. Each pipeline will then perform computations on a section of the video sequence. All the frames are sent to the same receiver module. A sender module is also needed that is responsible for sending out the output blocks to the correct encoder pipelines.

![Figure 59: Parallel M-JPEG encoder architecture](image)

### 6.4.3 Module Descriptions

The sender divides the video sequence into several sub-sections depending on the number of parallel pipelined encoders. All the frames in a single sub-section can only be sent to a specific encoder. The unsigned pixel data from a frame is grouped into blocks of 8*8 pixels and sent block by block and frame by frame. 1 pixel represents for our simulations 1 byte of data. A block of data from each sub-section is transmitted to the respective parallel encoders at a given time interval depending on the rate of the slowest module in the pipeline. The first part of the encoder is a 2D-DCT (Discrete Cosine Transform) [48] module. A level shift in the 2D-DCT module transforms an unsigned integer block into a block of signed integers. Pixels can then have a range from -128 to 127. The 2D-DCT itself transforms the block from the spatial domain to the frequency domain. It receives a block on its input and outputs on a block to block basis the data to the quantization unit. Execution of the 2D-DCT begins when it has received the complete block. A latency of 72 cycles is experienced for the first pixel in each block. In addition, the 2D-DCT module must wait for the complete block before being allowed to send it to
the quantization module. An extra latency of 63 cycles is therefore experienced for the block. We assume that every block will obtain the 72 cycles latency experienced for the first pixel. All the modules in the pipeline must wait for the complete block before execution can begin. The quantization unit divides the 64 DCT coefficients by values stored in a quantization table. A block of data is quantized with a total latency of 66 cycles. This is done in order to reduce the amplitude and to increase the number of zero values. It will output a block of data of 64 pixels. For our simulations we assume that a pixel data is 8-bit also after the quantization. In the encoding module we find a Zig-Zag operation, run length encoding and Huffman encoder. The Zig-Zag operation places the coefficients of the 2-D block in a sequence from low to high frequencies which is needed for Huffman coding. This 1-D block is then sent to the run length encoder. Here the data is compressed by conversion to Run-Size pair data bytes. Huffman encoding techniques are then implemented. Huffman coding is a variable-length coding method. Fewer bits are used to represent a symbol which appears more frequently than others. The output data length from the encoding module will vary due to the Run-Length and Huffman compression techniques. A data stream generator therefore packs the variable length words into packets. The size of the packet will comprise of 4 variable length words. This packet will be sent to the receiver periodically. A flit that is not filled up completely will be padded with extra bits.

6.4.4 Simulation Implementation

Information required for the three feasibility tests are dependent on the computation time and period for the message streams. We are not interested in the content that is being transmitted. The Motion JPEG encoder will therefore only be implemented on the abstraction level of the message flow. Nodes will not perform computations, but will output information as if they were. The information flow will be packets of 18 flits on a periodically basis, except for the information flow between the encoding module and the receiver where the packet sizes vary. As this size varies we will need a message trace from the output of the Huffman encoder. A message trace was generated from a 512x512 8 bits per pixel greyscale image. Only the first 400 blocks of image data were traced. The trace is then divided into groups of four blocks each. The encoding modules will then contain an array of 100 possible outputs. As we have implemented several pipelined encoders, each encoding module randomly selects a value from the array and sets the packet size equal to this value. The latency of the encoding module is assumed
to be equal to 63 cycles. A module can only begin execution upon reception of the last flit in the packet. In the simulation, we do not implement a real M-JPEG encoder. We only output a packet at a specific moment in time. A packet may however miss its deadline and the output of the module should consequently also be delayed. However, due to the complexity of such a simulation a module will still output at the specified given time. The delayed message will however be marked as infeasible as it did not arrive before its deadline.

Only the image blocks data are transmitted between the nodes. We assume that the receiver contains the information needed for storing the header for the M-JPEG encoder. An M-JPEG header includes the quantization tables and Huffman table. For our simulations we further assume that we have a link width of 4 bytes. A flit is then set to 32 bits. An 8*8 block is consequently sent as a packet of 18 flits. 16 flits are reserved for the payload and two flits are used for the header and tail respectively. LBU calculations after the simulation and feasibility tests ignore the extra header and tail flits.

