Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2015

MSc THESIS

Modeling and performance analysis of a high bandwidth, low power ring interconnect

Rahul S Kukreja

CE-MS-2015-9

Abstract

As technology is improving and the performance of a single core has reached its peak performance, Multicore Systems on Chip have emerged as the trend of System on Chip designs to meet the performance requirements of high throughput embedded applications. The communication infrastructure (interconnect) of such systems are as vital as its various other computational and storage units. A good design of the interconnect plays a significant role in improving the performance of the system. Bandwidth, area and power requirements of the system make the interconnect design a challenging task.
At Intel, heterogeneous Multicore System on Chips are designed for imaging applications. The current system bus based interconnect used in these systems do not meet the performance, area and power requirements of future generation chips. Furthermore, it suffers from being fully connected. For this reason, the interconnect design is migrating to a ring based Network on Chip interconnect. This thesis implements a flexible framework to test and validate the ring interconnect (RI). Using this framework, one can analyze the response of the ring infrastructure for different topologies, reservation mechanisms and traffic scenarios and then configure the RI for a real world traffic scenario. We propose distinct RI configurations to meet the requirement of such scenario. Furthermore, this framework will allow Intel to verify if the infrastructure fulfills the required performance of imaging applications in the pre-silicon stage.

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Modeling and performance analysis of a high bandwidth, low power ring interconnect

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

Embedded Systems

by

Rahul S Kukreja
born in Bangalore, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Modeling and performance analysis of a high bandwidth, low power ring interconnect

by Rahul S Kukreja

Abstract

As technology is improving and the performance of a single core has reached its peak performance, Multicore Systems on Chip have emerged as the trend of System on Chip designs to meet the performance requirements of high throughput embedded applications. The communication infrastructure (interconnect) of such systems are as vital as its various other computational and storage units. A good design of the interconnect plays a significant role in improving the performance of the system. Bandwidth, area and power requirements of the system make the interconnect design a challenging task.

At Intel, heterogeneous Multicore System on Chips are designed for imaging applications. The current system bus based interconnect used in these systems do not meet the performance, area and power requirements of future generation chips. Furthermore, it suffers from being fully connected. For this reason, the interconnect design is migrating to a ring based Network on Chip interconnect. This thesis implements a flexible framework to test and validate the ring interconnect (RI). Using this framework, one can analyze the response of the ring infrastructure for different topologies, reservation mechanisms and traffic scenarios and then configure the RI for a real world traffic scenario. We propose distinct RI configurations to meet the requirement of such scenario. Furthermore, this framework will allow Intel to verify if the infrastructure fulfills the required performance of imaging applications in the pre-silicon stage.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2015-9

Committee Members :

Advisor and Chairperson:	Koen Bertels, CE, TU Delft
Advisor:	Carmen Garcia Almudever, CE, TU Delft
Member:	Zaid Al-Ars, CE, TU Delft
Member:	Dick Epema, PDS, TU Delft
Member:	Giuseppe Garcea, ICG, Intel

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

CB Completion buffer

DUT Design under test

FIFO First in first out buffer

IPU Image Processing Unit

MCSoC Multi-core system on chip

MBS Maximum burst size

MIMT Multiple initiators, multiple targets

MIST Multiple initiators, single target

NoC Network on chip

RAT Reserve again Threshold

RB Reservation Budget

RI Ring interconnect

ROB Re-ordering buffer

SIST Single initiator, single target

SRMD Single read request, multiple data

TDP Total number of data packets

TI Traffic initiator

Acknowledgements

I would like to thank my entire family in India for always being there for me. A special thank you to my parents, sister, Brandy and Django (my pet dogs) for their unconditional love and everlasting support.

Special thank you to my professor Koen Bertels for his invaluable advice and support. I am very grateful to Carmen G. Almudever for her patience and guidance while reviewing my report.

I really appreciate the support I received form all my colleagues at Intel. I am grateful to Fiona Chua, Giuseppe Garcea, Alessandro Paschina and Cocco Mauro for their guidance and support. A special thank you to Giuseppe Garcea for his guidance and valuable time throughout my thesis.

I would like to thank all my friends at Eindhoven for making my life so easy and enjoyable away from home.

Rahul S Kukreja
Delft, The Netherlands
September 24, 2015

Introduction 1

With single computational cores having reached their peak performance [1] and the transistor density of chips still doubling every couple of years [2], system designs have shifted focus from improvements in instruction-level parallelism towards throughput-oriented energy efficient architectures [3]. Multicore System on Chips (MCSoCs) are designed to meet today's needs of high-bandwidth, compute-intensive applications by taking advantage of the improvement in technology. By adding many resources such as general purpose processors, specific IPs, etc to build an MCSoC, its communication infrastructure design becomes a crucial issue. Applications are broken down and mapped onto different computational elements of the system to take advantage of the available resources on MCSoCs. Synchronization and communication between the different computational elements are required to run the application successfully. Thus, the choice of interconnect greatly influences the system performance. Power and area constraints of embedded applications do not always allow the optimal design of an interconnect to achieve the best possible performance. Hence, depending on system requirements, interconnects are designed with trade-offs between performance, energy and area efficiency.

The Imaging and Camera Technologies Group (ICG) at Intel provides Image Processing Units (IPUs) for mobile imaging and video applications. The IPUs are heterogeneous in nature. They contain flexible general purpose DSPs along with algorithmic specific IPs. The algorithmic specific IPs offer more computation power and are energy efficient as they are customized for specific tasks. As the computational needs of image processing applications are growing, and technology is improving, the IPUs highlight the need for a new interconnect architecture and protocol to support layout robustness, low power consumption and high bandwidth requirements. The current system bus interconnect, and its protocol used by IPUs are not designed for these requirements. It suffers from the structure being fully connected (star-like). This complicates the back-end design due to a large number of wires and their long lengths. The long length of wires requires the back-end design to insert more pipe-stages to meet timing. This causes an increase in latency, area and power. Furthermore, as the size of components on an IPU are shrinking, the current interconnect is becoming pin-constrained. The interconnect's perimeter can't be reduced more because of the number of pins the interconnect requires.

The ring interconnect (RI), and its protocol are designed to solve the issues mentioned above. Also, the new design provides:

- **Scalability**. Scalable in terms of data width and address width.
- **Composability**. Parts of the system can easily be removed and added.
- **Quality of Service (QoS)**. Supports a higher number of devices. Also, allows giving priority to certain devices compared to others.

- **Hierarchical structure**. Allows a main ring (called the high ring) to be connected to multiple other rings (called low rings).
- **Power management**. Allows powering down low rings when they are not in use.
- **Area reduction**. Since the interconnect structure is ring-like and not star-like, the number of wires and their corresponding lengths are reduced.

1.1 Problem statement

The thesis aims at:

- Modelling the proposed ring interconnect and its protocol.
- Providing a flexible platform to simulate the ring interconnect to validate its protocol and measure its performance.
- Exercising the platform to identify the optimum configuration of the ring interconnect for a real-world use case.

1.2 Thesis Organization

The thesis is structured in 6 chapters. Chapter 2 introduces MCSoCs, classifies the interconnects they use and review a few interconnects proposed in the literature.

Chapter 3 provides the specification of the ring interconnect. It gives an overview of the protocol used by the ring interconnect and briefly explains the different type of nodes present in it. It also describes the Traffic Initiator, a model that is used to test and stress the RI.

Chapter 4 explains how the ring interconnect, and the Traffic Initiator are modelled. It also describes how the test environment is setup and simulated using Intel's in-house simulator.

Chapter 5 preforms Design Space Exploration (DSE) of the ring interconnect. The behaviour of the ring interconnect to changes in its different parameters is analyzed. The results of the DSE are then used to determine the optimal configuration of the ring interconnect for a real-world use case.

Finally, Chapter 6 draws the conclusions and highlights unexplored aspects that will further improve the performance of the ring interconnect.

State of the Art 2

This chapter introduces multicore system-on-chips (MCSoCs). It describes the need and different types of such MCSoCs as well as the importance of the on-chip communication infrastructure (interconnects) they use. It also highlights the basic types of interconnects that are in use today along with their corresponding advantages and disadvantages.

2.1 Multicore Systems

An MCSoC is a system that has more than one computational element on a single chip. Figure 2.1 depicts how such a system looks like [4].

Along with other components (such as I/O devices), such systems have three main components:

- **Computational elements**. Computation elements of such systems can range from general purpose processors to application specific Intellectual property (IP) cores.

- **Memory**. Memory is used by computational elements to store both instruction and program data.

- **Interconnect**. Computational elements use the interconnect to transfer data between each other and memory. The interconnect is mainly defined by the network topology and the protocol it uses. The network topology describes the connections between the computational elements and memory whereas the protocol defines the mechanism of data transfer between them.

The following subsection classifies MCSoCs based on two different taxonomies.

2.1.1 Classification of Multi-Core System on Chips

Depending on the type of computational elements used, such systems are classified into [5]:

1. **Heterogeneous MCSoCs**. The computational elements can be of different types ranging from general purpose processors to application specific hardware accelerators. Compared to homogeneous systems, these systems are generally more efficient in terms of computation, area occupation and energy consumption while they still offer a significant degree of flexibility [6].

2. **Homogeneous MCSoCs**. All the computational elements in such systems are identical. The commonly used type of computational elements are general purpose

processors. Hence, such systems have a greater degree of flexibility and scalability compared to homogeneous systems [7].

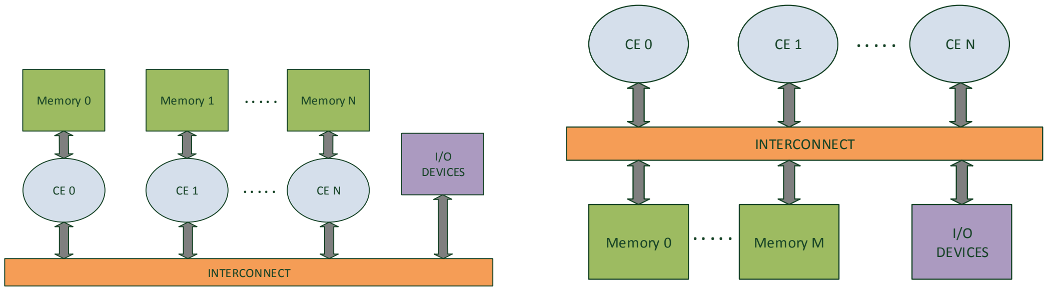
Depending on the type of memory used, such systems are classified into [4]:

1. **Distributed memory MCSoCs**. Figure 2.1b shows a distributed memory MCSoC. Every computational element has its local memory. They communicate with each other by passing information through the interconnect. Scalability of such systems greatly depends on the interconnect they employ.

2. **Shared memory MCSoCs**. Figure 2.1a depicts a shared memory MCSoC. All the computational elements have the same memory space. Computational elements use the interconnect to load or store data in memory. Depending on the physical location of memory, such systems can be further classified into:

   (a) **Uniform memory access systems**. The computational elements of such system can use the shared memory equally. The access time to a memory location by any computational element in the system will be the same. Such systems are generally not scalable because increasing system size increases the latency of the interconnection network, which impacts performance.

   (b) **Non-uniform memory access systems**. In such systems, memory is physically distributed among computational elements. Thus, a computational element takes less time to access a local memory location compared to a remote memory location.



(a) Shared memory MCSoC

(b) Distributed memory MCSoC

Figure 2.1: MCSoC block diagram. Green and purple rectangles represent memories and I/O devices, respectively. Blue ellipses are the computational units and the orange rectangle is the interconnect.

2.2 On-chip Interconnects

The interconnect is the backbone of the chip which components use to communicate. With chips now days being able to accommodate billions of transistors, the following issues with the deep sub-micrometer (DSM) technology need to be considered while designing an interconnect:

1. **Electrical wires**. The following problems arise with electrical wires:
   (a) Relative to gate delay the delay of fixed length wires continue to increase [8]. Length of global wires doesn't scale with technological improvements, thus increasing the latency involved with respect to the global communication. Repeaters are used to reduce global wire transmission delay. But they come at the cost of increased chip area and power consumption [9].
   (b) Signaling factors like crosstalk noise, electromagnetic interference (EMI), and non-determinism due to process, die and temperature variations [10] increase the probability of transmission errors.

2. **System synchronization**. With increasing clock speeds and the number of components on a chip, the demand from the clock tree to maintain a globally synchronized clock is challenging. The clock tree needs more power and area on the chip to fulfill this requirement [11]. Also, with smaller clock cycle times, the clock skew time relative to the total cycle time increases.

3. **Number of computational elements**. The number of computational elements on an MCSoC is growing dramatically [12]. Increasing the number of computational elements, increases the requirements of the interconnect such as low latency and area efficiency [13]. If interconnects aren't fast enough, even though the computational units of such systems are fast enough to process data, they spend most of their time waiting for data. Hence, poor interconnect designs can slow down systems and pose a serious bottleneck, especially for big data applications.

4. **Thermal management**. Increasing transistor densities on chips imply higher power consumption and thereby an increase of chip temperature. High power consumption makes the temperature of the chip one of the main limitations of MCSoC evolution [14].

5. **Design cycle time**. With technological improvements, designs of MCSoCs are becoming more and more complex. A design methodology is required which allows reuse of components and allows design effort to scale linearly with system complexity [15].

These factors make the design of high performance, low area, energy-efficient, scalable and reliable interconnect challenging.

Characteristics of interconnects greatly depend on the application. In the following subsections we will classify and describe some interconnects that have been proposed and used.

2.2.1 Classification of interconnects

The following subsections categorize interconnects based on different taxonomies.

2.2.1.1 Network Topology

Based on how the different components of a system are connected, interconnects can be classified as [16]:

1. **Shared-Medium interconnect**. In such MCSoCs, all the computational elements use the same communication medium to communicate with each other and memory. Buses and directly shared memory are classic examples of such interconnects. Directly shared memory systems consist of many processing elements directly connected to the ports of a memory component. Buses connect communicating nodes (can be a computational element or memory) using the same set of wires. The nodes communicate with each other by a bus protocol. Such systems are not scalable and perform well only when they use a few number of communicating elements [17].

2. **Direct interconnect**. These type of interconnects have communication links that directly connect one node to another. Some examples are point-to-point interconnects and Network on Chips (NoCs). Point-to-point networks connect all the nodes to each other via dedicated communication links. NoCs use the ideas of large-scale networks that are scaled down for MCSoCs. In NoCs, the nodes of the system are interfaced to routers. Routers of nodes are connected to each other via communication links. Using this setup, data is transferred from a node to the other in terms of packets. Such interconnects can prove to be expensive in terms of area and energy consumption depending on factors such as the number of communication links they use, the complexity of the routers, etc. Some examples of NoC topologies are ring, mesh, torus, tree, etc.

3. **Indirect interconnect**. These interconnects connect node to switches. Switches are connected to one another via communication links. Crossbars and Butterflies are examples of this type of network. A crossbar uses switches to connect its inputs to its outputs. Butterflies have multiple stages of crossbars. Such interconnects attempt to reduce the area and energy consumption by reducing the number of communication links however they require a complicated switching logic.

4. **Ad hoc interconnect**. Such interconnects are customized for particular applications to improve bandwidth, reduce area and power consumption. They are a combination of shared-medium, direct and indirect interconnects; taking the advantages of each type.

2.2.1.2 Protocol

The protocol of the interconnect defines the mechanism it uses to transfer data between the nodes connected to it. It is defined by the **Switching technique**, **Routing strategy** and **Flow control mechanism**.

- The **Switching technique** decides how data flows from the source node to the destination node. Switching techniques can be classified as [18]:

  1. **Circuit switching**. In this technique, switches are multiplexed such that there is a direct physical path from the source to the destination node. After a connection is established, the source node transfers data to the destination node. Till the source node finishes data transfer, the path (or part of it) can't be used by another node. Thus, the interconnect pattern at a given point in time depends on the nodes that are communicating. Crossbars, buses, and butterflies use this switching technique.
  2. **Packet switching**. A data message is transferred from the source to destination in terms of fixed sized packets. NoCs use packet switching techniques.

- The **Routing strategy** selects the path used to transfer data from source to destination. Routing strategies are classified as [19]:

  1. **Source routing or distributed routing**. In source routing, the source node decides the path of data transfer. In distributed routing, every intermediate node between the source and destination influences the path of data transfer.
  2. **Deterministic routing or Adaptive routing**. While using deterministic routing, data transfer between a source and destination always takes the same path. Whereas in adaptive routing, the path from source to the destination depends on network conditions such as the amount of traffic congestion.
  3. **Minimal routing or Non-minimal routing**. Minimal Routing always uses the shortest path to transfer data from a source node to destination node. On the other hand, Non-minimal routing allows data to be transferred over paths longer than the shortest path from source to destination.

- The **Flow control mechanism** determines how resources of the interconnect such as channel bandwidth are allocated to transfer data. It decides how data is transferred using the selected routing strategy and switching technique. It also defines how the interconnect behaves in congestion situations, for example if it drops data or not. Flow control techniques interconnects can be of two types [20]:

  - **Bufferless**. Such interconnects don't use buffers to save on cost. Thus, in case of channel contention, data has to be misrouted or dropped. Circuit switching interconnects use bufferless flow control.
  - **Buffered**. Packet switching interconnects use buffered flow control. They use buffers to store data packets in situations of contention. Hence, data packets don't have to be routed or dropped. This causes an increase in the latency of transporting data packets. Data packets are composed of multiple flits. A flit is the elementary packet on which flow control is performed. Flits are in turn made up of one or more phits. A phit is the unit of data that is transferred from one node to the other in a given cycle. Some types of buffered flow control mechanism are:

1. **Store and forward**. A data packet is transferred from one node to the next only if the receiving node has enough buffer space for the entire packet. A node can forward a data packet only when it has been received in its entirety. This technique is not often used because of large buffer requirements.
2. **Virtual cut through**. This technique forwards flits of a data packet to the next node when the next node has enough space to receive the data packet. The source doesn't wait for the complete data packet to arrive before beginning the transmission. It reduces latency over the store and forward technique and but still has the same buffer requirements.
3. **Worm whole routing**. In this technique, a flit of a data packet is forwarded if the receiving node has enough space to store a flit, thus reducing buffer requirements. Flits of a data packet can take different paths to reach it's destination which increases the probability of link blocking and congestion.

As we mentioned previously, in this thesis we will focus on ring interconnect: a NOC network that uses the packet switching technique, the store and forward flow control mechanism and the source routing strategy which is deterministic and minimal.

2.2.2 Evolution of interconnects

Interconnects have been evolving with the improvement in technology and increasing design complexity. Figure 2.2 [21] summarizes how interconnects have evolved. Initially, SoCs consisted of very few components such as a processor, a memory module, and I/O interfaces. Dedicated point to point links were used to communicate between these components. With the increase in component density on a chip, these links proved to be expensive in terms of area and power consumption. Hence, SoCs migrated to buses. As the number of components on a chip grew further, limitations of buses because of contention and long wires were exposed. System designs moved to hierarchical buses as a temporary solution. Complex designs and non-predictable latencies still highlighted the need for a better solution. Crossbar based interconnects were then adopted for communicating between different components. They offered better bandwidth since they have dedicated paths between communicating nodes. But, the area cost of a crossbar increases quadratically as the number of communicating nodes increases. Thus, using the concepts of networking on distributed systems, NoCs were then designed to make systems easily scalable while supporting their performance requirements. The inherent design and structure of NoCs help in alleviating some of the problems faced with the DSM technology. Since NoCs transfer data in terms of packets, they cause an increase in latency even though they allow messages to be transferred parallelly. They can also prove to be expensive in terms of implementation depending on a number of factors like router complexity, buffer sizes, the number of channels used, etc. Hence, in recent years system designs have moved towards ad hoc or hybrid interconnects. These interconnects combine the advantages of the other types of interconnects depending on the system requirements.

Thus, the choice of interconnect mainly affects three factors of a system: its performance, scalability and cost (area). Figure 2.3 compares the commonly used interconnects in terms of these factors [22].

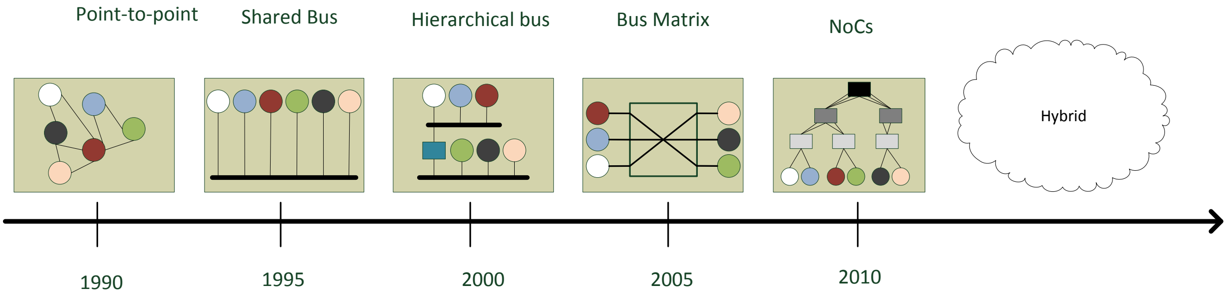


Figure 2.2: Evolution of on-chip interconnects

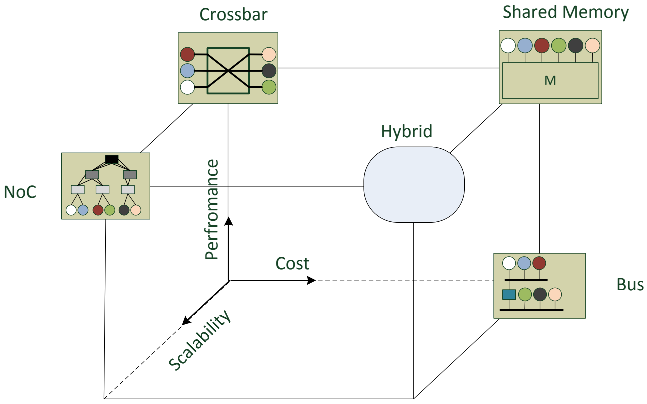


Figure 2.3: Comparison of on-chip interconnects

2.3 Related Work

As it is anticipated that future generation chips will use hundreds of high throughput oriented communicating components on an MPSoC, NoCs are designed to provide a scalable interconnect that can meet the requirements of such systems [23]. They use packets to transfer data from the source node to the destination node via a communication infrastructure that consists of switches and links. This infrastructure enables the communicating hardware blocks to be designed independently and then connected in lego fashion [24]. NoCs are also capable of supporting the QoS requirements of heterogeneous systems in which different components need different levels of performance from the interconnect [25]. Also, because of their inherent structure and design, they have the potential to overcome many of the limitations observed with the DSM technology [21].

Examples of well-known topologies of a NoC are the ring, 2D-Mesh, torus, tree, etc. [26]. The performance and power trade-off of a system is greatly influenced by the NoC topology [27]. For example, a torus offers less latency compared to meshes but its cost in terms of number of wires is higher [28]. In this section, we review a few interconnects that are based on the ring architecture have been proposed in the literature. Since rings are simple, power-efficient and area-efficient compared to other NoC topologies, many industrial designs such as the IBM Cell processor [29], [30] and high-end graphical processors from ATI [31] employ multiple rings as their interconnect networks.

Chris et al. [32] proposes a high performance and energy-efficient network architecture that makes use of hierarchical rings. Since the design is ring-based, the router design of the NoC is very simple. Compared to traditional mesh-based networks, this reduces the amount of energy the interconnect consumes and die area it occupies. By using hierarchical rings, they demonstrate that the network maintains the bi-sectional width of a mesh topology. Thus, the performance of the network scales well by using energy and area efficient routers. For a set of workloads, they then show that the hierarchical rings give an average network power reduction of 52.4% compared to a conventional buffered mesh and offer an average speed up of 0.6%.

In [33], the authors use simple and hierarchical rings along with mesh networks to overcome the limitations of the mesh network (communication latency scalability and concentration of traffic in the center of the mesh). They partition meshes into sub-meshes, which are connected via a local ring. All the local rings are then connected via a global ring. They explore 2 different topologies to reduce global latencies of traffic. The first topology is a slotted ring implementation and the second one uses worm-whole routing and virtual channels to provide flexibility and improve performance. Their simulation results show that the proposed topologies decrease the latencies and hop counts of global traffic when the number of nodes is lesser than 44. They attribute this to the fact that the amount of traffic passing through the ring interconnect increases quadratically with the number of nodes, and at N = 44, the hierarchical rings reach a saturation point because of which throughput degrades.

Yang et al. [34] propose an energy-efficient asynchronous ring bus to overcome the power requirements of the clock tree in synchronous designs. In this case, nodes are connected via an uni-directional ring that is 33 bits wide and they use data packets to setup dedicated paths between them for communication. Furthermore, every node has a built-in collision solver to avoid situations of dead-lock and arbitrate access to the communication channels in situations of contention. They simulate the ring for different arbitration strategies and traffic patterns. They show that performance of the ring is strongly dependent on the type of arbitration strategy used for a particular traffic pattern.