Since the modules in the encoder are connected in a pipeline fashion they will have different starting times. The 2D-DCT, quantization and encoding module will still have the same number of cycles between outputs to conform to the slowest module. After the first block is received at the receiver, each module in the pipeline will output one block at the same time. The period of the encoding module is four times that of the other modules in the pipeline as a packet contains 4 blocks of data. The period of the slowest stage in the pipeline needs to be calculated. In the M-JPEG encoder the slowest module is the 2D-DCT which has a total latency for a block of 135 cycles. The blocks also experience latency due to the message communication. The period between message transmissions will be equal to the message communication between the sender and the 2D-DCT and the execution time of this module. As the number of hops varies between the 2D-DCT and the sender, the total latency of the 2D-DCT module will vary. We will have a minimum communication latency of 18 cycles (a packet contains 18 flits) assuming the switch delay and routing delay is one cycle. The total minimum latency will then be 153 cycles. If we assume a not so stringent schedule for this module we can set the maximum allowed latency to 320. The sender, 2D-DCT and quantization modules will all have a period between outputs of 320 cycles. For the encoding module we will consequently have a period of 1280. The first output block will be sent at the receiver at
time 1280. From Table 18 we can see the message attributes that are outputted from the modules. It must be mentioned that LBU calculations after the simulation and feasibility tests do not consider the extra header and tail flits. The LBU calculations uses the payload as the size.

<table>
<thead>
<tr>
<th>Message from</th>
<th>Deadline (in cycles)</th>
<th>Period (in cycles)</th>
<th>Size (in flits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>185</td>
<td>320</td>
<td>18</td>
</tr>
<tr>
<td>2D-DCT</td>
<td>254</td>
<td>320</td>
<td>18</td>
</tr>
<tr>
<td>Quantization</td>
<td>256</td>
<td>320</td>
<td>18</td>
</tr>
<tr>
<td>Encoding</td>
<td>1280</td>
<td>1280</td>
<td>[4-14]</td>
</tr>
</tbody>
</table>

Table 18: Message stream attributes for M-JPEG real-case

The start-up time needed before we have the maximal throughput is ignored. All the nodes will therefore output at simulation time 0. In order for this to work, we need to output the first two blocks from the encoder instead of the regular four for the first output. This is however ignored for our simulations. For our simulations we investigate the network when the network utilization is at a maximum. We therefore do not need to consider the start-up time.

A pipeline is mapped randomly onto the mesh. Blocks that are bound on one pipeline, are not allowed to use modules of other pipelines even if they are closer. The sender and receiver are identical for all pipelines. A message is not allowed to be sent to itself so the 2D-DCT, quantization and encoding modules in the pipeline will be mapped on distinct nodes. The 2D-DCT, quantization and encoding modules from another pipeline however may be mapped on top of a previous one. Pipelines are created randomly until we reach specified network utilization. Like for the random message set simulations we specify a threshold from 0.1 to 0.9. Pipelines are created until the network utilization of all the generated pipelines exceeds the threshold. These streams are then sent to one of the feasibility models or directly to the OMNeT++ simulator.

Each parallel pipelined JPEG encoder is only feasible if all the modules pass the feasibility test. A missed deadline in the feasibility test makes all the streams in the same pipeline infeasible. In order to make the tests less complex, we incorporate a priority scheme that gives all the message streams in the pipeline a priority in the same priority
interval. A pipeline is given a priority interval based on the sum of the H3 for all the streams in the pipeline. The pipeline with the lowest sum will be given the highest priority interval. As an example of 40 streams, the pipeline with the lowest H3 sum would be given the highest interval from 36 to 39. The stream in the pipeline with the lowest H3 receives highest priority in the interval.

An example of the architecture is illustrated in Figure 60. N parallel pipelined M-JPEG encoders compute frames in parallel. The number n is defined by the given LBU threshold specified by the user. In Table 19 we see the maximum communication allowed for a message and the latency of the next module for the first two pipelines. For Channel A we can only have a maximum communication latency of 185 in order for the 2D-DCT to be able to output at a period of 320. The deadline of the message streams from the encoding module equals the period. There is no processing done in the receiver so we can have a maximum latency equal to the period. A message must however be received and stored before the next one is received.