In [35], the authors propose 2 variants of a control network to configure circuit switched routers. The circuit switched routers transfer streaming data for Hiper-LAN/2, UMTS, Bluetooth and digital radio applications. The data transferred by the control networks configure the circuit switched routers to setup dedicated links between communicating nodes. The dedicated links are then used to transfer data with guaranteed throughput and latency requirements. The authors analyze a bus-based and ring-based control network and compare their performance. In the bus architecture, all the communicating nodes and a arbiter are connected to a bus. The arbiter allows the nodes to use the bus in a round-robin fashion. Thus, at a given point of time only a single data packet can be transported by the bus. In contrast, the ring architecture uses a bidirectional slotted ring. Then, for N nodes for N nodes it can have 2 * N data packets in flight at a given point in time. The authors then compare the 2 proposed networks in situations of uniform traffic for a different number of communicating nodes. They conclude that the ring offers approximately 3.4 times more throughput for the same power consumption compared to the bus. Furthermore, it consumes less energy per transported bit and 3.2 times less power for the same throughput requirements.

Motivated by how ring roads avoid traffic congestion in a city, [36] proposes a NoC called the Ring Road. The NoC consists of multiple rings that are connected via ring switching elements. Each ring contains nodes that communicate via data packets. A packet sent between 2 nodes on the same ring will never leave the corresponding ring. For communication between nodes on different rings, the ring switching elements are used to route the data packet. Thus, by placing communicating nodes optimally, the authors show that congestion of traffic in the center of the ring is avoided.

In [37], the authors propose a NoC called The Proteo for heterogeneous systems. It is a hierarchical network topology that uses a global bidirectional ring to connect the clusters of the system. The cultures typically use point-to-point networks to communicate internally. The topology, packet format, etc. of each cluster network is selected based on its performance requirements. The proposed network was the first one that implemented the Virtual Socket Interface Alliance (VSIA) standard [38] to enhance the productivity of the SoC design.

Bourduas et al. [39] propose a 2-tier hierarchical ring network to take advantages of the following properties of a ring:

- In unidirectional rings the switching logic is simple and fast. This gives the benefits of high bandwidth and low latencies.
- Rings uses point-to-point connections. Thus, the capacitive loads is small when compared to shared medium approaches, resulting in low energy consumption.
- Two-dimensional ring structure results in an area-efficient planar layout when compared to more complex multidimensional ring-based networks

They model a two-level hierarchical ring network that consists of four local rings connected to a global ring network using inter-ring interfaces. Each local ring has four stations connected it using station-ring interfaces. Each station contains a processing elements and a dedicated physical memory. On performing design space exploration to minimize energy consumption, they show that energy is minimized without adversely affecting the performance. They then use a dynamic clock throttling scheme to further minimize energy during periods of inactivity.

2.4 Summary

In this chapter, we introduced MCSoCs and provided two taxonomies based on which they are classified. MCSoCs mainly consist of computational elements, memory units and a communication infrastructure (interconnect). Based on the type of computational units they use, they can either be homogeneous systems or heterogeneous systems. Based on how memory is distributed in the system, they can be either shared memory systems or distributed memory systems.

We then highlighted the challenges interconnect designs face because of the DSM technology and system requirements, and the impact that they have on the performance of MCSoCs. After that, we provided different taxonomies based on which interconnects are classified. Based on the topology and protocol they use, interconnects can be classified in numerous categories. Also, the type of interconnect a system employs greatly depends on the system requirements. Based on the choice of interconnect used there is always a trade-off between cost (in terms of area and energy) and performance. The ring interconnect studied in this thesis is a NoC, that uses the packet switching technique and the store and forward flow control mechanism. It uses a source routing strategy which is deterministic and minimal. We continued to show how interconnects have evolved based on improvements in technology and changes in system requirements. We then compared commonly used interconnects based on cost, performance and scalability.

Finally, we reviewed some of the work done with respect to the ring interconnect in the literature. As the ring is simple in its design, area-efficient and energy-efficient it is commonly used.

3 Specification

This chapter introduces the ring interconnect (RI). It describes how it is modeled and details the test environment used to analyze it. Section 3.1 briefly explains the RI, its protocol, and components. Section 3.2 introduces the traffic initiator; a model used to test the RI.

3.1 The Ring Interconnect

A slotted RI is a direct cyclic NoC in which every node is connected to its neighboring node in a point-to-point fashion. The interconnect uses the packet switching technique along with the store and forward flow control mechanism. Figure 3.1 shows a simple RI. A node of the RI is typically either a computational unit or a storage unit. To reduce traffic congestion, optional pipe-stages (buffers) can be used to connect nodes. In this interconnection network, a data packet is transferred from one node to the other every clock cycle.

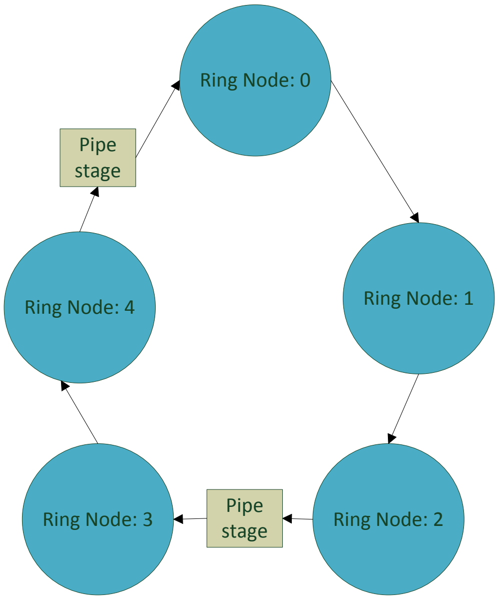


Figure 3.1: Simple RI structure

The RI can be extended to have multiple hierarchies as shown in Figure 3.2. Such networks have a special type of node called the bridge node that is a part of two simple

RIs and allows communication between them.

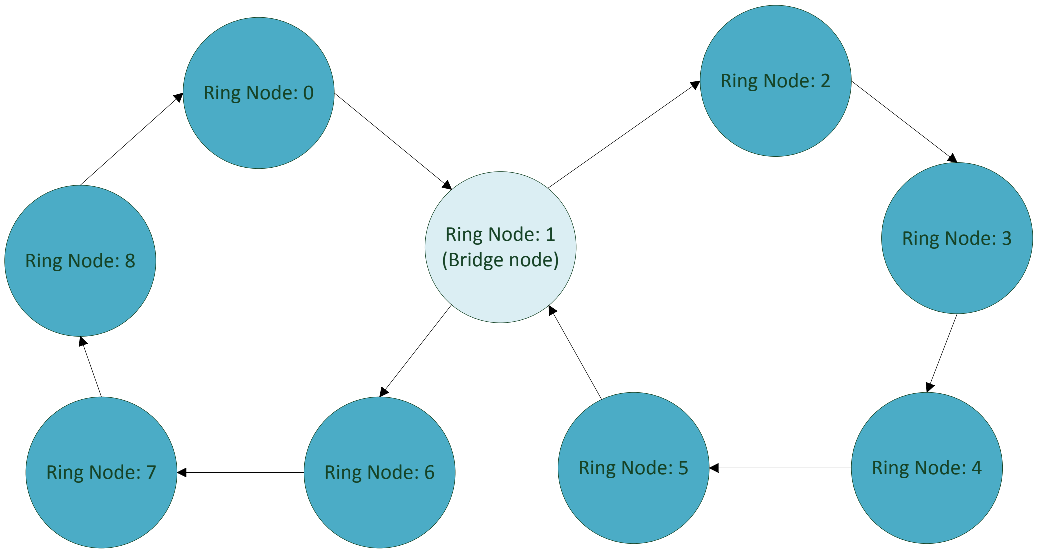


Figure 3.2: Hierarchical ring RI

RIs can be further extended to have multiple channels that are used to communicate between neighboring nodes. Figure 3.3 shows an RI with two channels between nodes. It is a bidirectional RI in which one channel is used to communicate data in the clockwise direction and the other in the anti-clockwise direction.

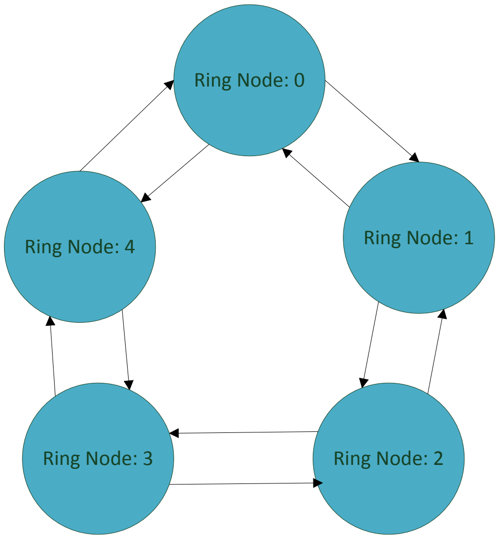


Figure 3.3: RI with 2 channels

Every clock cycle in an RI, the source node of a channel transfers a packet of data, and the destination node of the channel accepts it. This guarantees there can be no deadlocks. The Total number of Data Packets (TDP) circulating every cycle is a constant, and is equal to:

$$TDP = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C_{i,j} * (P_{i,j} + 1) \tag{3.1}$$

Where,

N is the Number of nodes.
$C_{i,j}$ is 1 if there's a channel between source node i and destination node j else 0.
$P_{i,j}$ is the number of optional pipe stages between source node i and destination node j when $C_{i,j}$ is 1.

The following sub-sections define the content of a data packet, introduce the different types of nodes used in the RI and detail their sub-components.

3.1.1 Data packet definition

Requests are transferred by nodes using the data packets circulating in the RI. Table 3.1 describes the different fields of a data packet along with the range of values they can take and their functionality.

Field	**Range of Values**	**Function**
Valid	$0 : 1$	**0** : Data packet doesn't contain a valid request. **1** : Data packet contains a valid request.
Command type	$0 : 5$	**0** : Write request. **1** : Read request. **2** : Completion. **3** : Reserved for completion. **4** : Initialization request. **5** : Clear reserved data packet request.
Source node ID	$0 : (N-1)$	ID of the node that created the request.
Destination node ID	$0 : (N-1)$	ID of the node that should consume the request.
Data	$Configurable$	A place holder for the data content that is to be stored or loaded by a node.
Address	$Configurable$	The address from which the data content is to be stored or loaded by a node.

Command order ID	$0 : (TDP - 1)$	Indicates the order in which the data packet is seen by a node. Situations in which a node is unable to consume a data packet, it associates an ID with the data packet and inserts it back into the RI. When it is able to consume data packets again, it consumes them in accordance with the value of this field. Thus, guaranteeing in order processing of data packets.
Command order ID valid	$0 : 1$	**0** : Command order ID is invalid **1** : Command order ID is valid.
Burst mode	$0 : 1$	**0** : Burst mode inactive. **1** : Burst mode active. Based on this field, read requests can be of two types: • **Single read request single data**. The data packet requests a node to load a single word. • **Single read request multiple data**. The data packet requests a node to load multiple words from incremental addresses. Thus, in response to this type of packet, the destination node inserts as many data packets of the type completion in the RI as the requested number of words to be loaded. The maximum number of requested words or maximum burst size (MBS) for this type of request is pre-defined to avoid congestion in the network.
Burst size	$1 : MBS$	Indicates the number of requested completions for a burst read request.
Completion order ID	$0 : (MBS - 1)$	Indicates the order in which a destination node responded to a burst read request. This allows source nodes of the corresponding read requests to absorb completion data packets in order.
Reserved node ID	$0 : (N - 1)$	ID of the node that the data packet is reserved for.
Reserved	$0 : 1$	**0** : Data packet is unreserved. **1** : Data packet has been reserved for a node.
Booked for bridge	$0 : 1$	**0** : Data packet is not booked for a bridge node. **1** : Data packet is booked for a bridge node.

Alert	0 : 5	**0** : Data packet contains a request that can be absorbed by a node in the RI. **1** : Destination ID of the data packet doesn't exist in the RI. **2** : Address field of the data packet is not in the address range of the destination node. **3** : A node received a completion it didn't request for. **4** : A storage node received a completion. **5** : A computational node received a read or write request.

Table 3.1: Data Packet definition

Depending on the **Valid** and **Reserved** fields, data packets circulating in the RI are used by nodes to insert requests. If a data packet contains a request, it is marked as valid, and another node can't insert a new request in it. Data packets are reserved for nodes by setting the **reserved** field and storing the ID of the node it is reserved for in the **Reserved node ID field**. A reserved invalid data packet can be used only by the node it is reserved for. This type of data packet guarantees a minimum non-zero bandwidth for nodes in the RI. Initially, all data packets circulating in the RI are unreserved and invalid. Table 3.2 summarizes which node can use a data packet depending on the values of the valid and reserved fields. How and when packets are marked as valid or invalid and reserved or unreserved is explained in section 3.1.2.

Valid	**Reserved**	**Who can use the data packet?**
x	x	Any node can use this data packet to insert its request in the RI. Also this data packet can be reserved by any node.
x	✓	It can be used only by the node it is reserved for to insert a request in the RI.
✓	x	Can be reserved for any node.
✓	✓	This data packet can only be consumed by destination nodes

Table 3.2: Data packet interpretation

A valid data packet can be interpreted as one of the following types:

1. **Initialization**. This is a special data packet that is inserted in the RI once after power-on. It is used to assign a unique ID to a node. Its data field is initialized as zero and when a node receives this type of packet, it increments the value of the data field in the packet and forwards it the next node it is connected to. It stores the incremented value of the data field as its node ID. As indicated in Table 3.1 Node ID's are used as fields of data packets to indicate the source node, destination node and the node for which the packet is reserved.

2. **Write request**. Request made by a node asking another node to store data present in the data packet.

3. **Read request**. Request made by a node asking another node to send its data from an address present in the data packet. When a source node inserts a read request into the RI, it reserves the data packet for the destination node.

4. **Completion**. It is a special write request generated by the destination node with the data for a read request it received before.

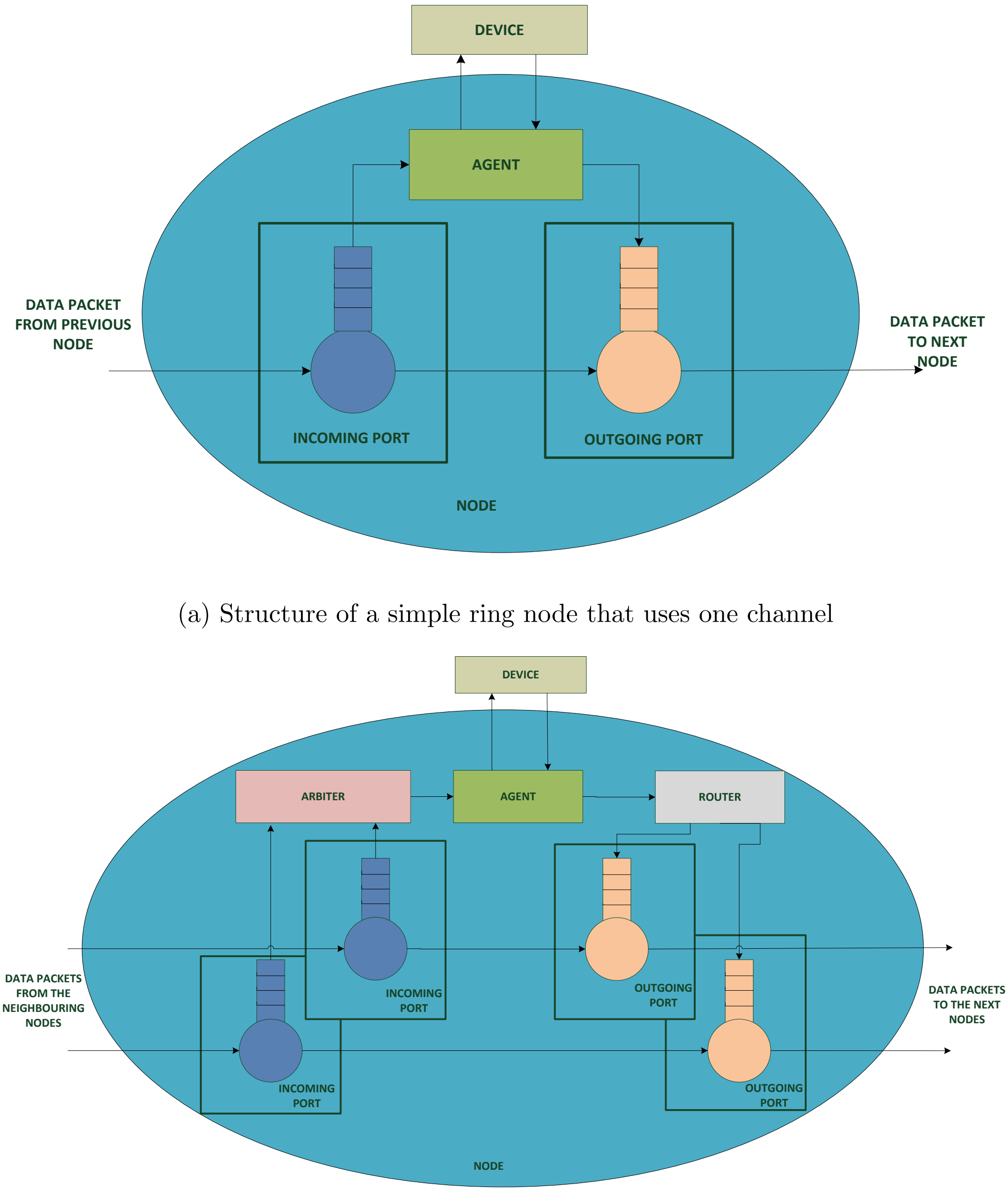
5. **Reserved for completion**. This data packet guarantees a node can insert completions for a read request it received. When a node receives a read request, it: makes a copy of the data packet, marks it as invalid, changes the command type to reserved for completion and inserts it into the RI. When this data packet arrives at the node again, and the node has a response for the corresponding read request, it marks the data packet as valid, fills the data packet with the requested data and inserts it back into the RI. While inserting the last completion corresponding to a read request, the destination node unreserves this type of data packet if it is already at the node or the next time it passes through the node.

6. **Alert**. A valid data packet that can't be consumed by any node. Such a packet has to be removed from the RI to avoid live-locks.

7. **Clear reserved data packet**. It is a data packet that asks a node to unreserve a reserved for completion data packet. When a read request for a node is an alert, this data packet is used to clear the reserved for completion data packet that was introduced into the RI as explained above.

3.1.2 Structure of ring node

Ring nodes are constructed using the following sub-components: Incoming port, Agent, Outgoing port, Router and Arbiter. A simple RI node has a general structure as shown in Figure 3.4a. For a multiple channel RI each node has a pair of incoming and outgoing ports for every channel. Figure 3.4b shows how the structure of nodes are extended when they use two channels. Note that routers and arbiters are only required in multiple channel RIs.



(a) Structure of a simple ring node that uses one channel

(b) Structure of a ring node that uses two channels

Figure 3.4: Structure of a simple RI node

Theses sub-components work as follows:

- The **Incoming port** is the interface between the RI and the node. Every cycle, it:
  - Receives a data packet from the RI.

  - Compares the node ID with the destination node ID tag of the data packet. If the data packet is for the node, it stores a copy of the data packet in its buffer. It then marks the received data packet as invalid and passes it to the outgoing port. If the data packet it received is a read request, it also changes the command type to reserved for completion before passing it to the outgoing port.
  - If the data packet is not for the node, it forwards the data packet to the outgoing port.

  As explained further in this section, there will be situations that cause the buffer(s) of the incoming port to become full. In such situations, depending on the type of node (as in Section 3.1.3), the incoming port behaves in one of 2 possible ways:

  - **Bounce with Re-ordering**. The buffer of the incoming port behaves like a circular re-ordering buffer (ROB). The purpose of the ROB is to ensure that the agent absorbs data packets in the same order the incoming port first sees them. If the ROB is full, the incoming port sets the **Command order ID valid** field and the value of the **command order ID** field with the value of the position it should have been stored in the ROB. After that, it forwards the data packet to the outgoing port. When there is space available in the ROB again, the incoming port then accepts and stores data packets in positions of the ROB based on the **Command order ID valid** and **command order ID** fields of the data packet.
  - **Bounce without Re-ordering**. The buffer of the incoming port behaves like a first in first out queue (FIFO) If the FIFO of the incoming port is full, it simply passes the data packet to the outgoing port without altering it. When it sees there is a place available in the FIFO, it accepts the data packet from the RI and stores it in the FIFO.

- The **Agent** is the interface between the node and the computational or storage unit (called **Device**). Every cycle, it handles:

  - Receiving data packets from the incoming port (or arbiter) and passing it to the device. (Sink for valid data packets in the RI)
  - Receiving data packets from the device and forwarding it to the FIFO of the outgoing port. (Source of valid data packets in the RI)
  - Protocol conversions if a device that doesn't understand the ring protocol is connected to it.

  The agent can communicate with the device in either direction simultaneously. Successful communication of data between the agent and device is guaranteed using a simple handshaking protocol. Using this protocol, the following situations can arise:

  - Depending on the state of the agent and the state of the outgoing port, the agent can back pressure the device (i.e., agent doesn't accept packets from the device).
  - The device can back pressure the agent depending on its state (i.e., the device doesn't accept data packets from the agent).
  - The agent can back pressure the incoming port (via the arbiter if used) depending on its state and the state of the device (i.e., the agent doesn't accept packets from the incoming port). This causes the buffer of the incoming port to become full.
  - The outgoing port (or router if used) back pressures the agent when the buffer of the outgoing port is full. (i.e., the outgoing port doesn't accept packets from the agent).

- The **Outgoing port** is the interface between the node and the RI. Every cycle, it accepts a data packet from the incoming port or its buffer (FIFO) and inserts it into the RI. The outgoing port is also responsible for reserving and unreserving data packets for nodes. It has a counter (*Reserved_counter*) to keep track of the number of data packets circulating in the RI that has been reserved for the node. It is allowed to reserve a data packet only when this counter value is below a threshold (Reservation Budget - RB). Thus making sure that every node can reserve only a limited number of data packets.

  Depending on the value of *Reserved_counter*, the contents of the data packet at the head of its FIFO and the one coming from the incoming port, it inserts a data packet into the RI as follows:

  - When the outgoing port receives an invalid and unreserved data packet from the incoming port and also receives a valid data packet from its agent, it inserts the data packet it received from its agent into the RI.
  - When the outgoing port receives an invalid reserved data packet or valid data packet (reserved or unreserved) from the incoming port, it inserts it into the RI. If there is also a valid data packet from the agent to be inserted into the RI, it leaves the command in the FIFO and increments a counter (*Can_reserve_counter*). Once the *Can_reserve_counter* passes a threshold (Reserve again Threshold - RAT), and the value of *Reserved_counter* is less than RB, the outgoing port can reserve an unreserved data packet (valid or invalid) that comes from the incoming port. The RAT puts a limit on the number of back to back data packets the outgoing port can reserve. After the outgoing port reserves a data packet, it increments the *Reserved_counter* and resets the *Can_reserve_counter*. By reserving a data packet, the outgoing guarantees that no other node can insert their corresponding request in the data packet when the data packet becomes invalid.
  - If the data packet is reserved for the node and the outgoing port is inserting a write request into the RI, it unreserves the data packet and decrements *Reserved_counter*.

  - If the data packet inserted is a read request, it reserves the data packet for the destination node. This guarantees that when the destination node responds to the read request with a completion, there is a data packet in the RI that will allow it to do so.
  - While inserting the last completion corresponding to a read request, if the data packet from the incoming port is of the type reserved for completion, it unreserves it. If not, it unreserves a reserved data packet for the node which is of the type reserved for completion the next time it arrives at the outgoing port.
  - When a reserved data packet for the node that is not of the type reserved for completion arrives from the incoming port and the outgoing port has nothing to insert in the RI, it unreserves the data packet and decrements *Reserved_counter*.

  When the outgoing port receives a clear reserved data packet, it unreserves a data packet of the type reserved for completion the next time it arrives at the outgoing port.

The router and the arbiter are only used when the RI uses multiple channels. They work as follows:

- **Arbiter**. The outputs of the incoming ports from all the channels are the inputs to the arbiter. The arbiter performs round-robin arbitration when there is contention between the output of the ROBs of the incoming ports. Otherwise, it just outputs the valid data packet.
- **Router**. The outputs of the router are the inputs of the buffers of the outgoing ports. The Router selects which outgoing port (or channel) the data packet from its input will be forwarded to. The channel selection is table based. In general, the table is configured such that when choosing between N channels, data packets will always take the shortest path. The table can be reconfigured for balancing bandwidth requirements.

3.1.3 Types of nodes

The RI comprises four types of nodes. The following subsections describe the functionality of each type of node and their sub-components.

3.1.3.1 Initiator Node

The device of the initiator node injects write and read requests for target nodes via the agent. In addition to the operations mentioned in the previous section, the agent performs the following for this type of node:

- The agent maintains a circular buffer to receive the completions for a read request. The expected completions from a read request is always less than or equal to

MBS. When the agent receives a read request from the device, it checks if there is enough unreserved space available in the completion buffer to store the expected completions from the request. Also, the agent doesn't allow its device to insert back to back read requests for the same target node. Since a target node would take at least N cycles to respond to a read request of N words, the agent waits for N cycles before inserting the next read request for the same target into the ring. This behaviour of a node will be referred to as "good citizen behaviour" in the next chapters.

– If there is enough space, it reserves space in the completion buffer and inserts a data packet corresponding to the request to the outgoing port. It sets the value of the **Completion order ID** field equal to the position in the completion buffer at which it expects the first completion back.

– If there is no space available in the completion buffer for the read request, it doesn't accept the command and back pressures the device connected to it via the handshaking protocol.

– When the agent receives a completion, it stores it in its buffer. Only if the buffer position that must be presented to the device contains valid data, the agent communicates the data to the device. When the device accepts the data, the agent pops the corresponding completion from the buffer. This way the agent guarantees that completions are presented to the device in order.

– If the agent receives a completion it didn't request for, it marks it as an alert and passes it to the outgoing port.

Since the device of the initiator node guarantees that completions are presented in order to the device, the incoming port uses an FIFO. If the incoming port receives a write or read request for the port, it marks the command as an alert and passes it to the outgoing port.

3.1.3.2 Supervisor Node

The RI contains one Supervisor node with the ID 0. It is a special initiator node. This node:

- Receives an initialization command from the device connected to it whenever there is a power up of the RI. The supervisor inserts this command into the RI with data field initialized as 0. When it receives the initialization data packet back at the incoming port, the data field of the command corresponds to the maximum ID of the node in the RI.

- Monitors the Destination ID tag of data packets that arrive at the incoming port. If a node with ID equal to the destination ID of the data packet doesn't exist in the RI, it removes the data packet and notifies the device connected to its agent with required debug information.

- When alert data packets arrive at its incoming port(s), it removes them from the RI and notifies the device connected to its agent with required debug information. If the received alert command was a read request, it issues a clear reserved data packet request to remove the corresponding reserved for completion data packet introduced in the RI.

The incoming port of the supervisor agent uses an ROB so that it can present alerts to the agent in the order that it sees them. This allows easy debugging when alert packets are removed from the ring.

3.1.3.3 Target Node

The incoming port of a target node receives read or write requests from initiator nodes. The port has an ROB associated to it so that data packets are consumed in the order in which they are first seen.

In addition to the operations mentioned in the previous section, the agent performs the following for this type of node:

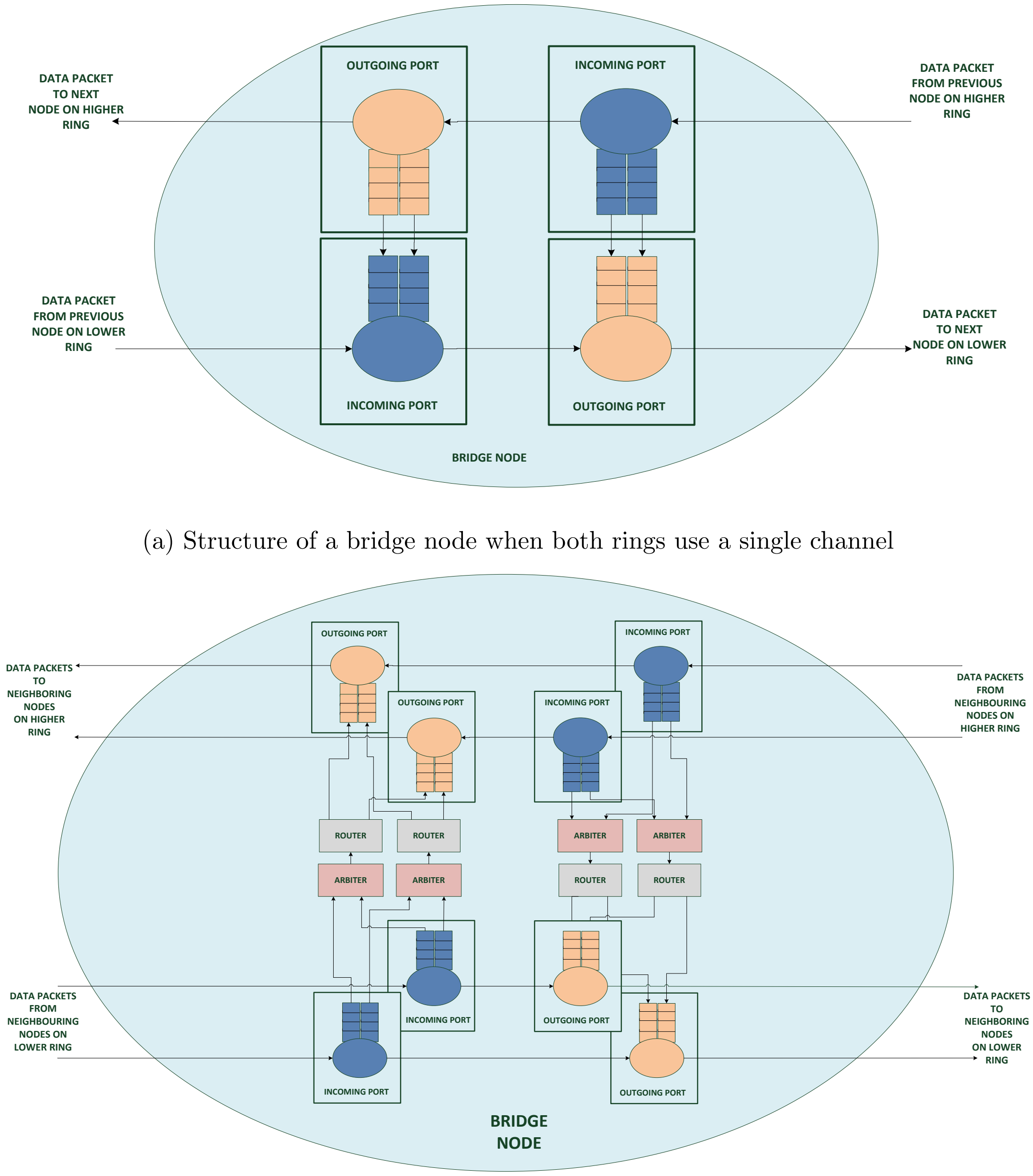
- The target agent receives requests from the incoming port and forwards them to the device.

- When the target receives a completion from the device, it creates a completion data packet to be inserted into the RI. For the first completion of the read request, it associates the value of the completion order ID it received from the corresponding read request. For further completion data packets corresponding to the same read request, it associates incremental values of the completion order ID of the read request.

3.1.3.4 Bridge node

A bridge node is a special node that connects two ring hierarchy levels: a higher ring and a lower ring. Figure 3.5a describes the structure of a bridge node used to connect two rings that use a single communication channel. On each hierarchical level, the bridge node has a pair of incoming and outgoing ports. The incoming port of one ring is connected in back-to-back fashion to the outgoing port of the other ring. The incoming and outgoing ports function as explained in 3.1.2. The ports use two buffers: one to store all requests but completions and the other to store only completions. This is done to avoid livelocks in either of the rings. Similar to simple RI nodes, figure 3.5b shows how the structure of a bridge node is extended when it uses two channels.

A bridge node always receives an initialization command from the higher ring. When it receives this command it stores the data field value as its ID (lower bound), increments the data field value and routes the data packet to the lower ring. When it receives the initialization data packet back from the lower ring, increments the data field and forwards the data packet to the next node on the higher ring. It stores the incremented value of the data field as its higher bound. Thus, the nodes with IDs between the lower bound and higher bound belong to the lower ring. The bridge node passes a command from the


DATA PACKET TO NEXT NODE ON HIGHER RING
OUTGOING PORT
INCOMING PORT
DATA PACKET FROM PREVIOUS NODE ON HIGHER RING
DATA PACKET FROM PREVIOUS NODE ON LOWER RING
INCOMING PORT
OUTGOING PORT
DATA PACKET TO NEXT NODE ON LOWER RING
BRIDGE NODE
DATA PACKETS TO NEIGHBORING NODES ON HIGHER RING
DATA PACKETS FROM NEIGHBOURING NODES ON HIGHER RING
ROUTER
ARBITER
DATA PACKETS FROM NEIGHBOURING NODES ON LOWER RING
DATA PACKETS TO NEIGHBORING NODES ON LOWER RING
BRIDGE NODE

(a) Structure of a bridge node when both rings use a single channel

(b) Structure of a bridge node when both the rings use 2 channels

Figure 3.5: Structure of a bridge node

incoming port of one ring to the outgoing port of the other ring based on the destination ID field of the data packet, the upper bound and the lower bound.

There can be a livelock in the RI when one of the rings the bridge node is connected to is full with requests for the other ring. In this scenario, the bridge will not be able to move request between rings. To prevent this situation, a bridge node will have a booked data packet for itself (**Booked for bridge node** field is set) on the lower ring. The data packet booked for the bridge node can't be reserved by any other node. It is used by the bridge node to insert read or write requests. When the data packet booked for the bridge node is used to insert a read request, this data packet is temporarily transformed into a Reserved for completion data packet (by the outgoing port of the target node) until the last element of the completion is sent by the outgoing port of the target node. The first request transferred from the higher ring to the lower ring is marked as the data packet booked for the bridge node.

The buffers of the incoming port are ROBs that apply the re-ordering scheme independently. The incoming port of the bridge node has special behavior while absorbing reserved data packets. Based on the reserved data packet absorbed, the incoming port does the following:

1. **Reserved data packet carrying a read request**. When the bridge node receives a read request from one ring for the other, the incoming port of the receiving ring makes a copy of the received data packet, marks it as invalid, reserves the data packet for itself, changes the command type to reserved for completion and forwards it to the outgoing port of the same ring. This data packet is unreserved by the outgoing port on receiving the last completion of the read request (as explained in section 3.1.2). The original data packet received by the incoming port is absorbed into its ROB. If it is booked for a bridge node, the incoming port unbooks it before absorbing it into its ROB.

2. **Reserved data packet carrying completion**. When the bridge node receives a completion from one ring for the other, the incoming port of the receiving ring makes a copy of the received data packet, marks it as invalid, and forwards it to the outgoing port of the same ring. If the original received data packet is reserved, the incoming port unreserves it before absorbing it into its ROB.

The outgoing port of a Bridge node performs arbitration between its non-completion request buffer and completion buffer. The arbitration is Round-Robin based with priority to the completion when the transiting transfer slot from the incoming port of the bridge node is reserved for the destination of the completion.

Tables 3.3 and 3.4 summarize the output of the incoming port and outgoing port for the different type of nodes depending on the type of data packet it receives. Table 3.3 describes the behaviour of the incoming port when it receives read, write and completion data packets for its node. The port either accepts the data packet or bounces it. Table 3.4 describes the behaviour of the outgoing port when it receives read, write and completion data packets from the agent. The port either passes the data packet onto the RI or stalls it in its FIFO. The behavior of the outgoing port of the Bridge node is the combined behavior of the outgoing ports of the Initiator and the Target node since it passes all types of commands.

Node type	Read		Write		Completion	
	Accept	Bounced	Accept	Bounced	Accept	Bounced
Target	**Clear flags:** Valid and Command Order ID valid **Set field**: Command type to reserved for completion	**Set flag:** Command Order ID valid **Set field:** Command Order ID	**Clear flags:** Valid and Command Order ID valid	**Set flag:** Command Order ID valid **Set field:** Command Order ID	**Set field:** Alert	N/A
Initiator	**Set field:** Alert	N/A	**Set field:** Alert	N/A	**Clear flags:** Valid and Command Order ID valid	Do nothing
Supervisor	**Clear flags:** Valid and Command Order ID valid **Clear field:** Alert	**Set flag:** Command Order ID valid **Set field:** Command Order ID	**Clear flags:** Valid and Command Order ID valid **Clear field:** Alert	**Set flag:** Command Order ID valid **Set field:** Command Order ID	**Clear flags:** Valid and Command Order ID valid **Clear field:** Alert	Do nothing
Bridge	**Clear flags:** Valid and Command Order ID valid **Set flags:** Reserved and Booked for bridge (if set in received data packet) **Set field:** Reserved node ID with Node ID	**Set flag:** Command Order ID valid **Set field:** Command Order ID	**Clear flags:** Valid and Command Order ID valid **Set flags:** Reserved (if set in received data packet) **Set field:** Reserved node ID with Reserved Node ID of absorbed data packet	**Set flag:** Command Order ID valid **Set field:** Command Order ID	**Clear flags:** Valid and Command Order ID valid **Set flags:** Reserved (if set in received data packet) **Set field:** Reserved node ID with Reserved Node ID of absorbed data packet	**Set flag:** Command Order ID valid **Set field:** Command Order ID

Table 3.3: Actions performed by the incoming port before forwarding the data packet to the outgoing port

Command Type	Action	Initiator/ Supervisor	Target
Read	**Pull from FIFO**	**Set flag:** Reserved **Set field:** Reserved node ID with destination node ID **Update State:** Decrement the Reserved_counter if the Reserved flag was set	N/A
	Stall FIFO	If the Can_reserve_counter is above the threshold and the data packet is not reserved already, then: **Set flag:** Reserved **Set field:** Reserved node ID with own node ID value	N/A
Write	**Pull from FIFO**	**Clear flag:** Reserved **Update State:** Decrement the Reserved_counter if the Reserved flag was set	N/A
	Stall by FIFO	If the Can_reserve_counter is above the threshold and data packet is not reserved already, then: **Set flag:** Reserved **Set field:** Reserved node ID with Node ID value	N/A
Completion	**Pull from FIFO**	N/A	If the Reserved_counter is greater than 0 or the last word of a SRMD is passed, then: **Clear flag:** Reserved if set **Update State:** Decrement the Reserved_counter accordingly
	Stall by FIFO	N/A	If the Can_reserve_counter is above the threshold and data packet is not reserved already, then: **Set flag:** Reserved **Set field:** Reserved node ID with Node ID value

Table 3.4: Actions performed by the outgoing port before forwarding the data packet to the RI

3.2 Traffic Initiator

So far the RI, its protocol and components have been described. This section introduces a mechanism to create and inject valid data packets into the RI. This is required to:

- Test and stress the RI.
- Measure the Quality of Service for a given number of devices connected using the RI. It ensures that the RI fulfills the performance needs set by IPU requirements for known tasks and scenarios.
- Perform on the fly data integrity checks.
- Detect livelocks in the RI.
- Highlight advantages and weaknesses of the RI.
- Determine optimal system configurations (buffer sizes, the number of optional pipe stages, outgoing port's reservation thresholds, etc) for commonly used scenarios.

A model called the traffic initiator (TI) was developed to achieve the above. It serves as a test bench for the RI which is the design under test (DUT). The TI emulates devices connected to initiator nodes of the RI. The following terminologies are used when describing the traffic initiator model:

1. An **action** is defined as a request from the device of the initiator node that results in the insertions of valid data packets into the RI.
2. A **task** is defined as a collection of actions. As explained later, a task is triggered only when certain preconditions are met. Once a task is triggered, it executes actions in sequential order.
3. A **traffic scenario** is a collection of tasks that is described by a **Task graph**. A task graph is an acyclic graph that associates a task to a node of the graph. For an acyclic graph, it has the exception that leaf nodes can be connected to initial nodes. Initial nodes (the topmost node) of the graph are initially triggered without any preconditions. After that, they can be re-triggered by leaf nodes. A child node of the graph is triggered only when all its parent nodes have completed their tasks. If leaf nodes trigger initial nodes, then a traffic scenario completes after a predefined number of loops. Otherwise, the scenario completes when all leaf nodes complete their tasks. Figure 3.6 depicts how a scenario is made using tasks and actions.

All the required information associated with a traffic scenario are described in the following XML files:

- **Task description XML**. This file describes the actions that are associated with a task and the order in which they are executed sequentially.
- **Task graph XML**. This file describes the criteria that must be met to trigger tasks of a scenario and hence the order in which tasks are executed.

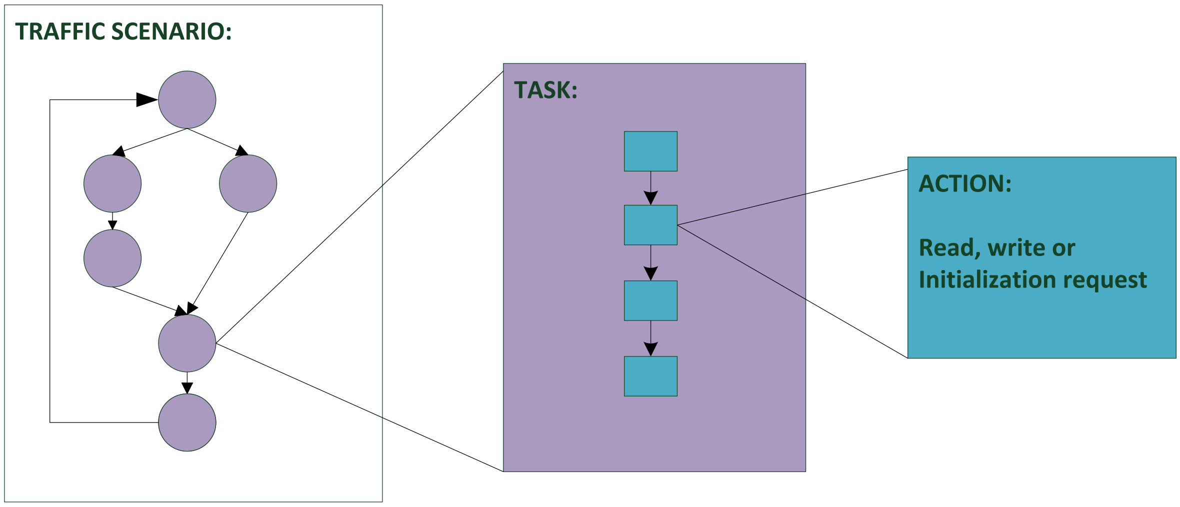


Figure 3.6: A traffic scenario is made using tasks. Tasks are in-turn made using actions

- **Ring topology XML**. This file defines the ring nodes used in the topology and describes the connectivity between them.
- **Node hardware mapping XML**. This file defines the nodes of the task graph that must be mapped onto devices of initiator ring nodes.

The advantages of having the scenario information distributed across these XML files are:

- Task definitions can be reused within the same scenario and across different scenarios.
- The order of executions of tasks is easily configurable.
- Node-task mapping can be easily changed.
- Having the same source files for different simulation platforms.

The TI is modeled using the information that is in the above mentioned XML files. It has the structure of a 2 level acyclic graph. It has one parent node and as many leaf nodes as the number of initiator nodes in the topology.

Based on information obtained from the task graph, a parent node: Starts simulation, triggers children nodes, communicates task completions among them and stops the simulation when the traffic scenario has completed. It also monitors the status of actions, tasks and the scenario. During simulation, Quality of service (QoS) is measured by:

- Measuring the latency of requests (Time between a request is injected into the ring and it is absorbed by an agent)

- Monitoring task deadlines. A task misses its deadline because of latency introduced by the RI.

The leaf nodes are the devices of initiator ring nodes. Nodes of the task graph are mapped onto the leaf nodes of the TI based on the Node hardware mapping XML. Based on information obtained from the task description XML and the task graph XML, a leaf node: injects requests to the corresponding initiator node, reports completions of tasks to the parent node and makes data integrity checks when a request injected has completed.

For example, consider figure 3.7, figure 3.8 and table 3.5 that depict the content of the ring topology XML, the task graph XML and the node hardware mapping XML. From figure 3.7, it can be observed that the ring topology has 2 initiator nodes (node IDs 1 and 2), 2 target nodes (node IDs 3 and 4) and a supervisor node (node ID 0). Thus, there are three nodes that inject requests into the RI. Figure 3.8 shows how a task graph with 7 graph nodes and 5 tasks looks like. In this task graph, node 1 is initially triggered without any preconditions. Later it is triggered again by nodes 7 and 6. Nodes 3, 4 and 6, 7 reuse the task definitions of 3 and 5 respectively. Table 3.5 describes the mapping of nodes of the task graph onto devices of the ring node. Using this information the TI model shown in Figure 3.9 is created. Task graph nodes are mapped onto leaf nodes of the traffic initiator. These leaf nodes are devices of corresponding initiator agents. Task graph nodes 1, 6 and 7 are mapped onto leaf node 0 that is the device of ring node 0. Thus, ring node 0 initially executes task 1 (Graph node 1) followed by task 2 (Graph Node 2). On completion of task 2, it executes task 5 (Graph node 6). It then executes Task 5 again (Graph node 7) when task 4 (Graph node 5 that is mapped to ring node 1) finishes. After that, the cycle is repeated till the traffic scenario completes.

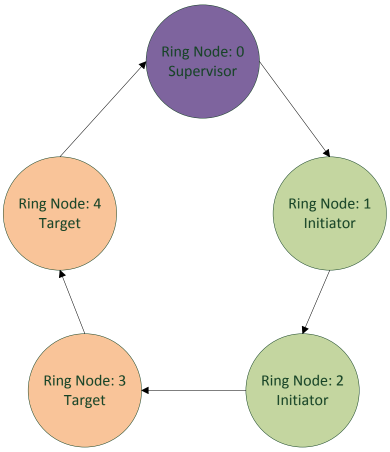


Figure 3.7: A ring topology

Ring node ID	**Task graph node ID**
0	1, 6 and 7
1	2 and 5
2	3 and 4

Table 3.5: Information present in the node hardware mapping XML

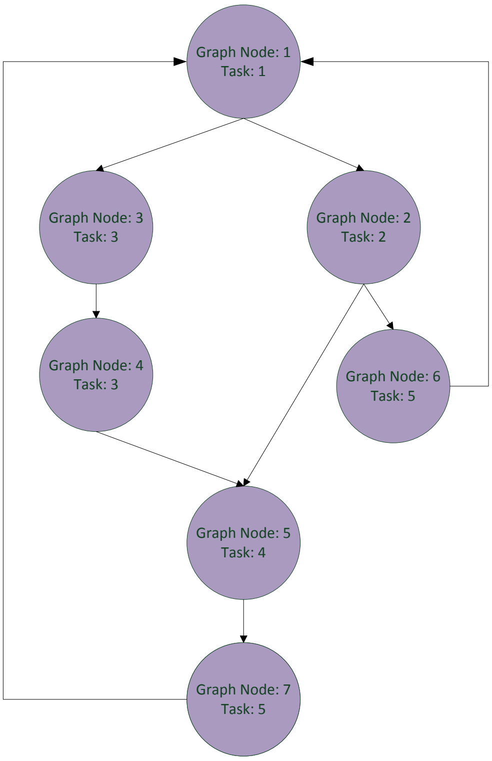


Figure 3.8: Task graph

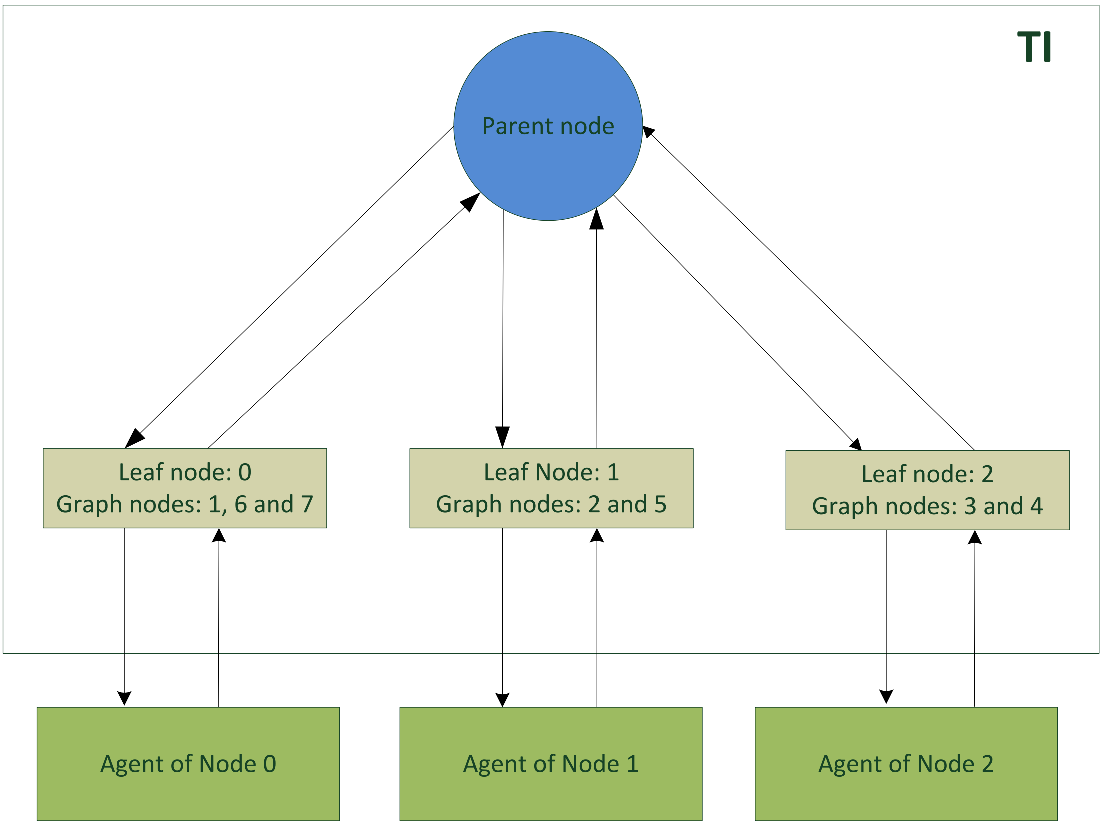


Figure 3.9: Traffic Initiator for figure 3.1

3.3 Summary

The RI is designed to replace the existing system bus interconnect in future generation Intel IPUs. By using lesser number of wires with shorter lengths, it overcomes limitations of the currently used fully connected interconnect. It is designed to meet the throughput, area and power requirements of future generation IPUs. It also provides scalability, composability, QoS and better power management techniques.