Figure 60: N pipelined parallel M-JPEG encoder architecture example
### 6.4.5 Simulations

A feasibility test followed by an OMNeT++ simulation and simulations without any feasibility tests are performed on a variable number of random generated parallel pipelined JPEG encoders depending on the LBU threshold specified by the user. For the tests we assume routing delay to be 1, number of virtual channels to be 4 and number of cells in a virtual channel lane buffer are 2. The network topology is a 3*3 mesh network.

#### Feasibility Testing $U_{LL}$, $U_{BDG}$ and $U_{CT}$ and $U_{OE}$

In Figure 61 we have plotted the LBU after the LL, BDG and CT feasibility tests. The OMNeT++ LBU is also shown. As explained earlier the system will never reach a steady state. The simulation time for the $U_{OE}$ is therefore set to 3840 which is three times the longest period. At an average offered traffic of 0.45 we get an average LBU of the BDG and LL of 0.17. The CT will have the largest number of feasible parallel pipeline encoders and therefore also the largest average LBU of 0.20 at offered traffic 0.45. The LL and CT model have average LBU that are 38.7 and 44.2 percent of the offered traffic at 0.45 respectively. If we perform an OMNeT++ simulation on the offered traffic we will get a high LBU of 0.42 after ended simulation. As we can see in the figure, the curves for the LL and the BDG are identical. The reason for this can be found when we look at the period of the message streams in the network. All the periods are equal or they have equal proportionality to each other. There are no possibilities for a scheduled slot in an indirect contention to be freed. The BDG model can not free up any slots, and it is conformed to the LL model. An average LBU increase of 5.5 percent as compared to the LL model is experienced when we use the CT feasibility test when we look at the difference to the offered traffic. There are no indirect contentions that the CT model can
take advantage of, but the model utilizes network disjoint messages to produce a higher LBU in comparison to the LL and BDG models. The average LBU after the CT feasibility test is for the real-case study much lower than the actual LBU from the offered message streams. One reason for this is the use of an infinite indirect contending message range. All indirect contending messages at any level are incorporated into the feasibility test of one message. An optimization of the CT model to only include 2 levels of indirect contention gave us a LBU of which is identical to having no tests performed on the message set. All the messages are feasible according to the test. The optimized feasibility test experiences the same average of 7 percent infeasible pipelines as UOE. This shows that only 2 indirect levels for the CT is too optimistic. Knowing the right number of indirect levels is difficult to uncover as each message stream may have a different need of the number of indirect levels. The large gap however between UOE and feasible traffic largely comes from the fact that we are operating on parallel pipeline encoders. A parallel pipeline encoder consists of 4 message streams. A single infeasible stream will make the complete pipeline infeasible, and as a result we will have a large difference between feasible and UOE. Remember that a pipeline is considered infeasible if one or more messages in the four streams miss their deadline. An increase in the time of the deadline would also close the gap between UOE and the feasibility tests.

Figure 61: Generated traffic vs. link bandwidth utilization for the real-case study
**OMNeT++ Execution for U_{OE LL}, U_{OE BDG}, U_{OE CT} and U_{OE}**

OMNeT++ simulations with a cycle time of 3840 are performed on the feasible M-JPEG pipelines after LL, BDG and CT model testing. In addition a graph of the LBU when no feasibility tests are performed is generated. As we want a maximal throughput from the pipelined JPEG encoder architecture, we look at the number of not feasible pipelines. A pipeline is infeasible if one or more messages in the pipeline miss their deadlines. As shown in Figure 62 all the pipelines for the LL, BDG and CT are feasible also after we input the accepted pipelines into the OMNeT++ simulator. In the figure the LL, BDG and CT all have curves plotted on the X-axis. We have no missed messages. For the simulation runs for the offered traffic were all pipelines are set feasible we start to experience missed deadlines after 0.2. If we do not perform a feasibility test on the offered traffic, we will have an average of 7 percent infeasible pipelines at an LBU of 0.45. The test shows that there are no missed messages for all the feasibility tests at 4 VCs and 2 Cells. The CT showed the best results with a 5.5% increase in LBU as compared to the LL model for the offered traffic. It further showed that without a feasibility test we would begin to experience messages missing their deadlines from an LBU of 0.2. At an LBU of 0.45, 7% of the parallel pipeline encoders will have one or more missed messages in their four streams.