This chapter introduced the ring interconnect, its components and protocol. In an RI, nodes are connected to each other via one or more communication channels. Every node is made up of an agent, a pair of incoming and outgoing ports for every communication channel, a router and an arbiter. An incoming port is an interface between a communication channel and the node whereas an outgoing port is an interface between the node and a communication channel. The agent is the interface between a computational or storage unit (called a device) and the node. Routers and arbiter are used only in nodes of a multichannel RI. A router decides which channel a data packet should use for traversal. An arbiter decides the channel from which a data packet should be selected.

Depending on its function, a node can either be an initiator node, target node, supervisor node or bridge node. Initiator nodes insert read and write requests in the RI and the target nodes absorb these request from the RI. Note that data packets are used by nodes to insert requests in the RI. The Supervisor node is a special initiator node, and an RI can contain only one such node. After power-on, it issues a command that assigns unique IDs to all nodes. These IDs are used as fields of the data packet to indicate its source and destination. The supervisor node also monitors the RI to make sure that there are no live-lock situations.

Every cycle, a data packet is forwarded from one node to the other. This guarantees the RI can never deadlock. Depending on the values of its fields, a data packet can be: valid & reserved, valid & unreserved, invalid & reserved or invalid & unreserved. A data packet is valid if it contains a request. When a node absorbs a request from the data packet, it marks it as invalid and forwards it to the next node. A reserved data packet is one that can be used only by the node that has reserved it. Nodes reserve data packets when they have a request to insert and the data packet passing by is unreserved & valid. This is done to guarantee that no other node can use the data packet to insert a request when the data packet becomes invalid. When a node inserts a write request, it unreserves the data packet. When the node inserts a read request, it reserves the data packet for the target node. This guarantees a minimum throughput of target nodes. A node unreserves a data packet reserved for itself when: it has nothing to insert or it inserts the last completion in response to a read request (only target node).

The traffic of the RI greatly depends on the buffer size of the incoming and outgoing ports, the buffer size of the initiator agent, reservation mechanism of the outgoing port, router policy and arbiter policy. To understand the effect of these parameters on the performance of the RI, a model called the Traffic Initiator was introduced. It inserts traffic in the RI: validate the protocol and measure its performance. The TI has the structure of a 2 level acyclic graph. It has one parent node and as many children nodes (called leaf nodes) as the number of initiators in the RI. The leaf nodes of the TI are connected to the agents of the initiator nodes of the RI. The TI is configured to execute a traffic scenario. A traffic scenario is a collection of tasks and a task is composed by a series of actions. In this traffic scenario, the parent node triggers leaf nodes to execute tasks based on the task graph. This makes the leaf nodes to start inserting write or read requests into the RI (actions). When a leaf node completes its task, it notifies the parent node that uses this information to trigger other tasks. The TI is then used to measure the performance of the RI for particular traffic scenarios and to make sure that the RI meets the performance needed by IPU applications.

Implementation 4

The previous chapter described the functionality of the RI and the model used to test and analyze it. Together, they make the test system. This chapter explains how the test system is modeled. Section 4.1 provides an overview of the environment used for modeling and analyzing the ring interconnect. Section 4.2 explains how the traffic scenario XMLs are processed to create the test system. Finally, Section 4.3 summarizes how the RI and TI nodes are modelled and simulated.

4.1 Test Environment

The test system is simulated using an in-house simulator. The simulator is built in 3 stages:

- **Build stage 1**. This stage involves compiling the different types of RI and TI nodes. C/C++ based behavioural models of the different types of RI and TI nodes are compiled along with the simulation library interfaces. The simulation library interfaces declare the necessary calls that the behavioural models should make to communicate with each other.

- **Build stage 2**. In this stage, the topology to be simulated is verified. A system description file written in an internally developed language is parsed. The file describes the connections between different models and also defines the parameters of the models that are configurable at design time (for example buffer lengths of the incoming and outgoing ports).

- **Build stage 3**. This stage creates the final simulation application using the outputs of the first two stages. Information from the system description file is used to link the binaries produced in the first stage along with an internally developed simulation library to get the simulator. The simulation library:
  - Instantiates the models described in the system description file at the start of the simulation. It uses the parameters from the system description file while instantiating them.
  - Simulates all the models in cycle-accurate fashion.
  - Communicates required data between the models based on the connections described in the system description file.
  - Creates log files that describe the state of the models during every cycle of simulation.

Figure 4.1 shows the steps involved in creating the simulator.

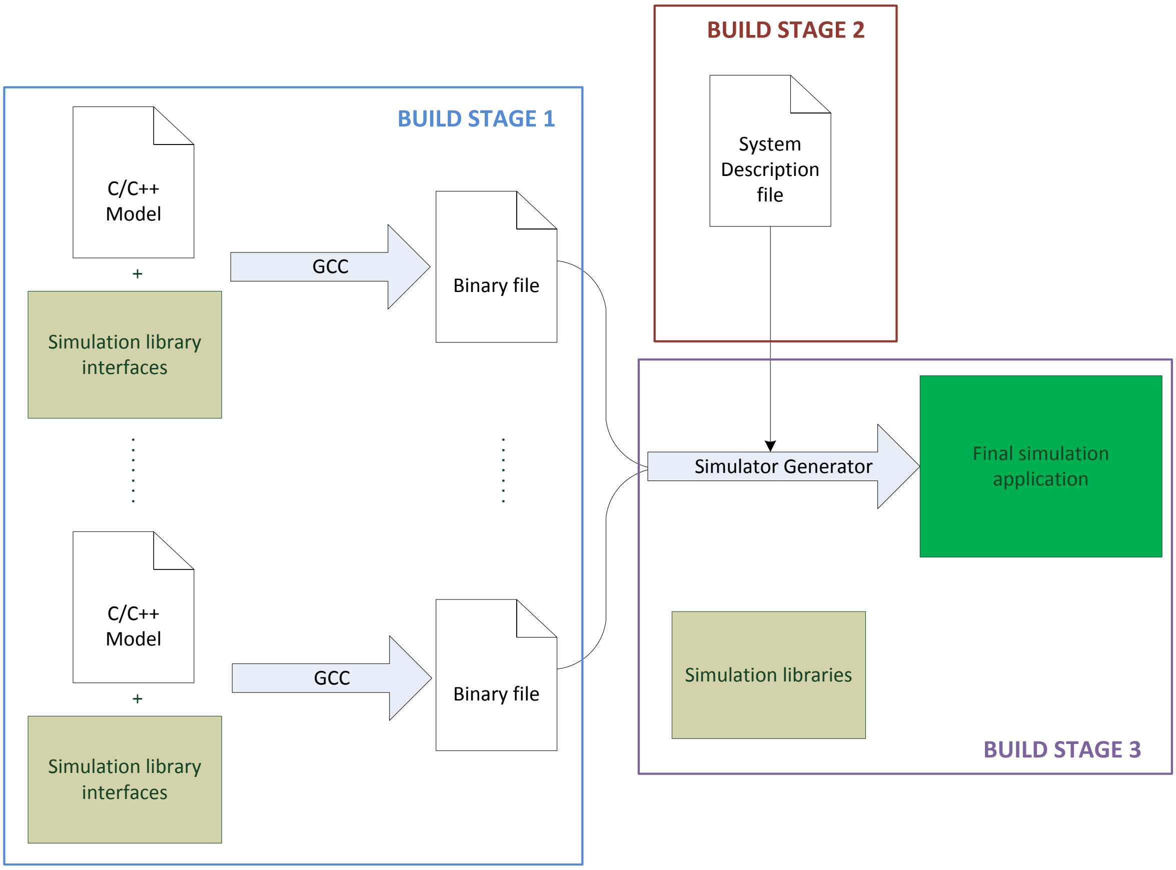


Figure 4.1: Generation of the simulation application

4.2 Parsing the Traffic Scenario XMLs

As shown in figure 4.2, the XML files of a traffic scenario are parsed to create 2 types of files:

1. Configuration files are created for every node of the TI. These files are read by different nodes of the traffic initiator during run-time when they are instantiated. The configuration files of parent nodes contain information extracted from the task graph XML. Configuration files of leaf nodes contain information extracted from both the task description XML and the task graph XML.

2. The system description file that describes connectivity between ring nodes, and between ring nodes and the TI. All the initiator nodes of the ring are connected to leaf nodes of the traffic initiator, and all the target nodes of the ring are connected to a device that emulates a memory.

The parser can be configured to produce only the system description file or only the configuration files or both. When the RI remains the same but a different traffic scenario is used, then only the configuration files have to be regenerated. The system description file that contains design-time parameters remains the same. Since only run-time changes are made, the time-consuming process of generating the final simulation application can be avoided.

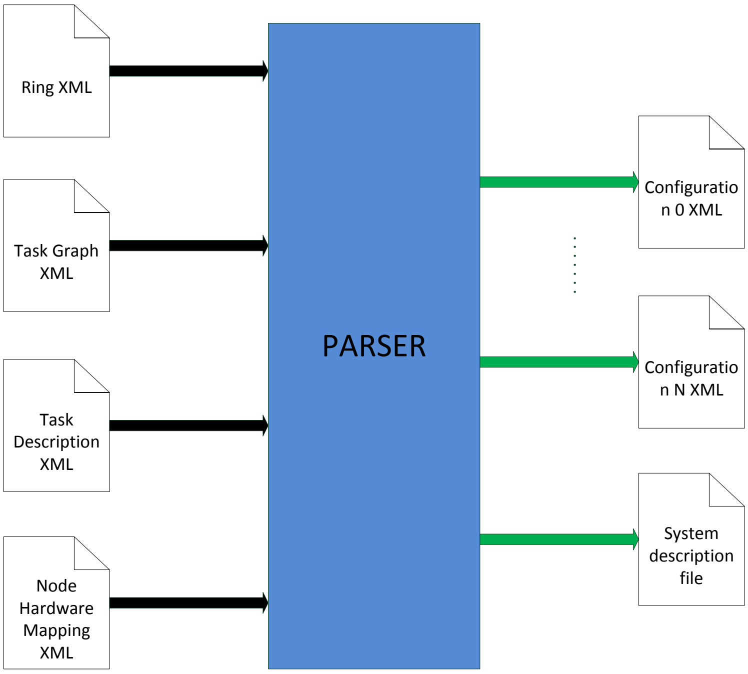


Figure 4.2: XML Parser that parses the Ring XML, Task graph XML, Task description XML and Task graph node hardware mapping XML to create configuration files for the nodes of the BTI and an interface file

Figure 4.3 pictorially represents the content of the system description file that is created to test the RI shown in figure 3.1 for a traffic scenario depicted in figure 3.8, figure 3.7 and table 3.5.

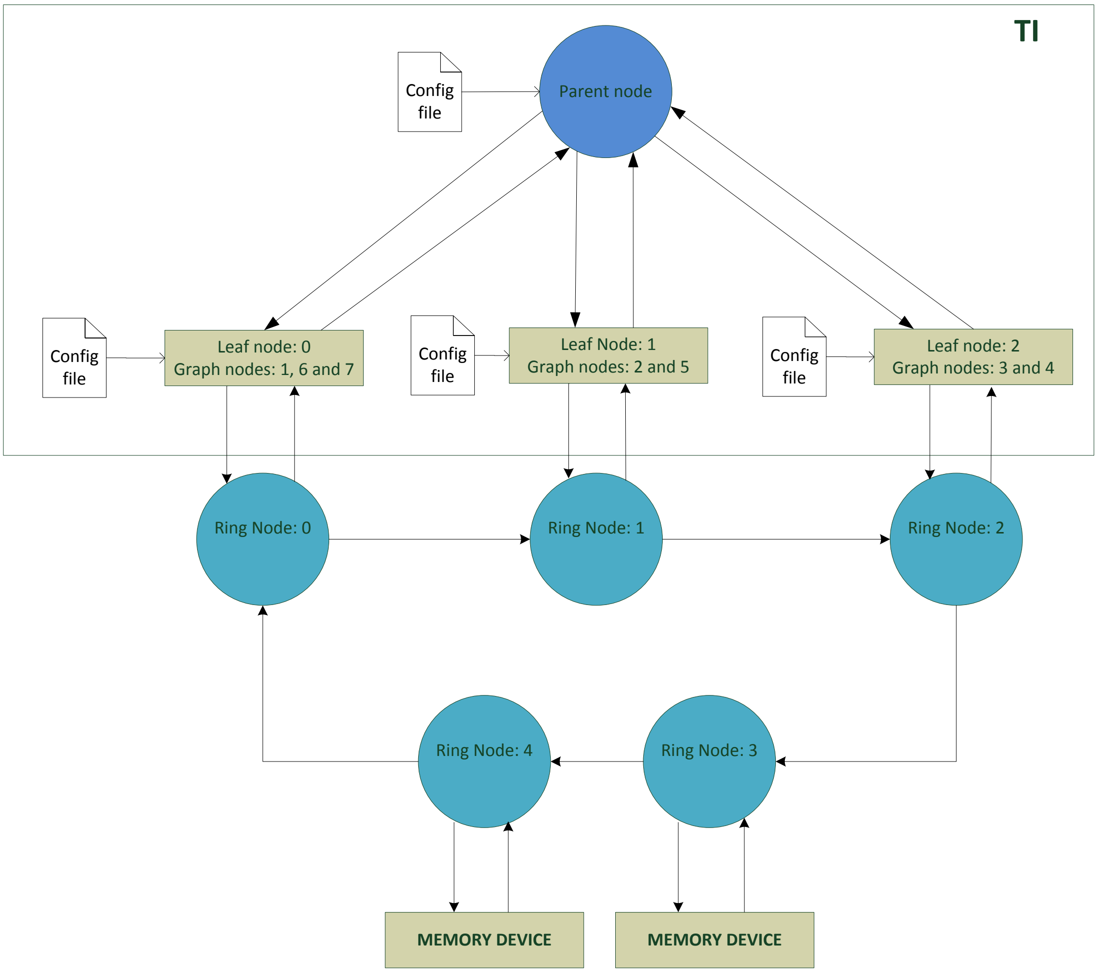


Figure 4.3: TI generated using information obtained from figure 3.8 and figure 3.7 and table 3.5

4.3 Modeling

The following sub-sections describe how nodes of the RI and nodes of the TI are modelled in C++.

4.3.1 RI models

Figure 4.4 describes the software architecture of an RI node using a UML diagram. C++ based models of the different sub-components of a generic RI node are implemented. Using the concepts of inheritance, they are extended and used to create node specific sub-components. Taking the advantage of polymorphism, the sub-components of different types of nodes are then combined to obtain the model of a specific type of node. For example, while constructing the target node, the incoming ports are built with ROBs and pipe-stages (optional), the agent is instantiated as a target agent, and an arbiter and router are constructed if the number of channels is more that one and the outgoing port is constructed with FIFO queues.

All the classes have a **run** function. It contains the main functionality of the class. The **run**, **send_output_data_packets**, **receive_input_data_packets**, **receive_data_packet_from_device** and **send_data_packet_to_device** functions of a node are linked to the simulation libraries. Every cycle of simulation, the simulation libraries call the run function of a node which calls the run functions of all its sub-components and manages the transfer of data packets from one sub-component to the other. After simulating the functionality of all its sub-components, the run function of a node ends with a synchronization point. It makes sure that, within the same simulation cycle, only after all nodes have reached this point, a node calls the send_output_data_packets function. When this function is called by a node, the simulation library calls the receive_input_data_packets function of another node based on the content of the system description file. Similarly, the simulation library calls the receive_data_packet_from_device function of the node when a device calls its send_data_packet_to_node function (introduced in the next sub-section).

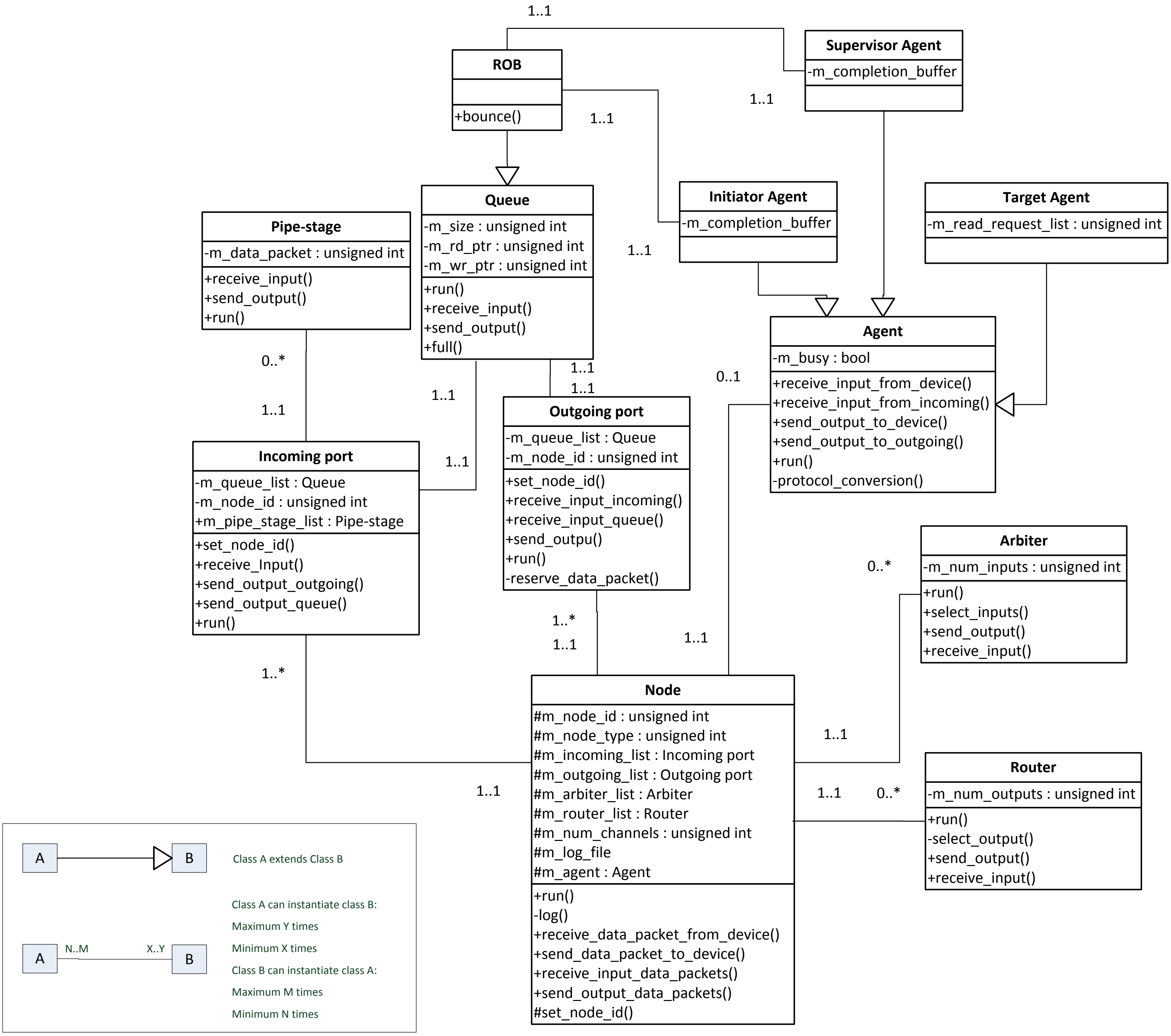


Figure 4.4: Software architecture of a ring node in terms of a UML diagram

4.3.2 Traffic initiator model

Figure 4.5 shows the software architecture of a node of the traffic initiator in terms of a UML diagram. The common operations and attributes of a TI node are implemented in a base class. The base class is then extended by leaf nodes and parent nodes. A leaf node class populates the list of tasks it's supposed to execute using the information from its configuration file. As we already mentioned, every task contains a list of sequential actions that define the task. When a leaf node finishes a task it notifies the parent node. Based on the content of its configuration file, the parent node then forwards this information to the leaf nodes that are dependent on the corresponding task completion. When a leaf node completes all its tasks, it notifies the parent node. When the parent node receives completions of all the leaf nodes, it either re-triggers the initial tasks or stops simulation based on the content of its configuration file.

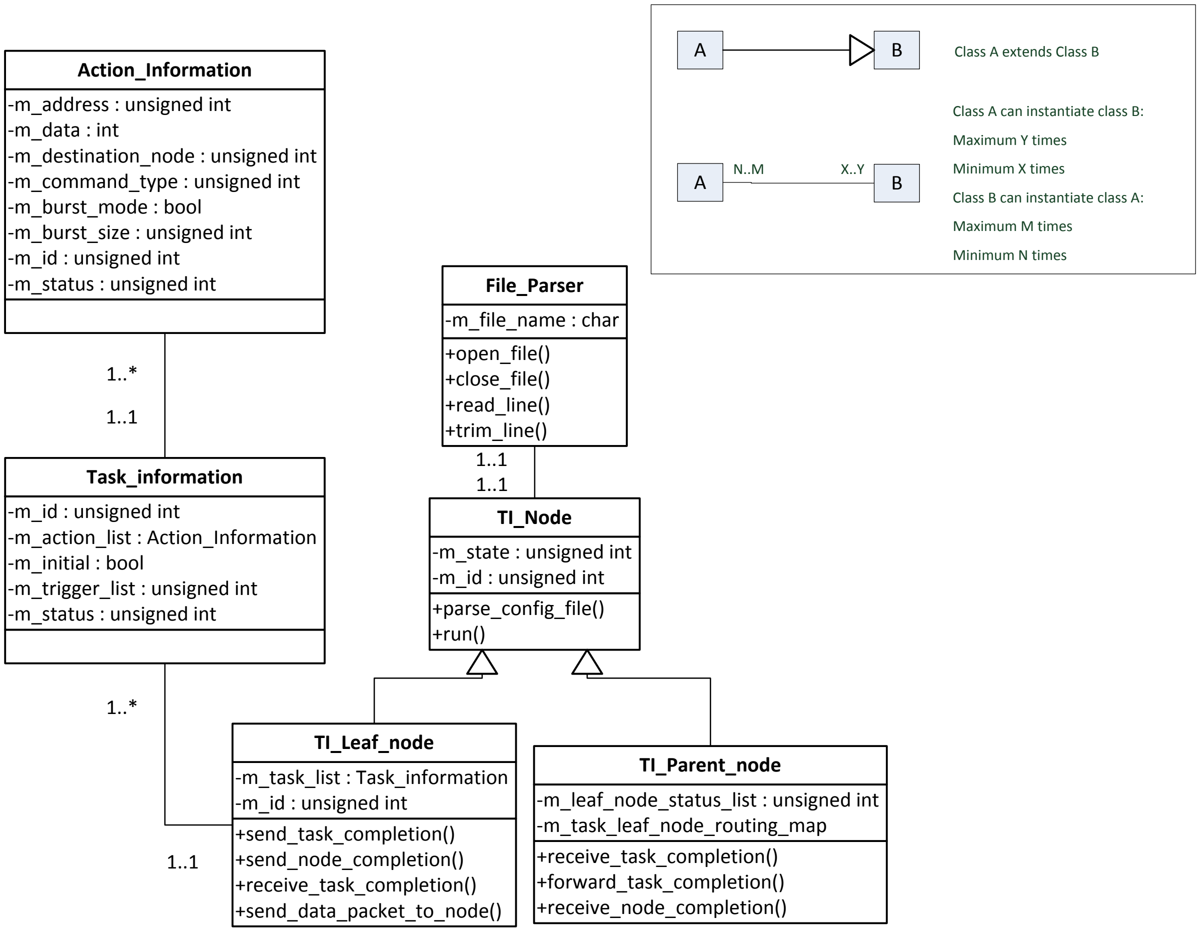


Figure 4.5: Software architecture of a traffic initiator node in terms of a UML diagram

Similar to the RI nodes, all the nodes of the traffic initiator have a **run** function that is called by the simulation library every cycle. It contains the main functionality of the class. Along with the run function, the **send_task_completion**, **send_node_completion** and **send_data_packet_to_node function** of a leaf node, and the **receive_task_completion**, **forward_task_completion** and **receive_node_completion** functions of the parent node are linked to the simulation libraries. Based on the content of the system description file, the simulation library calls the:

- receive_task_completion function of the parent node when a leaf node calls the send_task_completion function.
- receive_task_completion function of the leaf node when the parent node calls the forward_task_completion function.
- receive_node_completion function of the parent node when a leaf node calls the send_node_completion function.
- receive_data_packet_from_device function of the RI node when a leaf node calls the send_data_packet_to_node function.

Figure 4.6 summarizes how the test setup is simulated in terms of a flow chart.

4.4 Summary

This chapter explains how the setup to test the RI is implemented. The test setup comprises of the RI (DUT) and the TI model. First, behavioural C++ models of the RI and TI nodes are implemented using the concepts of polymorphism and class inheritance. Then, an XML parser uses the traffic scenario XMLs to create the configuration files for the TI nodes and a system description file. The configuration file of the parent TI node contains task graph related information whereas the configuration files of the TI leaf nodes contain both task graph and task-related information. The system description file describes the topology of the RI, and connections between the RI and TI nodes in an Intel proprietary language. Finally, a simulator is built using the C++ models of the RI nodes and TI nodes, internally developed simulation libraries and the system description file. The content of the system description file is used to link function calls of the simulation libraries to functions of the RI and TI nodes' models. The simulator along with the configuration files for the TI nodes are used to simulate the RI for the required traffic scenario in cycle accurate fashion. Since the content of the system description file remains the same for a given RI topology. Different traffic scenarios for the same RI topology are simulated by only regenerating the configuration files.

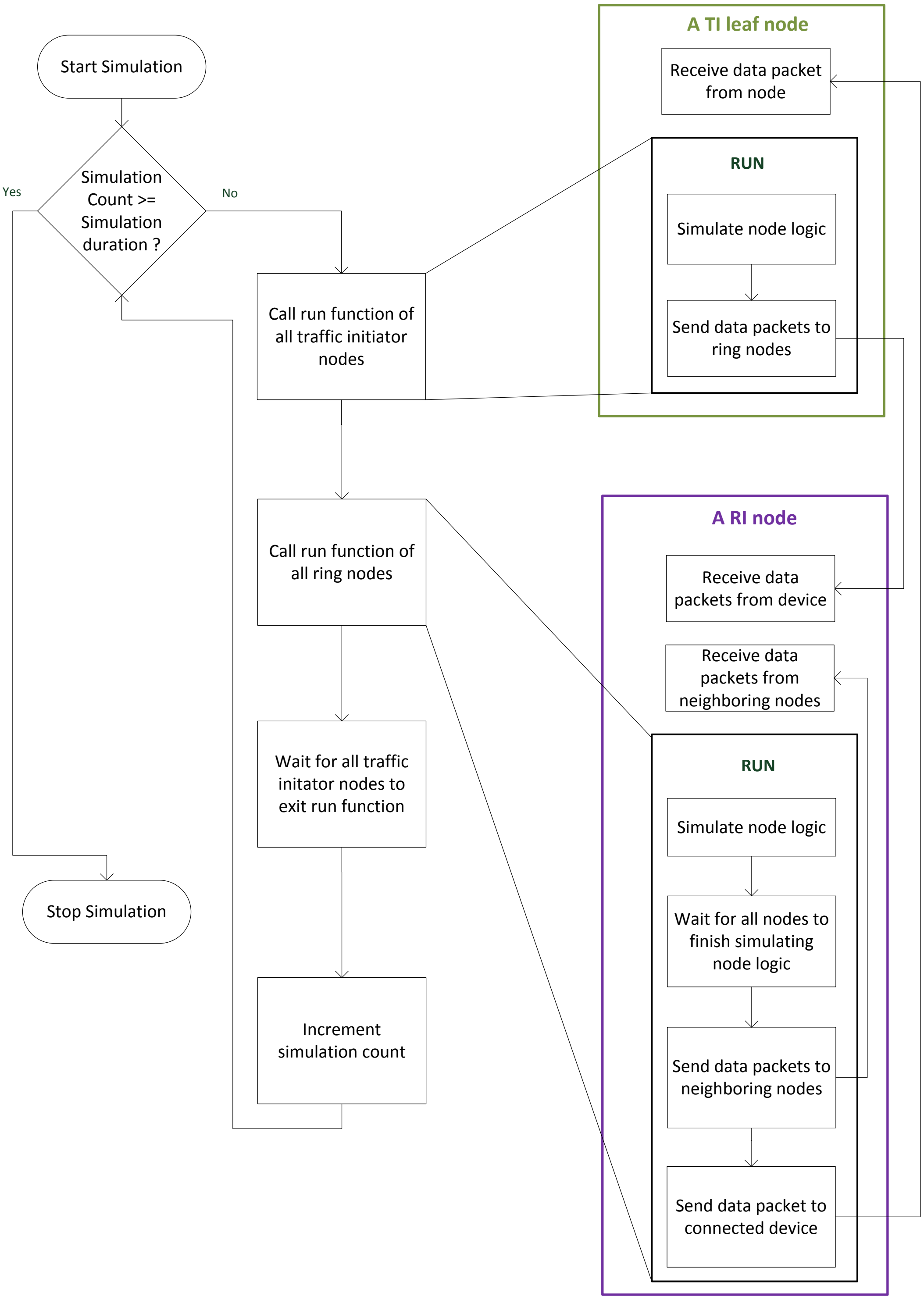

A TI leaf node
Receive data packet from node
RUN
Simulate node logic
Send data packets to ring nodes
Start Simulation
Simulation Count >= Simulation duration ?
Yes
No
Call run function of all traffic initiator nodes
Call run function of all ring nodes
Wait for all traffic initator nodes to exit run function
Increment simulation count
Stop Simulation
A RI node
Receive data packets from device
Receive data packets from neighboring nodes
RUN
Simulate node logic
Wait for all nodes to finish simulating node logic
Send data packets to neighboring nodes
Send data packet to connected device

Figure 4.6: Flow diagram indicating how the test system is simulated

5 Experimental Results

In this chapter we use the Traffic Initiator to perform Design Space Exploration (DSE) of the ring interconnect and then measure its performance for a real world traffic scenario. Section 5.1 introduces the different metrics that are monitored to quantify the performance of the RI. Section 5.2 then uses synthetic traffic scenarios to arrive at corner cases to study the performance of the RI and its mechanisms. Finally, section 5.3 uses a practical use-case to verify that the RI achieves the performance required by IPU applications, focusing on timely completions and bandwidth.

5.1 Metrics

Using the environment described in the previous chapter, the performance of the RI is quantified by monitoring the following parameters:

1. **The Latency of injected requests**. Latency is the time elapsed from when a request has arrived to the agent of a node until it has been completed. A write request completes when the agent of the target node forwards it to its device. A read request completes when the agent of the initiator node forwards the last completion of the corresponding request to its device. Latency is measured in units of simulation clock cycles.

2. **Effective bandwidth**. It is defined as the ratio of the number of data packets coming from the agent that are inserted into the RI by the outgoing port ($D_{Transmitted}$) and the total number of data packets inserted by the outgoing port into the RI when its FIFO is not empty (D_{Out}).

$$Effective\ Bandwidth = \frac{D_{Transmitted}}{D_{Out}} \tag{5.1}$$

3. **Throughput**. It is defined as the number of requests that complete per cycle. It is measured for all initiator nodes and the RI.

5.2 Design Space Exploration

Table 5.1 summarizes the parameters of the RI that can be configured at design time. In this section we investigate the behaviour of the RI to changes in these parameters. The parameters Node position and Number of communication channels decides the topology of the RI. The parameters Reservation Budget and reserve-again-threshold control the QoS that the RI provides to its nodes. The parameters Completion buffer size, incoming port buffer size, and burst size of SRMD read requests are read request related parameters that influence the amount of traffic in the RI. The number of pipe-stages between adjacent nodes influences the total number of data packets that are circulating in the RI at a given point of time. To understand the effect of changing these parameters on the performance of the RI under boundary conditions, experiments are performed with the TI and ring node models using the environment described in the previous chapter. Due to the size of the design space, while analyzing the effect of a parameter we fix the values of other parameters with practical values. When we ran the experiments, the following assumptions were made:

- Every cycle, the device of initiators (i.e., leaf nodes of the traffic initiator) can insert a request. They can insert a maximum of 200 requests.
- Devices of targets (i.e., memories) can absorb a data packet every cycle. A memory device responds to a read request by passing one completion to its agent every cycle. When a memory is serving a read request, it doesn't accept any other requests.

Parameter	**Description**
Node position	Decides the neighbors and the unique ID of the node in the RI
Number of communication channels	Decides the number of communication channels that a node uses to communicate with its neighbors
Reservation budget (RB)	Number of credits for reserving data packets of the RI
Reserve-again-threshold (RAT)	Number of cycles that the outgoing port must wait to reserve a data packet
Incoming port buffer size	Size of the incoming port's buffer. Affects the number of bounced data packets
Completion buffer size of the initiator agent	Decides the maximum size of a SRMD request
SRMD burst size	Number of completions sent by a target node in response to an SRMD read request
Number of pipe-stages between adjacent nodes	The number of data packets circulating in the RI at a given point of time. It is always a constant and depends on the number of pipe-stages in the RI.

Table 5.1: Parameters of the RI that can be changed at design time

In this section, while measuring the throughput, only the throughput of initiator nodes are measured. Since all the initiator nodes are configured to insert the same number of requests and they all start inserting requests from the start of simulation, the throughput of the RI is a multiple of the minimum throughput of all the initiator nodes.

5.2.1 Topology

In this section, different topologies are analyzed across different traffic scenarios to understand how topology affects the performance of the RI. The following assumptions were made during the experiment:

- Read requests inserted are SRMD and have a burst size of 16. The completion buffer size is set so that the devices of initiators can insert back-to-back read requests.
- Between all communicating nodes, a pipe-stage is present.
- Buffer sizes of the incoming and outgoing port are set to two. Two is the minimum size allowed for the buffers. With sizes two and above, different locations of the buffer can be read and written from simultaneously.
- Outgoing ports can reserve as many data packets as they want (i.e., the value of Reservation budget is ∞ and reserve-again-threshold is 0).

The following subsections summarize the results obtained by analyzing topologies for different scenarios.

5.2.1.1 Multiple Initiators, Single Target (MIST)

The goal of this experiment is to understand the interference of requests on the performance of initiator nodes competing on the same target node. The topology shown in Figure 5.1 is simulated by increasing the number of active initiators. First, the topology is simulated with only IO inserting requests. Then, it is simulated with I0 and I1 inserting requests. Finally, it is simulated with I0, I1 and I2 inserting requests.

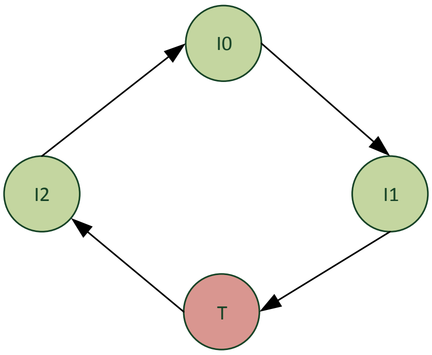


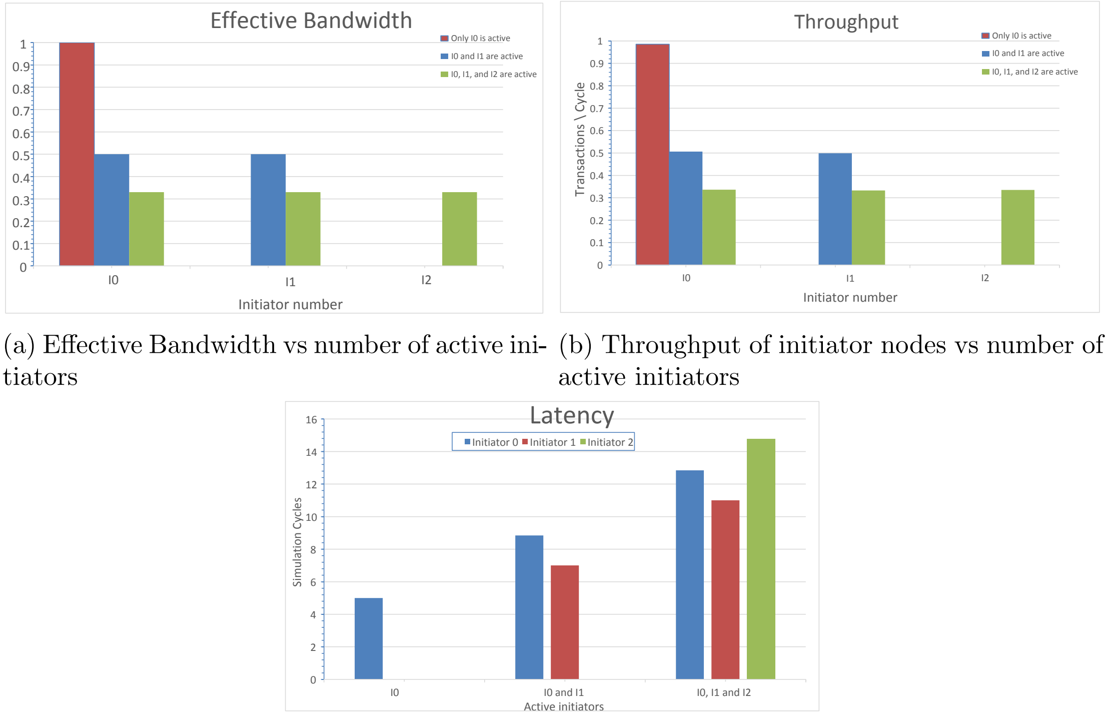
Figure 5.1: Three initiators inject requests for the same target

Three scenarios are simulated:

1. **Scenario 1: The initiators inject only write requests**. As shown in Figures 5.2a and 5.2b, the effective bandwidth and throughput of a node scales down proportionally as the number active initiators increases. The latency of write requests increases as the number of active initiators increase (Figure 5.2c). This is because when only initiator 0 is active (Single initiator, single target - SIST), it is not blocked by requests from other nodes and can insert as many data requests as it wants. The effective bandwidth of the node is 1 and its throughput is maximum. The latency of the write requests and the throughput of the node only depend on the number of pipe-stages between the initiator and the target node. When multiple initiators are active, the reservation mechanism of nodes allows all the initiators to insert write requests fairly. The latency of requests and throughput of the node then depend on the distance of the initiator from the target and the parameters that influence the reservation mechanism of the nodes.

2. **Scenario 2: The initiators inject only read requests**. Nodes wait for burst size number of cycles before injecting another read request for the same target. This good citizen behaviour of nodes guarantees all initiators use the target fairly and also help in reducing the bouncing of read requests. Figures 5.3a and 5.3b show that the latency of read requests and throughput of the initiator nodes scale proportionally as the number of active initiators is increased. Figure 5.3b shows that as the number of active initiators is increased, the effective bandwidth of the node depends on its position in the RI. The active initiator node that is situated immediately towards the right of the target node has the least effective bandwidth. This is because the completions from the target prevents the node from inserting read requests.

3. **Scenario 3: Initiator 0 injects read requests and initiators 1 and 2 inject write requests**. Figures 5.4a and 5.4b compare the throughput and latencies of the initiators to their respective SIST case (i.e., the scenario in which only the corresponding initiator was inserting requests). In this scenario, the throughput of write requests degrades much more than the read requests. The good citizen behaviour of nodes towards read requests and the presence of a reserved data packet to insert completions do not degrade the performance of read requests much. Furthermore, the memory device of a target node takes burst size amount of time to cater to a read request. This causes the target to bounce requests once its ROB is full which increases the latencies of requests.

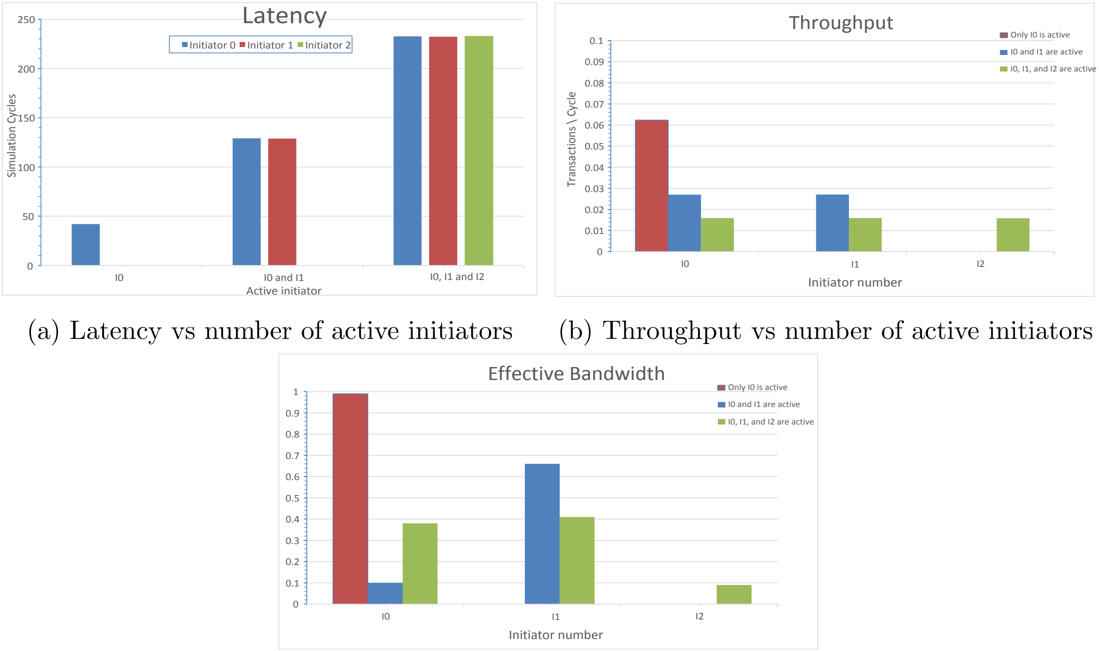


(a) Effective Bandwidth vs number of active initiators

(b) Throughput of initiator nodes vs number of active initiators

(c) Latency vs number of active initiators

Figure 5.2: Performance of write requests

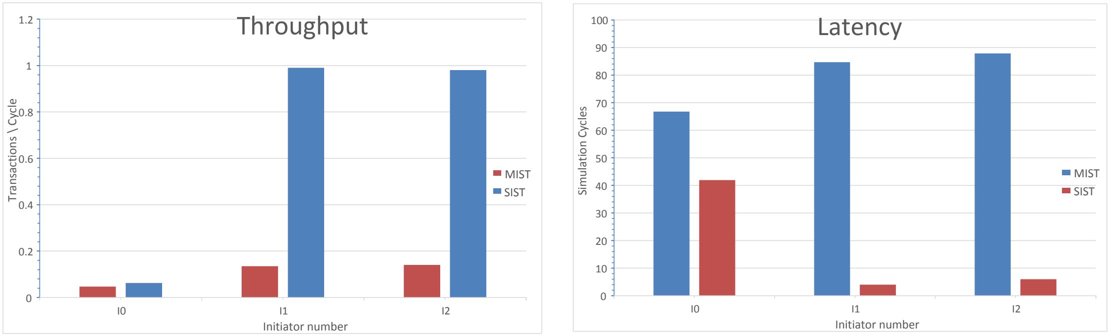


(a) Latency vs number of active initiators

(b) Throughput vs number of active initiators

(c) Effective Bandwidth vs number of active initiators

Figure 5.3: Performance of read requests



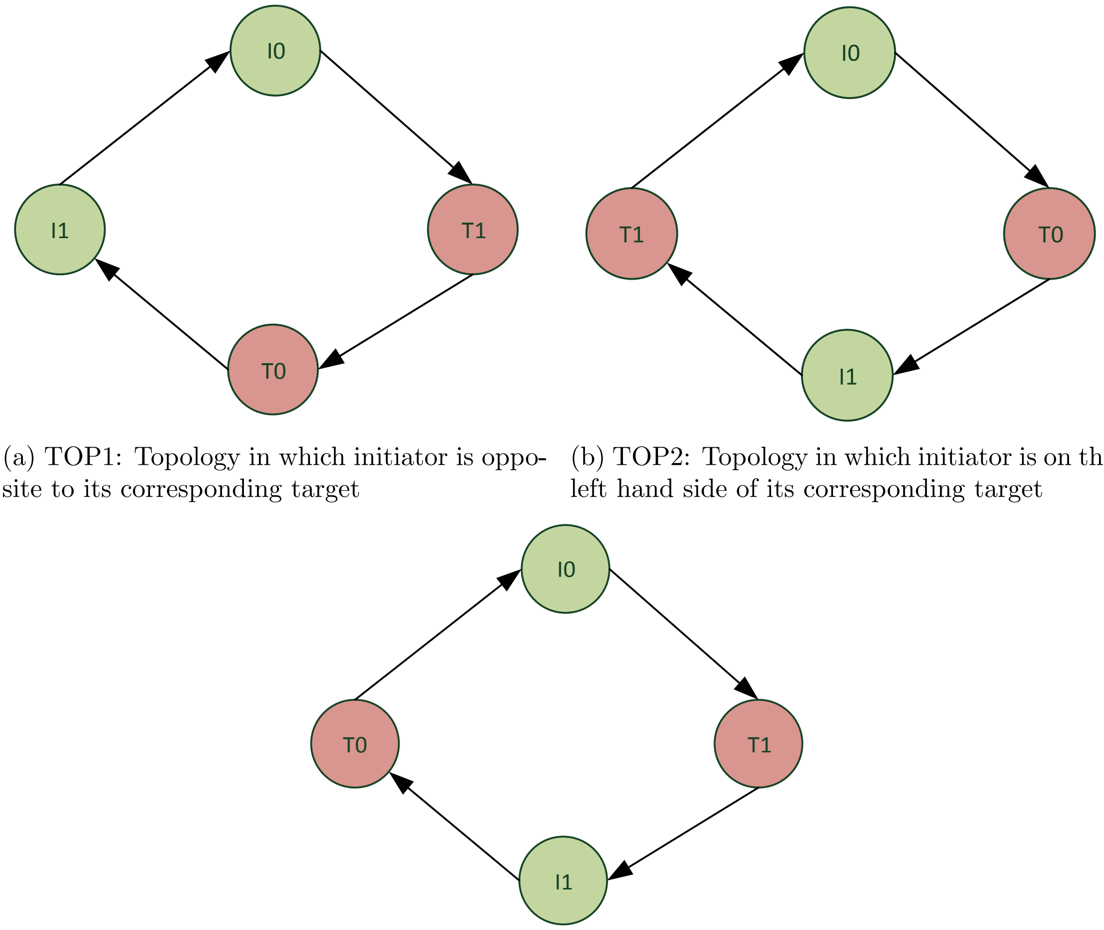
(a) Throughput of MIST vs SIST case

(b) Latency of MIST vs SIST case

Figure 5.4: Performance obtained when initiator 0 inserts read request, and initiators 1 and 2 insert write requests

5.2.1.2 Multiple Initiators, Multiple targets

The topologies shown in Figures 5.5a, 5.5b and 5.5c are analyzed to understand the effect of changing the position of the target node about the initiator node. In all the topologies initiator 0 and initiator 1 inject requests for target 0 and target 1, respectively.