![Figure 62: Link bandwidth utilization vs. number of infeasible pipelines for the real-case study](image-url)
Chapter 7

Conclusion

Real-time feasibility tests are desired by NoC designers in order to predict the feasibility and practicability of a NoC mapping. It is essential that a feasibility test results in a high percentage of the actual link bandwidth utilization and a high pass ratio. The efforts needed by the designer in order to improve the link bandwidth utilization are reduced when a feasibility test conforms to these criteria.

In this study, there was successfully developed a wormhole switched network simulator which was employed in order to evaluate the LL, BDG and CT feasibility models. The network simulator was developed with the use of OMNeT++ that proved to be a very flexible simulation tool. With OMNeT++ we could simulate both fast command line simulations as well as graphical user interface simulations that were used for testing, debugging and demonstrational purposes. The network simulator was validated for correctness and simulations show that it has a base latency of 12.5 cycles and saturation point at a link utilization capacity of around 0.6.

We can draw the following conclusions from the evaluation of the three feasibility tests as follows:

- For uniformly distributed random message streams simulations at link utilization below 0.4 and with 4 virtual channels and 2 cells per VC, the feasibility tests LL, BDG and CT result in 93.8%, 94.2% and 96% of the link utilization, compared with that from network simulation. The CT has a 3.9% and 3.2% improvement over the LL and BDG respectively. The difference between the LBU of BDG and LL is only 0.6%.
- An M-JPEG real-case study showed an improvement of the CT over the LL and BDG of 14.4%. Not implementing a feasibility test on the real-case study would lead to missed deadlines from an LBU of 0.2.
• All the feasibility tests achieve a high degree of schedulability and no missed messages with only 4 virtual channels and 2 cells per VC.

• The CT is the preferred choice as it results in the highest percentage of the actual link bandwidth utilization and the highest pass ratio. The main reason for this is that disjoint concurrent messages have a larger impact on the results of the feasibility tests than indirect contention.

In future work the contention tree feasibility model will be implemented into a feasibility analysis flow. By using the feasibility analysis flow, we can efficiently conduct several interesting analysis by exploring the application-level, mapping-stage and architectural-level design space. The first stage in the feasibility analysis flow comprises in finding the message characteristics. As stated in this study, on-chip network communication can be related to predictable events, so the message characteristics can easily be predicted as the application to be ported is known to the designer. The second stage is to build a contention tree. Since the contention tree is affected by several design decisions such as the message mapping, priority policy and the routing algorithm, we can build different contention trees by simply exploring these possibilities. After creating the contention tree we will be able to employ the contention tree feasibility test. The feasibility test will consequently produce different pass ratios and link bandwidth utilizations based on the design decisions made. These two measures can serve as the criteria to make the design decisions. By changing either the message mapping, priority policy or the routing algorithm we will be able to improve the design criteria until we are satisfied with the end result.
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D-DCT</td>
<td>Two Dimensional - Discrete Cosine Transform</td>
</tr>
<tr>
<td>BDG</td>
<td>Blocking Dependency Graph Model</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CT</td>
<td>Contention Tree Model</td>
</tr>
<tr>
<td>FCS</td>
<td>Frame Check Sequence</td>
</tr>
<tr>
<td>FES</td>
<td>Future Event Set</td>
</tr>
<tr>
<td>H3</td>
<td>Heuristic 3</td>
</tr>
<tr>
<td>LBU</td>
<td>Link Bandwidth Utilization</td>
</tr>
<tr>
<td>LL</td>
<td>Lumped Link Model</td>
</tr>
<tr>
<td>Motion JPEG</td>
<td>Motion Joint Photographic Experts Group</td>
</tr>
<tr>
<td>NI</td>
<td>Network Interface</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>OMNeT++</td>
<td>Objective Modular Network Testbed in C++</td>
</tr>
<tr>
<td>OPNET</td>
<td>Optimized Network Engineering Tools</td>
</tr>
<tr>
<td>RA</td>
<td>Routing Arbitration</td>
</tr>
<tr>
<td>RMA</td>
<td>Rate Monotonic Algorithm</td>
</tr>
<tr>
<td>RMS</td>
<td>Rate Monotonic Scheduling</td>
</tr>
<tr>
<td>RNG</td>
<td>Random Number Generator</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>VC</td>
<td>Virtual Channel</td>
</tr>
<tr>
<td>VCID</td>
<td>Virtual Channel Identifier</td>
</tr>
</tbody>
</table>
Reference List