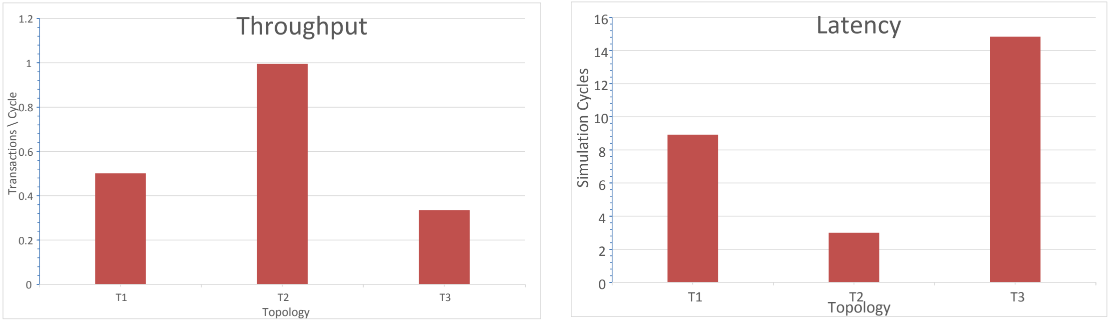
(a) TOP1: Topology in which initiator is opposite to its corresponding target

(b) TOP2: Topology in which initiator is on the left hand side of its corresponding target

(c) TOP3: Topology in which initiator is on the right hand side of its corresponding target

Figure 5.5: Different topologies analyzed to understand the effect of position of target node with respect to its corresponding initiator node

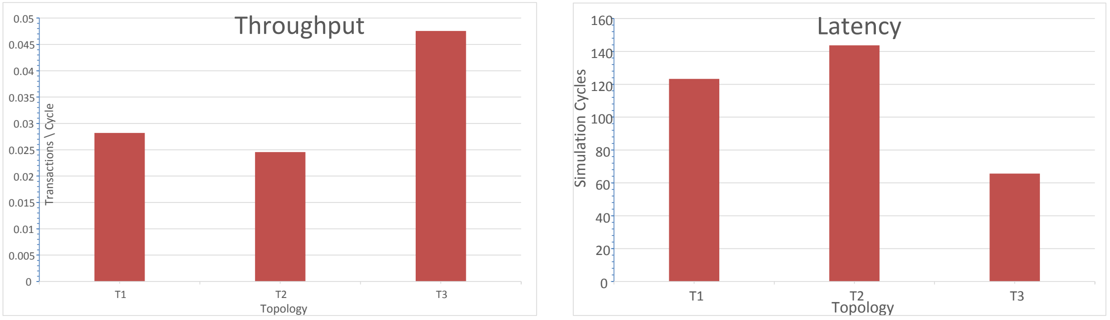
The topologies are simulated twice; first with the initiators inserting write requests only and then with the initiators inserting read requests only. Figures 5.6a and 5.6b show that TOP2 is best suited for write transactions. This is because the write requests from initiator 0 to target 0 and from initiator 1 to target 1 have to travel minimum distance. Also, the requests don't interfere with each other (i.e, requests from initiator 0 to target 1 doesn't pass via initiator 1 and vice versa). On the other hand, Figures 5.7a and 5.7b show that reads suffer since the completions have to travel longer paths. Also, since both the targets are responding to completions, there is an increase in traffic. TOP3 is optimum for read transactions since completions have to travel shorter paths. Writes suffer because of increase in distance from initiators to targets. TOP1 provides better results for writes compared to TOP3 and better results for reads compared to TOP2. This is because the distance of write and completions of read requests to their destinations is reduced compared to that in TOP2 and TOP3 respectively.



(a) Throughput of write requests

(b) Latency of write requests

Figure 5.6: Performance of write requests in TOP1, TOP2 and TOP3



(a) Throughput of read requests

(b) Latency of read requests

Figure 5.7: Performance of read requests in TOP1, TOP2 and TOP3

The topology shown in Figure 5.8 makes use of two layers to take the benefits of TOP2 and TOP3 for writes and reads respectively. The read commands are sent clockwise whereas the write commands are sent anti-clockwise. This way, the write requests achieve the same performance as obtained by TOP2 and the read requests achieve the same performance as obtained in TOP3. Figure 5.9 indicates the same.

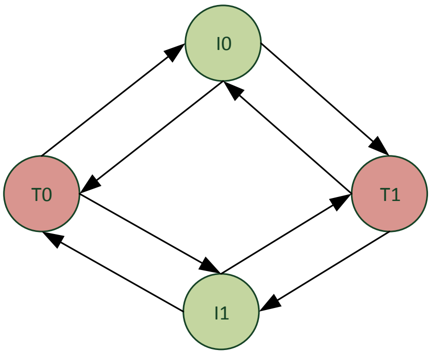
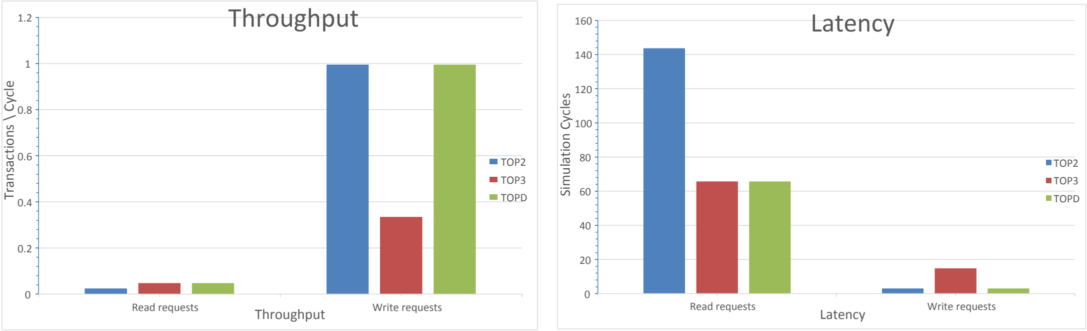


Figure 5.8: TOPD: Topology in Figure 5.5c with 2 layers



(a) Throughput

(b) Latency

Figure 5.9: Performance of TOPD compared to TOP2 and TOP3

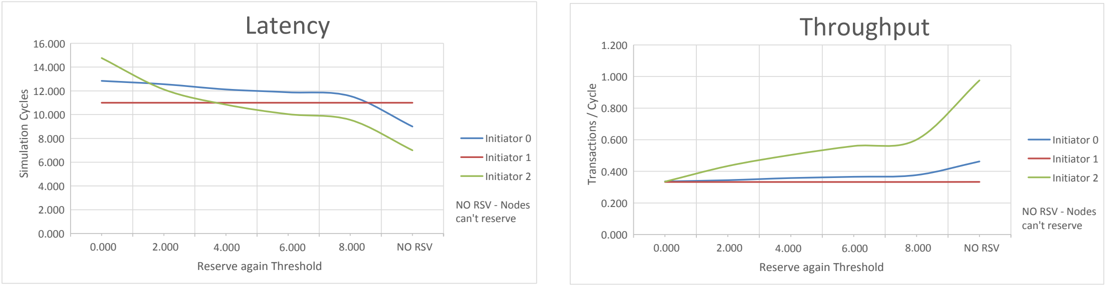
5.2.2 Reservation Mechanism

The Reservation Budget and reserve-again-thresholds of nodes influence how they reserve data packets. These parameters influence the QoS the RI provides to its nodes. By varying these parameters certain nodes in the RI are given priority over other nodes. In this section, we analyze how these two parameters affect the performance of the topology shown in Figure 5.1. The following assumptions were made while carrying out the experiment:

- Between all communicating nodes, a pipe-stage is present.
- Buffer sizes of the incoming and outgoing port are set to two.
- Read requests inserted are SRMD and have a burst size of 16. The completion buffer size is set so that devices of the initiators can insert back-to-back read requests.

First, to observe the effect of only the reserve-again-threshold parameter, we set the Reservation budget to infinity and vary the reserve-again-threshold parameter. We consider the same 3 scenarios mentioned in section 5.2.1.1:

- **Scenario 1: All initiators insert only write requests**. Figures 5.10a and 5.10b show the variation of the latency and the throughput when the parameter reserve-again-threshold increases. As the reserve-again-threshold increases, the number of back-to-back data packets that initiators 0 and 1 can reserve reduces. This gives priority to the write requests from initiator 2, followed by write requests from initiator 0 and finally write requests from initiator 1. The net effect of which is to cause the throughput of initiator 0 and 2 to increase (by approximately 51% and 203% respectively). With an increase in reserve-again-threshold, the amount of traffic in the RI reduces because initiators closer to the target can't insert data packets till initiators further away from the target finish. This causes the latency of requests from initiators 0 and 2 to reduce (by approximately 31% and 53% respectively). The throughput and latency of initiator 1 don't change because of its position in the RI. Thus, to summarize, increasing the reserve-again-threshold increases the throughput of nodes that are further away from the target. The latency of requests reduces because the amount of traffic in the RI from other initiators reduce.

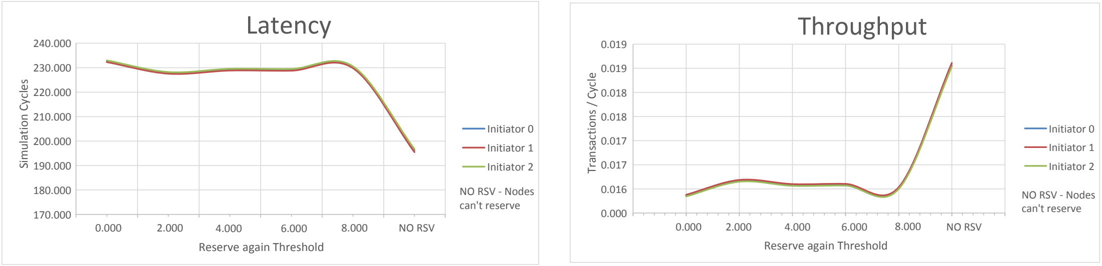


(a) Latency vs reserve-again-threshold

(b) Throughput vs reserve-again-threshold

Figure 5.10: Performance of topology in Figure 5.1 vs reserve-again-threshold. Initiators insert only write requests

- **Scenario 2: All initiators insert only read requests**. Read requests have inherited fairness in their definition because of the good citizen behaviour of nodes and constraints based on the completion buffer size. When the reserve-again-threshold increases, the number of data packets that nodes can reserve reduces. This increases the available bandwidth. The effect of this is an increase in the throughput of initiator nodes (by approximately 40%) and a decrease in the latencies of their corresponding read requests (by approximately 23%) as it is shown in Figures 5.11a and 5.11b.

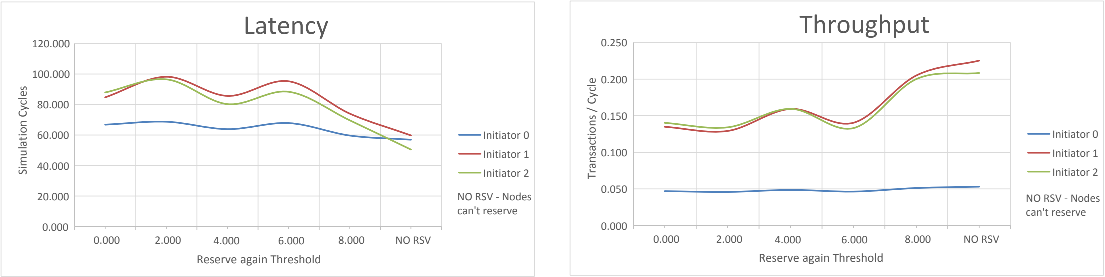


(a) Latency vs reserve-again-threshold

(b) Throughput vs reserve-again-threshold

Figure 5.11: Performance of topology in Figure 5.1 vs reserve-again-threshold. Initiators insert only read requests

- **Scenario 3: Initiator 0 inserts read requests and Initiators 1 and 2 inserts write requests**. As the reserve-again-threshold increases, the same behaviour of write and read requests is observed as in the above two experiments. With an increase in the reserve-again-threshold, the latency of write and read requests reduce (by approximately 33% and 20% respectively) because the amount of traffic in the ring is reduced (Figure 5.12a). Also, as the threshold increases, write requests from initiator 1 get more priority compared to write requests from initiator 2. Thus, with an increase in the threshold, the throughput of initiator 1 becomes more than the throughput of initiator 2. This is illustrated in Figure 5.12b. The throughput of initiators 1 and 2 improves approximately by 60% and that of initiator 0 improves approximately by 4%.

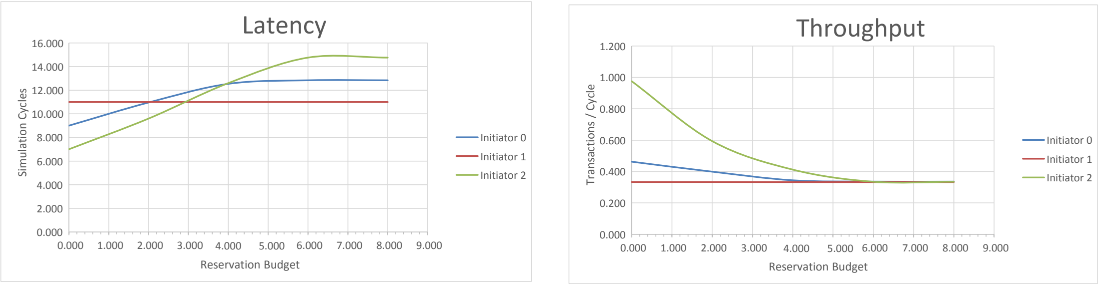


(a) Latency vs reserve-again-threshold

(b) Throughput vs reserve-again-threshold

Figure 5.12: Performance of topology in Figure 5.1 vs reserve-again-threshold. Initiator 0 inserts read requests, and initiators 1 and 2 insert write requests

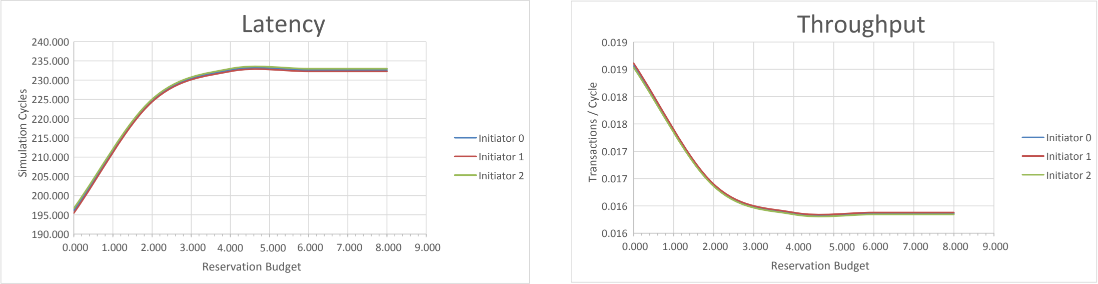
Then, we set the reserve-again-threshold to zero and we vary the reservation budget to study how it influences the QoS the RI provides to its nodes. We use the same topology under the three scenarios explained above. Figures 5.13, 5.14 and 5.15 show how this parameter affects the latency an the throughput of the RI for the three different scenarios. We observe that the results obtained by decreasing the reservation budget are similar to the results obtained while increasing the reserve-again-threshold. The results are more sensitive to change in reservation budget compared to change in reserve-again-threshold. In the experiments performed, changing the reservation budget from 0 to 2 has almost the same effect as changing the reserve-again-threshold form 6 to a very high number (so that nodes can't reserve again). To achieve maximum effective bandwidth, the reservation budget must be set to zero. To achieve equal fairness amongst all initiator nodes, reservation budget must be infinity and the reserve-again-threshold must be zero.



(a) Latency vs Reservation Budget

(b) Throughput vs Reservation Budget

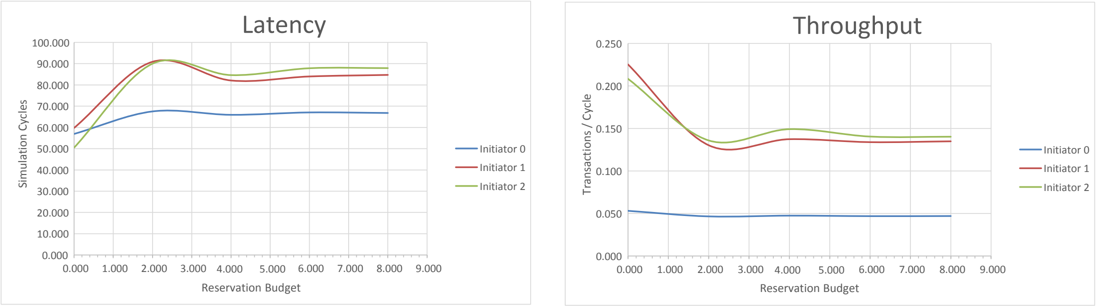
Figure 5.13: Performance of topology in Figure 5.1 vs Reservation Budget. Initiators insert only write requests



(a) Latency vs Reservation Budget

(b) Throughput vs Reservation Budget

Figure 5.14: Performance of topology in Figure 5.1 vs Reservation Budget. Initiators insert only read requests



(a) Latency vs Reservation Budget

(b) Throughput vs Reservation Budget

Figure 5.15: Performance of topology in Figure 5.1 vs Reservation Budget. Initiator 0 inserts read requests, and initiators 1 and 2 insert write requests

5.2.3 SRMD read request related parameters

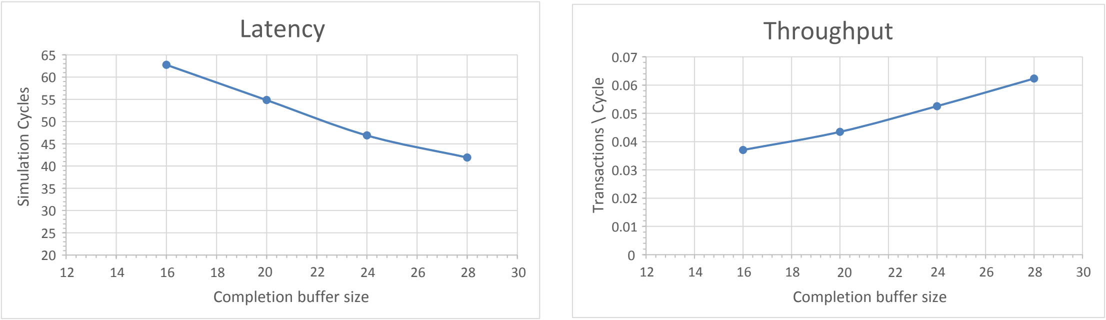
Finally, in this section, we want to analyse how the RI performs to variations in the SRMD related parameters (Completion buffer size, SRMD burst size and incoming port buffer sizes). The completion buffer size of initiator nodes influence the number of back-to-back read requests initiator nodes can insert into the RI. Increasing their size increases the number of back-to-back requests an initiator node can insert into the RI. The incoming port buffer sizes control the number of data packets that are bounced by nodes. Increasing their size reduces the number of data packets that are bounced. Burst size of an SRMD read request is the number of completions an initiator requests from a target node. Large burst sizes allow to read the same amount of data from the target in a lesser amount of time by reducing the number of read commands. On the other hand, it introduces additional traffic on the ring that impacts the performance of other nodes. In the topologies used for our experiments, there is a pipe-stage between all the communicating nodes.

5.2.3.1 Completion Buffer Size

A completion buffer is used by an initiator node to store and send the response to read requests in order to the device. The completion buffer size must be large enough to hold the response of a single SRMD burst request. An initiator node can insert a read request in the RI only if there is enough space available in the completion buffer. Increasing the completion buffer size to enable back-to-back reception of completions reduces read latencies. The minimum size of the completion buffer to enable back-to-back reception of read completions is:

$$min(Completion\, buffer\ size) = Burst\ size + Latency\, of\ read\, request \tag{5.2}$$

The topology shown in Figure 5.1 is analyzed with only initiator 0 injecting read requests of burst size 16. The incoming and outgoing ports of the nodes have buffer sizes of two. Figures 5.16a and 5.16b show how increasing the completion buffer size to enable back-to-back reception of completions help in reducing the read latency and increasing throughput.



(a) Latency vs completion buffer size

(b) Throughput vs completion buffer size

Figure 5.16: Performance of initiator 0 Vs completion buffer size

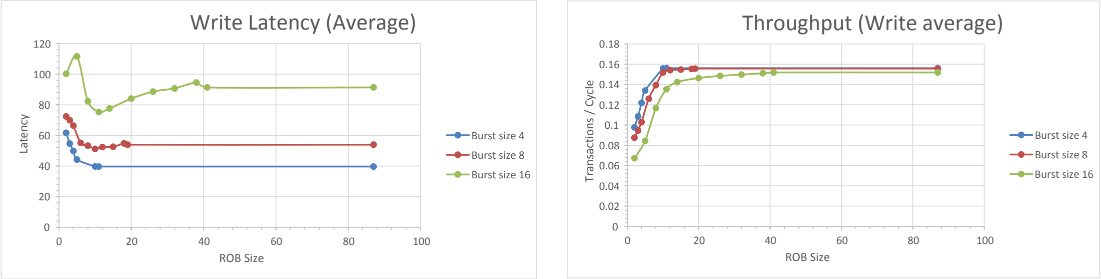
5.2.3.2 Incoming port buffer size and Burst sizes

The incoming port's reordering buffer (ROB) size of a target should be able to buffer all expected requests and prevent bouncing on the ring. The minimum size of the ROB of a target should be:

$$min(ROB\ size) = ceiling(Completion\, buffer\ size/Burst\ size) \tag{5.3}$$

The topology shown in Figure 5.1 is analyzed to understand the effect of burst size and ROB size on topologies that share a target node. In the scenario analyzed, initiator 1 and 2 inject write requests, and initiator 0 injects read requests.

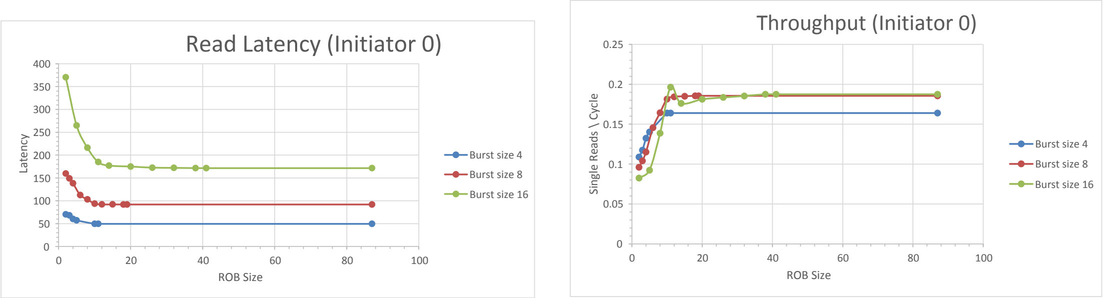
Figures 5.17a, 5.17b, 5.18a, 5.18b and 5.19 show how the ROB size affects the RI performance for different burst sizes.



(a) Latency vs ROB size

(b) Throughput vs ROB size

Figure 5.17: Performance of topology in Figure 5.1 for varying completion buffer sizes and ROB sizes. Initiators insert only write requests



(a) Latency vs ROB size

(b) Throughput vs ROB size

Figure 5.18: Performance of topology in Figure 5.1 for varying completion buffer sizes and ROB sizes. Initiators insert only read requests

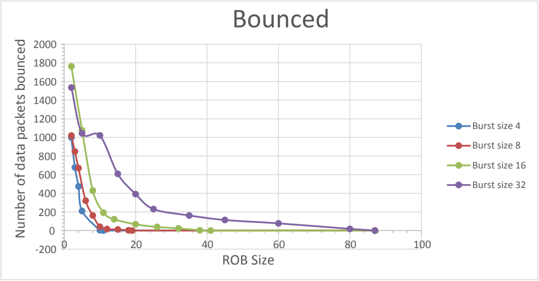


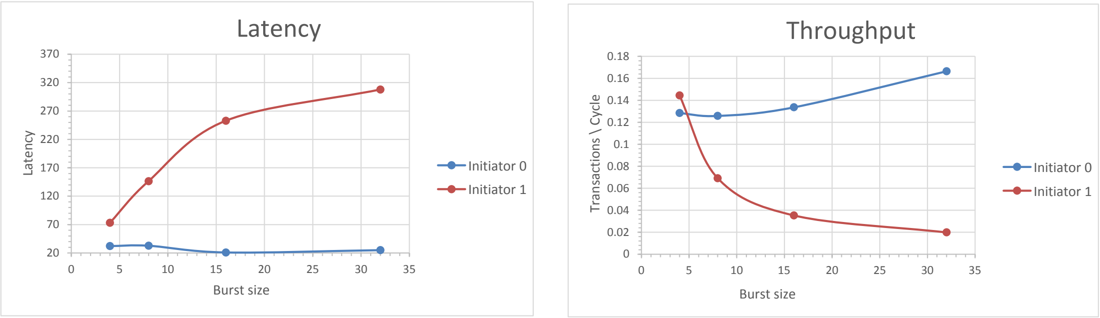
Figure 5.19: Number of data packets bounced by the target node for varying completion buffer sizes and ROB sizes

Figures 5.17a and 5.17b show that increasing the burst size reduces the average throughput of write requests and increases their latency. Smaller burst sizes increase the number of read commands for the target but reduce the number of completions the target has to send out. Increasing the burst size increases the time the target takes to respond to a read request. The resulting effect of which is to increase the latency of write requests and reduce the throughput of their corresponding initiators. Figures 5.17b and 5.17a also show that for a burst sizes 8 and above, as the ROB sizes are increased, the latency of write requests reduce till a point and then increase again. The latency initially reduces because of reduced bouncing of data packets. But after a certain point, the writes suffer because the target is busy responding to read requests.

As shown in Figures 5.18a and 5.18b, we observe that increasing the ROB size decrease the latency of read requests and increase the throughput (Data read per cycle) of their corresponding initiator. Also, as the ROB size increases, the number of data packets bounced are reduced (Figure 5.19). This helps in reducing read latencies and thereby increases the throughput of initiator 0. It can also be observed that increasing the burst size requires larger ROB sizes to reduce the amount of bouncing. In Figure 5.18a we can also observe that the latencies of read requests increase as the burst size increases. Since the number of completions a request asks for increases, their latency also increases. Figure 5.18b shows that the throughput of read requests increase with an increase in the burst size. As a single request asks for more completions, the same amount of data can be read in lesser amount of time.

The topology shown in Figure 5.5a is analyzed to understand how burst size effects the performance of topologies that have multiple target nodes. In the scenario analyzed, initiator 0 injects write requests, and initiator 1 injects read requests.

Figures 5.20a and 5.20b show the performance obtained by increasing the burst size. For this topology, it can be observed that increasing burst size helps in decreasing the latency of write requests and increasing the throughput of their corresponding initiator. Increasing the burst size reduces the number of read commands from initiator 1 to target 1. This reduces bouncing of read commands and also releases bandwidth for the write commands from initiator 0 to target 0.



(a) Latency vs burst size

(b) Throughput vs burst size

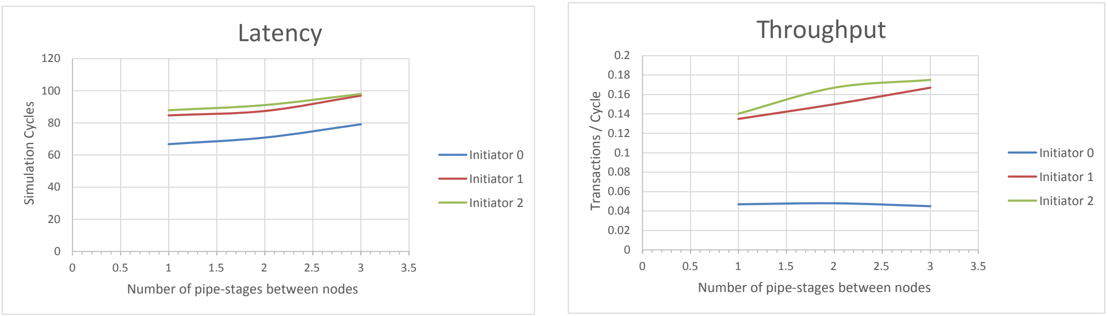
Figure 5.20: Performance of topology in Figure 5.1 for varying burst sizes. Initiator 0 inserts write requests and initiator 1 inserts read requests

Thus, we can conclude that write transactions are very sensitive to burst size of read requests. Furthermore, when increasing the burst size, the latency of write requests also increases in MIST topologies but it decreases in MIMT topologies. Extending ROB sizes help in reducing the latencies of requests by reducing the number of requests that are bounced.

5.2.4 Number of pipe-stages between adjacent nodes

As explained earlier, the total number of data packets circulating in a RI at any given point of time is a constant. In this section, the effect of increasing the total number of data packets that are circulating in the RI on its performance is studied. The total number of data packets in an RI is increased by increasing the number of pipe-stages between nodes. We use the topology shown in Figure 5.1 under a scenario in which initiator 0 and 1 inject write requests, and initiator 0 injects read requests.

We can observe that both the latency of all requests and the throughput increase as the number of pipe-stages between nodes increases (Figure 5.21b and Figure 5.21a). The increase of the latency is due to the distance that a request has to travel from source to destination also increases, whereas the the throughput grows because of the number of data packets that can be used to insert requests increases. Note that the throughput of read requests are not affected much because of the inherent fairness in their definition (good citizen, constraints on completion buffer and the presence of a reserved data packet in the RI for them to insert completions). As explained in the previous subsection, increasing the incoming port's buffer size helps in increasing the throughput of nodes as well as reducing the latency of their corresponding requests. Hence, increasing the incoming port's buffer size rather the number of pipe-stages between adjacent nodes proves to be a better design choice.



(a) Latency vs varying number of pipe-stages between adjacent nodes

(b) Throughput vs varying number of pipe-stages between adjacent nodes

Figure 5.21: Performance of topology in Figure 5.1 for varying number of pipe-stages between adjacent nodes. Initiator 0 inserts read requests, and initiators 1 and 2 insert write requests

5.3 Real World Scenario

An image processing unit (IPU) consists of one or more pipes. Each pipe performs a series of operations on the input image. The output of one pipe serves as the input for another. The execution of pipes are pipelined to make use of the available hardware resources. The Direct Memory Access unit (DMA) transfers data between the Dynamic Random Access Memory (DRAM) of the system and the smaller vector memories within the processors of a pipe. It is also responsible for transferring data from one pipe to another. Also, when pipes are not used, they can be powered down to save power. Thus, from the topology point of view, the RI is always hierarchical. The DMA along with the supervisor node comprises the high ring which is always powered on. Via bridge nodes the high ring is connected to as many low rings as the number of pipes. The lower rings can be powered off when not in use.

In this section, an IPU that contains a single pipe is analyzed. The high ring of the IPU consists of two initiator nodes: I0 and I1. A DMA acts as the devices of these two initiator nodes. The low ring represents a pipe of IPU. It consists of three initiator nodes and two target nodes. The devices of I2, I3, and I4, are a hardware accelerator, and two scalar processors respectively. The devices of T0 and TI are two vector memories.

A real world scenario is used to verify if the RI achieves the performance required by its application. Table 5.2 describes the content of the node hardware mapping XML and the task graph XML of the scenario. For each graph node, its corresponding initiator node, triggers, task ID and maximum execution period is described. If the duration of a task from when it has started to when it completes exceeds its corresponding period, the RI fails to meet the required performance. Figure 5.22 represents the task graph of the scenario. Initially, the required input data for the pipe is transferred from the RAM to the vector memories by the DMA. The transfer is done by graph nodes 2 and 3. Graph nodes 11, 12, 13, 21, 22 and 23 collect the required outputs from the pipe during and after the execution of the pipe and stores them in the RAM. The remaining graph nodes represent the necessary data transfers between the processing units and memories during the execution of the pipe. Table 5.3 describes the task description XML of the scenario. For every task, its destination node and sequence of actions is described. For example, consider a TI leaf node executing task 3. First, the TI leaf node inserts a write request 14 times. It then inserts a read request followed by a write request into the RI 181 times. Finally it inserts a write request 21 times. Table 5.4 summarizes the total number of read and write transactions an initiator inserts into the RI.

The topologies shown in figures 5.23a and 5.23b are used for analyzing the IPU. The topologies are chosen based on the results obtained in section 5.2. In order to reduce latency, the nodes are placed such that:

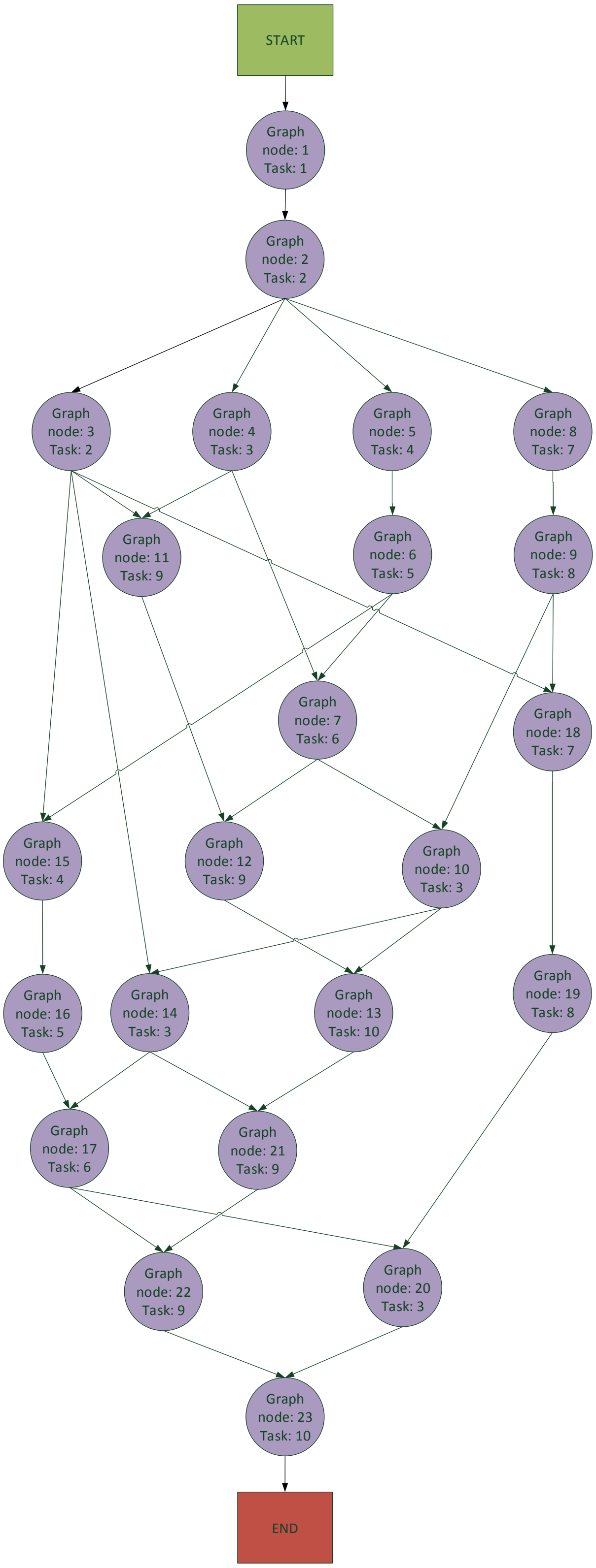
- Write requests always take the shortest path from source to destination.

- Read requests take the longest path from source to destination such that completions are obtained using the shortest path.

The traffic initiator is used to mimic the scenario described. The DMA, hardware accelerators and scalar processors are mimicked by the leaf nodes of the TI. Memories are used as devices of the target nodes. The traffic initiator monitors the execution period of each task. Using the results obtained in Section 5.2, the parameters of the RI are tuned so that all tasks finish within their corresponding maximum execution period. The following subsections summarize the results obtained using different combinations of the incoming port's buffer size, Reservation budget and reserve-again-threshold for varying burst sizes.

Graph Node ID	Initiator	Trigger	Period (Cycles)	Task ID
1	Supervisor	Start	7	1
2	Initiator 0	1	2048	2
3	Initiator 0	2	2048	2
4	Initiator 2	2	2048	3
5	Initiator 3	2	2048	4
6	Initiator 3	5	2048	5
7	Initiator 2	4, 6	2048	6
8	Initiator 4	2	2048	7
9	Initiator 4	8	2048	8
10	Initiator 2	7, 9	2048	3
11	Initiator 1	3, 4	2048	9
12	Initiator 1	11, 7	2048	9
13	Initiator 1	12, 10	2048	10
14	Initiator 2	10, 3	2048	3
15	Initiator 3	3, 6	2048	4
16	Initiator 3	15	2048	5
17	Initiator 2	14, 16	2048	6
18	Initiator 4	3, 9	2048	7
19	Initiator 4	18	2048	8
20	Initiator 2	17, 19	2048	3
21	Initiator 1	13, 14	2048	9
22	Initiator 1	17, 21	2048	9
23	Initiator 1	20, 22	2048	10

Table 5.2: Description of the scenario used to measure the performance of the RI



build 0.18

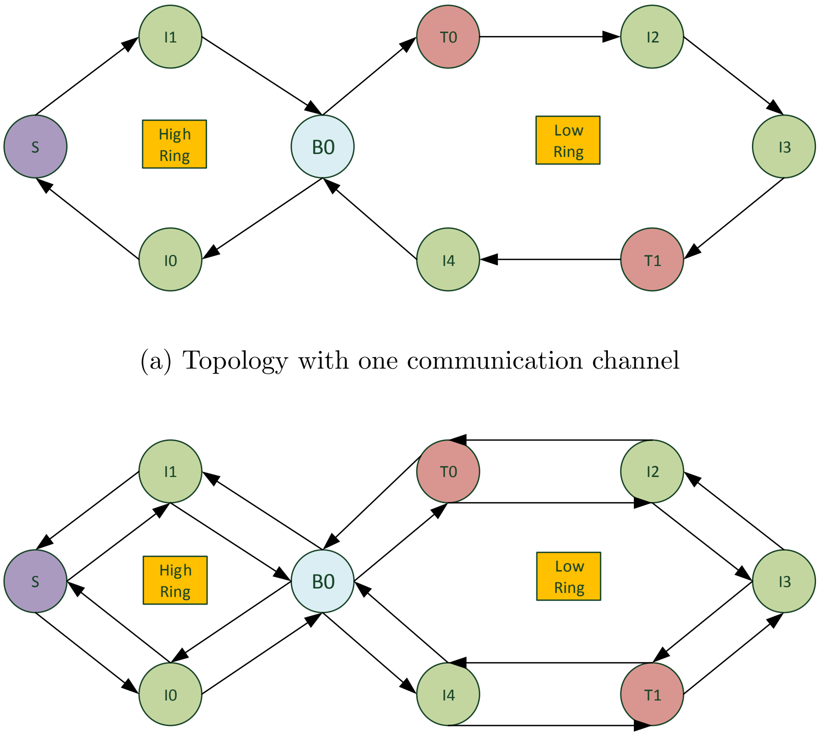
Figure 5.22: Task graph of the scenario used to measure the performance of the RI

Task ID	Destination	Task Description
1	NA	INITIALIZATION
2	Target 0	192 * write
3	Target 0	14 * write → 181 * (read + write) → 21 * write
4	Target 1	16 * read → 160 * read → 16 * write
5	Target 1	384 * write
6	Target 1	14 * write → 181 * (read + write) → 21 * write
7	Target 0	16 * read → 160 * read → 16 * write
8	Target 0	384 * write
9	Target 1	192 * read
10	Target 0	192 * read

Table 5.3: Description of the tasks used to measure the performance of the RI

Initiator	Writes	Reads
I0	384	0
I1	0	1152
I2	780	1196
I3	768	384
I4	768	384

Table 5.4: Number of Write and read requests the initiators insert



(a) Topology with one communication channel

(b) Topology with two communication channels

Figure 5.23: Topologies used to measure the performance of the RI

5.3.1 Single layer RI

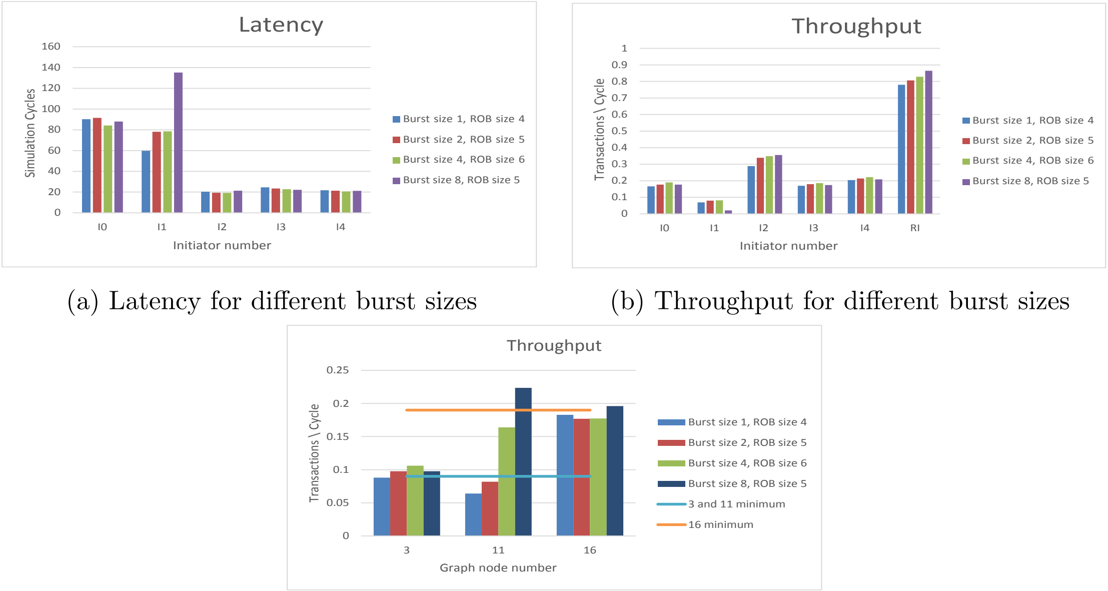
In this section, we summarize the performance of the topology in Figure 5.23a for the scenario described above.

First the topology was simulated for the following values of parameters:

- RB : infinity
- RAT : 0
- Incoming port buffer size : 2
- Number of pipe-stages between adjacent nodes : 1
- Burst size of read requests : 1

For this setup, graph nodes 3, 11 and 16 don't finish within their maximum execution period. As indicated by tables 5.3 and 5.2, graph nodes 3 and 11 send requests from the higher ring to the lower ring. Graph node 16 makes initiator 3 insert write requests for target 1. Its performance suffers because of the traffic congestion in the lower ring. Since the ring is configured for all the nodes to insert requests fairly, reducing the reservation budget or increasing the reserve-again-threshold makes the performance of initiator 3 worse (because initiator 3 is to the left of target 1) because of reasons explained in Section 5.2.2. As explained in section 5.2.3.2, the incoming port's buffer size (ROB size) can be increased to reduce the number of data packets bounced by the bridge and target nodes. This helps to reduce the latency of requests from the higher ring to the lower ring. Also, the number of consecutive read requests inserted by initiator nodes can be reduced by increasing the burst size of read requests. This releases bandwidth for write requests in MIMT topologies.

For these reasons, the performance of the topology for different burst sizes vs ROB sizes were measured. Results obtained follow the same trend as explained in Section 5.2.3.2. The related graphs and explanation are provided in Appendix A.1. As expected, with increase in the burst size and ROB size, the performance of graph nodes 3, 11 and 16 improves. Figure 5.24 summarizes the best performance of the RI obtained for each burst size. It shows that only burst size 8 along with ROB size 5 helps in achieving the performance required by the scenario.



(a) Latency for different burst sizes

(b) Throughput for different burst sizes

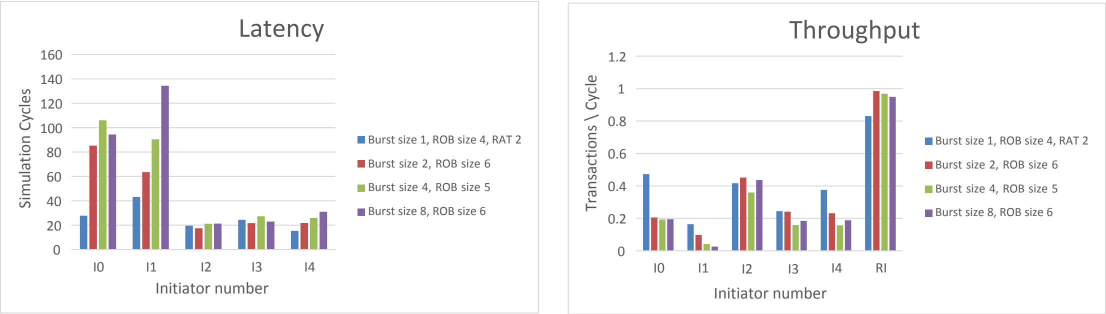
(c) Throughout of graph nodes 3, 11 and 16 for different burst sizes

Figure 5.24: Best performance obtained for different burst sizes when the topology uses one communication channel

5.3.2 Double layer RI

In this section, we summarize the performance of the topology shown in Figure 5.23b. The topology uses 2 communication channels between nodes to provide additional bandwidth for requests. As explained earlier, nodes with multiple channels use routers and arbiters. The routers are configured to send requests on the shortest path from source to destination. In situations of contention, the arbiters forward data packets from their respective inputs in round-robin fashion. For the same reasons explained in Section A.1, the performance of the topology is studied for different combinations of burst and ROB sizes. Results obtained follow the same trends as explained in Sections 5.2.3.2 and 5.2.2. The related graphs and explanations are provided in Appendix A.2.

Figure 5.25 summarizes the best performance of the RI obtained for each burst size. All the burst sizes meet the required performance of the RI.

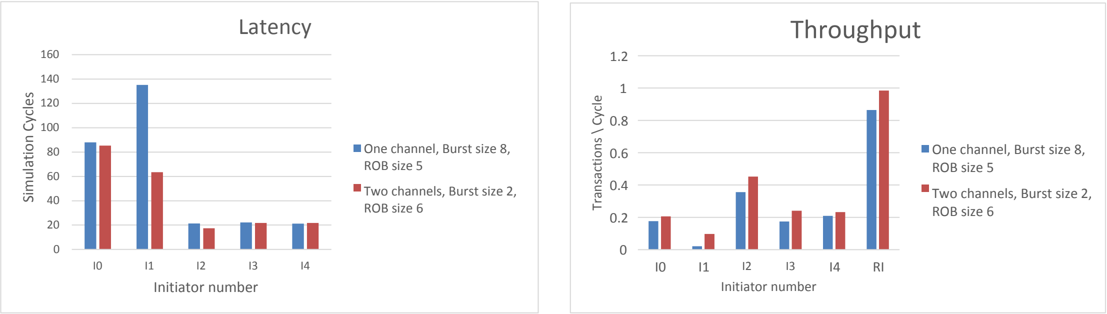


(a) Latency for different burst sizes

(b) Throughput for different burst sizes

Figure 5.25: Best performance obtained for different burst sizes when the topology uses two communication channels

In Figure 5.25 we compare the best performance of the RI obtained for both the topologies in Figure 5.23. The topology with 2 channels improves throughput of the RI by 15% and reduces the average latency of requests by 27%. Thus, based on the requirements of the system, a design trade-off has to be made between the number of communication channels and the performance.



(a) Latency

(b) Throughput

Figure 5.26: Comparison of the best performance obtained by the 2 topologies in Figure 5.23

5.4 Conclusion

Using the TI model described in Chapter 3, here we analyzed the performance of the RI under different parameter configurations. The configurable parameters of the RI are its topology, reserve-again-threshold, Reservation Budget, Incoming port buffer size, Completion buffer size and the total number of data packets circulating in the RI.

First, synthetic traffic scenarios were used to understand the individual effect of each parameter on the performance of the RI. The following can be concluded:

- The type of RI (MIST or MIMT topology) and the position of nodes in the RI greatly influences its performance:
  - In MIST topologies, when all the initiators inject the same type of request and fairly, the throughput and latency of requests scale proportionally as the number of active initiators are increased. When the initiators inject different type of requests, the performance of write requests degrade more than read requests because of reduction in available bandwidth (i.e., the number of unreserved & invalid data packets) for the write requests.
  - In MIMT topologies, the position of the target node with respect to its corresponding initiator makes a big difference in the performance of the node. Write requests perform better when the path of the request from the initiator to the target is the shortest. Read requests perform better when the path of the completions from the target to the initiator is the shortest.
  - RIs with multiple channels reduce the amount of traffic. This helps in decreasing the latency of requests and increasing the throughput of corresponding initiator nodes. The gain comes at the expense of number of wires.
- The reserve-again-threshold and the Reservation budget control how fairly nodes can inject requests. By varying these parameters and observing the performance of the RI, we conclude that:
  - When nodes insert only write requests, increasing the reserve-again-threshold reduces the throughput of nodes that are closer to the target and improves the throughput of nodes that are further away from it. Larger values of the reserve-again-threshold implies nodes can reserve data packets less frequently. This limits the throughput of nodes that are in the path of requests from other nodes.
  - When nodes insert only read requests, lower values of the reservation threshold worsens their performance. Read requests already have inherent fairness in their definition. Low values of the reserve-again-threshold causes reduction in the total available bandwidth that impacts the performance of read requests.
  - Varying the reserve-again-threshold and Reservation budget have the same effect on the performance of the RI but with different sensitivities. The performance of the RI is more sensitive to changes in the reservation budget.
- When we varied the burst size of SRMD read requests, completion buffer size, and the buffer size of the incoming ports, we can observed the following:

  - To reduce the latency of SRMD read requests, the completion buffer size of initiator nodes must be set large enough to allow back-to-back reception of completions.
  - The ROB sizes of incoming ports must be large enough to buffer anticipated requests. Larger ROB sizes help in reducing bouncing requests, reducing their latency and increasing available bandwidth. However, very large ROB sizes are not practical because of area constraints.
  - Increasing the burst sizes of SRMD read requests increases the latency of write requests in MIST topologies. As burst size increases the target node is occupied for longer durations by read requests. This impacts the performance of write requests. In MIMT topologies, large burst sizes help in releasing bandwidth for write requests meant for other target nodes. Since large burst sizes reduce the number of read commands, more bandwidth is available for initiators to insert requests.

- Increasing the number of pipe-stages between adjacent nodes of the RI increases the available bandwidth by increasing the number of data packets that can be used for inserting requests. This helps in improving the throughput of initiator nodes but increases the latencies of requests because of longer paths.

We then used the above-mentioned results to configure the RI for a real world traffic scenario. Based on the traffic scenario XMLs, we chose an RI topology to minimize the latency of requests. Initially, it was simulated with one communication channel between adjacent nodes, ROB size equal to 2, RAT equal to 0, RB equal to -1, burst size of read request equal to 1 and one pipe-stage between adjacent nodes. Two bottlenecks were observed and solved as follows:

- The bridge couldn't insert requests from the high ring to the low ring quickly enough because of high traffic in the low ring. The bottleneck was removed by increasing the ROB size of the incoming ports to free bandwidth by reducing the number of packets bounced.

- In the low ring, write requests suffered because a burst size of 1 was used for read requests. Increasing the burst size helped in releasing bandwidth for write requests. The topology met the required performance of the scenario with burst size 8 and ROB size 6. Changing the reserve-again-threshold or Reservation budget don't help this topology because the write requests are inserted by initiators that are very close to the target.

The same topology was then studied with two communication channels between adjacent nodes. The same bottleneck with respect to the bridge node was observed and solved as mentioned above. This topology met the required performance of the scenario for burst sizes 1, 2, 4 and 8. Moreover, in the most optimum configuration, it improved the throughput of the RI by 15% and reduced the average latency of requests by 27% compared to the topology that uses only one communication channel.

6 Conclusions and future work

This chapter concludes the thesis and highlights improvements that will further improve the performance of the ring interconnect.

6.1 Conclusions

The RI is a new interconnect designed to meet the needs of future generation IPUs developed by Intel. In this thesis, we addressed the need for a flexible framework to test and validate the ring interconnect in pre-silicon stages that will help in making design choices. We used the framework to analyze the ring interconnect and make sure it reaches the performance requirements of a read world use case.

In Chapter 1, we addressed the need of a new interconnect in future generation IPUs. The current generation system bus based IPU has many limitations because of being fully connected. Moreover, it is not designed for the throughput, area and energy requirements of the current system.

Chapter 2 described the Multicore Systems on Chip and the interconnects they use. It highlighted the challenges faced because of the DSM technology and system requirements while designing an interconnect. It showed that the choice of interconnect greatly depends on system requirements. So, the choice of interconnect always has to be made while keeping in mind the trade-off between cost (in terms of area and energy) and performance. We concluded the chapter by presenting some work that has been done in the literature with respect to the ring interconnect.

Chapter 3 introduced the ring interconnect, its protocol and different components. It also introduced the Traffic Initiator model that is used to test and validate the ring interconnect. The Traffic Initiator mimics the sources and sinks of traffic in the RI. It configures the initiator nodes of the RI to insert requests based on a pre-defined task graph. It also monitors the latency of requests and throughput of the initiator nodes.

In Chapter 4, we described how we implemented the test system using the simulation libraries developed at Intel. The test system comprises the RI (Design under test) and the Traffic Initiator (stimuli). The Traffic Initiator and RI nodes were modelled in C++ using the concepts of class inheritance and polymorphism. These behavioural models were then linked with the simulation libraries developed at Intel to obtain a simulator application for the test system. The simulator application along with configuration files that contain information about the traffic scenario are used to simulate the RI in cycle accurate fashion.

Finally, in Chapter 5, we used the test system to analyze the RI. First, we used the Traffic Initiator for Design Space Exploration of the RI. From the study, we concluded the following:

- The performance of a node in the RI greatly depends on the topology and the type of requests circulating in the RI at a given point in time. The order of nodes in the RI greatly influences their performance. Also, the presence of read requests and completion requests in the path of write requests adversely affects the performance of write requests.

- The reserve-again-threshold and reservation budget parameters of the RI largely influence the amount of bandwidth the RI offers a node. They can be configured to meet the QoS requirements. A reservation budget of infinity and reserve-again-threshold of 0 ensures all nodes insert requests fairly.

- Burst sizes of read requests can be configured to improve the performance of both read requests and write requests. In general, large burst sizes improve the performance of read requests.

  Large burst sizes of read requests in MIST topologies degrade the performance of write requests. On the other hand, large burst sizes of write requests in MIMT topologies improve their performance.

- Large sizes of completion buffers used by initiator nodes enable back-to-back reception of read requests. This helps in reducing the latency of read requests and improving the throughput of the corresponding initiator nodes.

- The higher the value of the incoming port buffer size, the lower the number of data packets it bounces. This has an effect of reducing the traffic and improving the throughput of nodes.

- Increasing the number of pipe-stages between adjacent ring nodes helps in increasing the throughput of nodes at the expense of an increased latency of the requests. The effect of improving throughput of nodes and reducing the latency of requests can be observed by increasing the size of incoming port's buffer.

We finally validated the obtained results by executing a real world scenario in the RI to verify if it meets the required performance. Due to the size of the design space, we restricted our analysis to a limited set of possible configurations. We observed that the RI performs better with large buffer sizes and multiple communication channels. Hence, a design trade-off has to be made between the cost and measured benefit.

6.2 Future work

As concluded in the previous chapter, various configurations of the ring interconnect meet the required performance of the practical traffic scenario introduced. However, by considering the following points, better performance of the ring interconnect can be obtained:

- The parameters such as the incoming port buffer size, burst size of SRMD read requests, completion buffer size of initiator agents, reservation budget and reserve-again-threshold were assumed to be the same for all the nodes in the RI. By configuring these parameters separately for each node of the RI, we can save on resources, reduce the amount of traffic in the ring, and improve the performance of individual nodes.

- All the nodes are configured to use the same router and arbiter policy. The arbiter uses a round-robin algorithm to choose data packets from the communication channels for the node. By using priority based algorithms, the performance of certain nodes can be improved at the cost of performance of other nodes. The router uses the minimal routing strategy. It can be configured with adaptive routing strategies to make efficient use of the available resources. This will help in reducing the traffic in the RI and improving the performance of nodes. Adaptive routing strategies come at the cost of increased node complexity because they need information about the system at run-time.

- The parameters reserve-again-threshold and reservation budget control how nodes reserve data packets and hence the amount of traffic in the ring. Based on the traffic condition, dynamically varying these parameters will help in controlling the traffic and improving the performance of nodes.

Similar to the work done in [40], an automated design methodology to provide a custom-tailored, application specific RI that satisfies the performance of the application and design constraints of the system is missing.

Bibliography

[1] L. Chai, Q. Gao, and D. K. Panda, "Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system," in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on.* IEEE, 2007, pp. 471–478.

[2] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier, 2011.

[3] J. Flich and D. Bertozzi, *Designing network on-chip architectures in the nanoscale era.* CRC Press, 2010.

[4] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms.* Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

[5] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, no. 7, pp. 33–38, 2008.

[6] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, no. 11, pp. 32–38, 2005.

[7] S. Saponara and L. Fanucci, "Homogeneous and heterogeneous mpsoc architectures with network-on-chip connectivity for low-power and real-time multimedia signal processing," *VLSI Design*, vol. 2012, p. 16, 2012.

[8] R. Ho, K. W. Mai, M. Horowitz *et al.*, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, 2001.

[9] C. Mohamed Yousuff, V. Mohamed Yousuf Hasan, and M. Khan Galib, "A survey addressing on high performance on-chip vlsi interconnect," *International Journal of Electronics and Telecommunications*, vol. 59, no. 3, pp. 307–312, 2013.

[10] J. Postman and P. Chiang, "A survey addressing on-chip interconnect: energy and reliability considerations," *ISRN Electronics*, vol. 2012, 2012.

[11] A. Jantsch, H. Tenhunen *et al.*, *Networks on chip.* Springer, 2003, vol. 396.

[12] J. Browne, "On-chip communications network report," *Sonics, Inc*, 2011.

[13] C. Pham-Quoc, Z. Al-Ars, and K. Bertels, "Heterogeneous hardware accelerators interconnect: An overview," in *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on.* IEEE, 2013, pp. 189–197.

[14] P. Chaparro, J. González, G. Magklis, Q. Cai, and A. Gonzalez, "Understanding the thermal implications of multi-core architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 8, pp. 1055–1065, 2007.

[15] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[16] J. Duato, S. Yalamanchili, and L. M. Ni, *Interconnection networks: An engineering approach.* Morgan Kaufmann, 2003.

[17] A. C. S. Beck, C. A. L. Lisbôa, and L. Carro, *Adaptable embedded systems.* Springer Science & Business Media, 2012.

[18] N. K. Kavaldjiev and G. J. M. Smit, "A survey of efficient on-chip communications for soc," 2003.

[19] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, pp. 62–76, 1993.

[20] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks.* Elsevier, 2004.

[21] S. Pasricha and N. Dutt, *On-chip communication architectures: system on chip interconnect.* Morgan Kaufmann, 2010.

[22] C. Pham-Quoc, J. Heisswolf, S. Werner, Z. Al-Ars, J. Becker, and K. Bertels, "Hybrid interconnect design for heterogeneous hardware accelerators," in *Proceedings of the Conference on Design, Automation and Test in Europe.* EDA Consortium, 2013, pp. 843–846.

[23] B. M. Al-Hashimi, *System-on-chip: next generation electronics.* IET, 2006, vol. 18.

[24] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 1, 2006.

[25] L.-S. Peh, S. W. Keckler, and S. Vangal, "On-chip networks for multicore systems," in *Multicore Processors and Systems.* Springer, 2009, pp. 35–71.

[26] S. S. Bhople and M. Gaikwad, "A comparative study of different topologies for network on chip architecture," *International Journal of Computer Applications*, 2013.

[27] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *Computers, IEEE Transactions on*, vol. 54, no. 8, pp. 1025–1040, 2005.

[28] M. Mirza-Aghatabar, S. Koohi, S. Hessabi, and M. Pedram, "An empirical investigation of mesh and torus noc topologies under different routing algorithms and traffic models," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on.* IEEE, 2007, pp. 19–26.

[29] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on.* IEEE, 2005, pp. 258–262.

[30] L. Seiler, D. Carmean, T. Juan, M. Abrash, J. Sugerman, P. Hanrahan, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins *et al.*, "Larrabee: A many-core x86 architecture for visual computing," *IEEE micro*, no. 1, pp. 10–21, 2009.

[31] S. J. Vaughn-Nichols, "Vendors draw up a new graphics-hardware approach," *Computer*, vol. 42, no. 5, pp. 11–13, 2009.

[32] C. Fallin, X. Yu, G. Nazario, and O. Mutlu, "A high-performance hierarchical ring on-chip interconnect with low-cost routers," *Computer Architecture Lab, Carnegie Mellon Univ, Tech. Rep*, vol. 7, p. 2011, 2011.

[33] S. Bourduas and Z. Zilic, "Modeling and evaluation of ring-based interconnects for network-on-chip," *Journal of Systems Architecture*, vol. 57, no. 1, pp. 39–60, 2011.

[34] K.-m. Yang, K.-f. Lei, and J.-c. Chiu, "Design of an asynchronous ring bus architecture for multi-core systems," in *Computer Symposium (ICS), 2010 International.* IEEE, 2010, pp. 682–687.

[35] P. T. Wolkotte, G. J. Smit, and J. E. Becker, "Energy efficient noc for best effort communication," in *Field Programmable Logic and Applications, 2005. International Conference on.* IEEE, 2005, pp. 197–202.

[36] H. Samuelsson and S. Kumar, "Ring road noc architecture," in *Norchip Conference, 2004. Proceedings.* IEEE, 2004, pp. 16–19.

[37] I. Saastamoinen, M. Alho, and J. Nurmi, "Buffer implementation for proteo network-on-chip," in *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, vol. 2. IEEE, 2003, pp. II–113.

[38] M. Birnbaum and H. Sachs, "How vsia answers the soc dilemma," *Computer*, vol. 32, no. 6, pp. 42–50, 1999.

[39] S. Bourduas, B. Kuo, Z. Zilic, and N. Manjikian, "Modeling and evaluation of an energy-efficient hierarchical ring interconnect for system-on-chip multiprocessors," in *Circuits and Systems, 2006 IEEE North-East Workshop on.* IEEE, 2006, pp. 201–204.

[40] S. Murali and G. De Micheli, "Sunmap: a tool for automatic topology selection and generation for nocs," in *Proceedings of the 41st annual Design Automation Conference.* ACM, 2004, pp. 914–919.

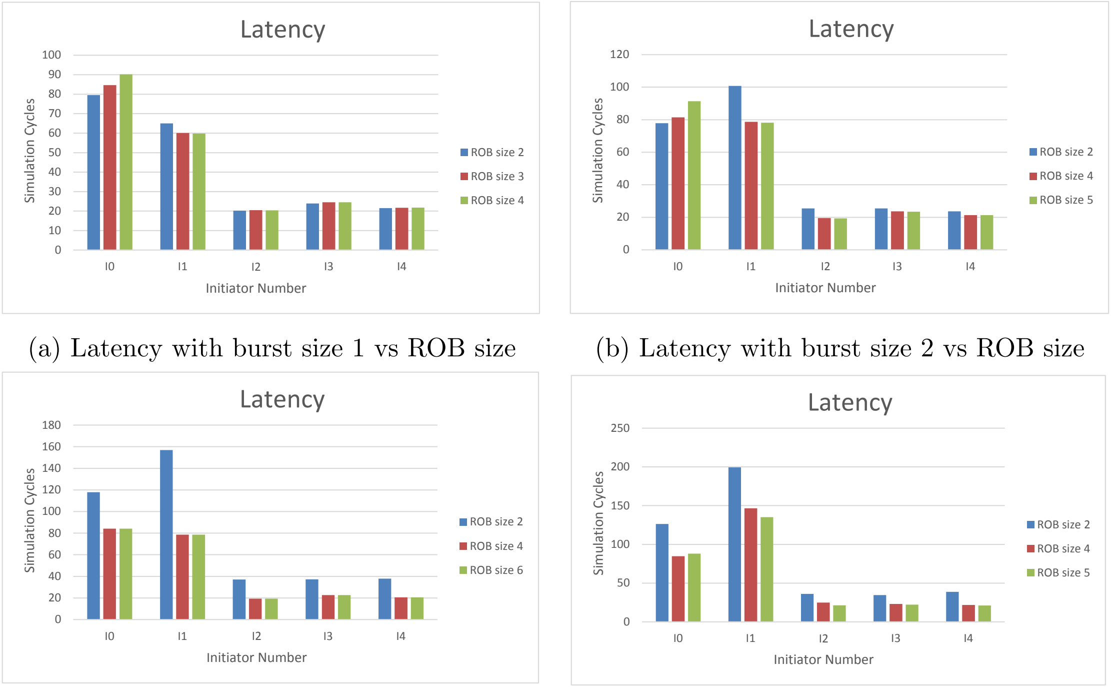
Real World Scenario A

In this appendix, we elaborate on the performance of the topologies shown in Figure 5.23 using the real world scenario explained in Section 5.3. We summarize how different incoming port buffer sizes and burst sizes affect the performance of the topologies.

A.1 Single Channel

In this section, we summarize the performance of the topology in Figure 5.23a. As expected, Figure A.1 shows that the latency of initiators mainly inserting read requests increases with an increase in burst size. For burst sizes 1 and 2, the latency of write requests from initiator 0 increases with an increase in the ROB size because more write requests spend time waiting in the ROB of the bridge. But for burst sizes 4 and 8, the latency of write requests from initiator 0 reduces with an increase in the ROB size because bandwidth in the lower ring is freed (Section 5.2.3).

In Figure A.2, we can see that increasing the burst size helps in increasing the



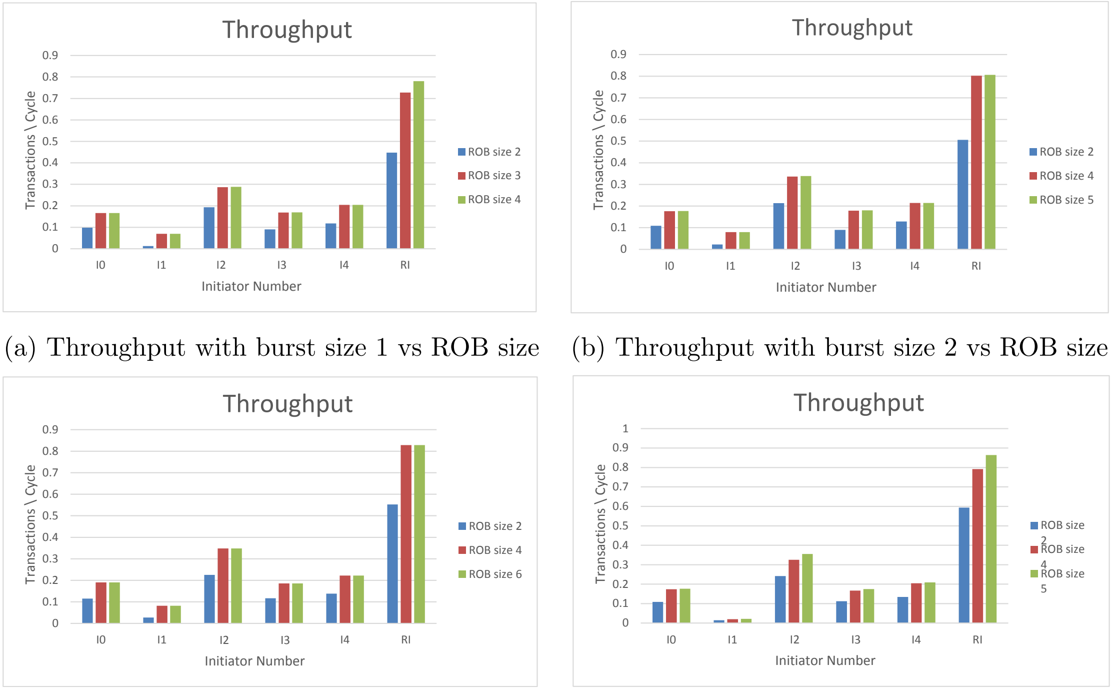
(a) Latency with burst size 1 vs ROB size

(b) Latency with burst size 2 vs ROB size

(c) Latency with burst size 4 vs ROB size

(d) Latency with burst size 8 vs ROB size

Figure A.1: Latency of requests from initiator nodes vs for different burst sizes



(a) Throughput with burst size 1 vs ROB size

(b) Throughput with burst size 2 vs ROB size

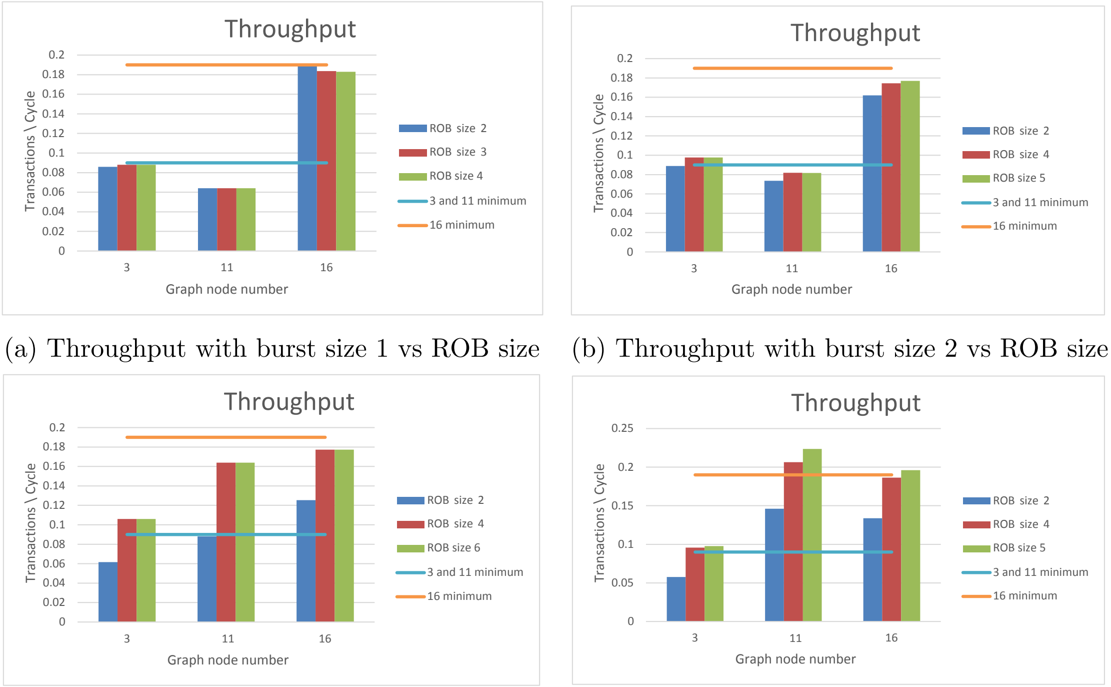
(c) Throughput with burst size 4 vs ROB size

(d) Throughput with burst size 8 vs ROB size

Figure A.2: Throughput of initiator nodes for different burst sizes

throughput of the initiators and the RI.

Figure A.3 shows how the throughput of only graph nodes 3, 11 and 16 vary for different combinations of burst and ROB sizes. The minimum required throughput of these nodes to complete execution before their maximum execution period is also indicated. We focus only on these graph nodes because the other graph nodes always meet the required performance of the scenario for the different combinations of burst and ROB sizes. For the different burst sizes, as the ROB size increases, Figure A.3 shows that graph nodes 3 and 11 finish within their maximum execution period. It also shows the throughput of graph node 16 improving with increase in burst and ROB size.



(a) Throughput with burst size 1 vs ROB size

(b) Throughput with burst size 2 vs ROB size

(c) Throughput with burst size 4 vs ROB size

(d) Throughput with burst size 8 vs ROB size

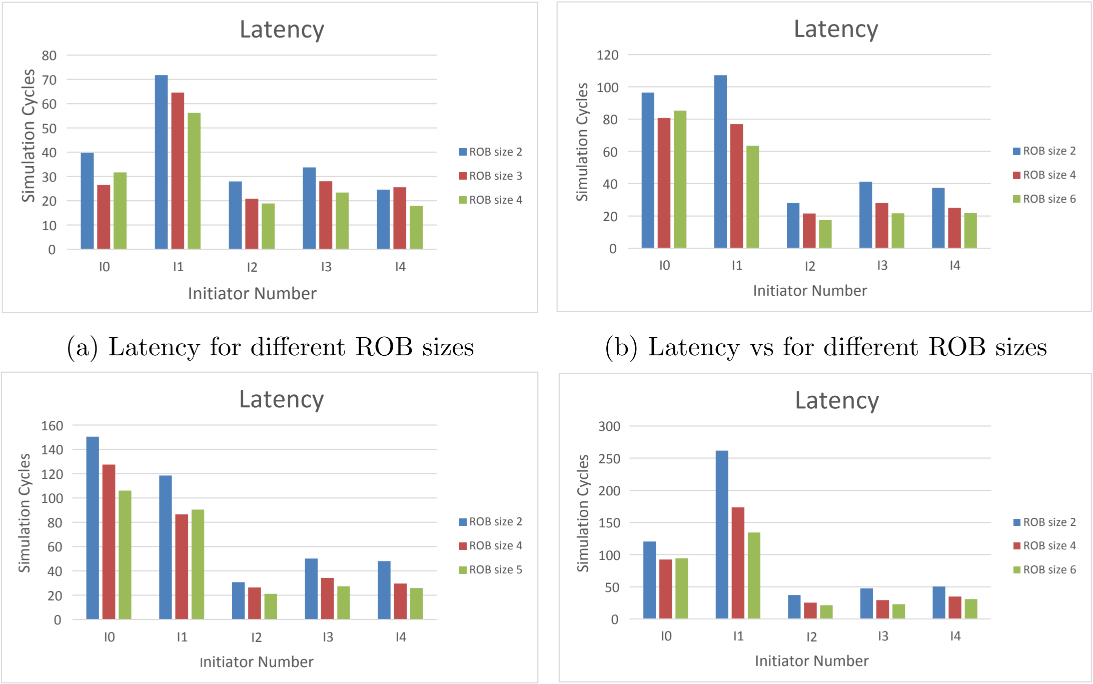
Figure A.3: Throughput of graph nodes 3, 11 and 16 for different burst sizes

A.2 Double Channel

In this section, we summarize the performance of the topology shown in Figure 5.23b. Figures A.4 and A.5 show the latency and throughput of initiator nodes for varying ROB sizes. The results obtained follow the same trend as explained in Section. For different burst sizes, the performance of different graph nodes vary for this topology. The results obtained for different burst sizes are as follows:

1. **Burst size 1**. From Figure A.6a it can be observed that graph nodes 6, 9, 11 and 16 don't meet the required performance of the scenario. As indicated by tables 5.3 and 5.2, graph nodes 6, 9 and 16 insert write requests in the low ring for targets in the low ring. Graph node 11 inserts read requests in the high ring for the low ring. In Figure A.6a we observe that increasing the ROB size increases the throughput of the corresponding graph nodes. For ROB size 4, only graph node 11 doesn't meet the required performance. As explained in Section 5.2.2, varying the RAT improves the performance of read requests. Hence, we observe an improvement in the throughput of graph node 11 by changing the RAT to 2.

2. **Burst size 2 and 4**. In Figures A.6b and A.6c we can observe that graph nodes 3, 16, 19, and 12 don't meet the required performance of the scenario. As indicated by tables 5.3 and 5.2, graph nodes 16 and 19 insert write requests in the low ring for targets in the low ring. Graph node 3 and 12 insert write and read requests



(a) Latency for different ROB sizes

(b) Latency vs for different ROB sizes

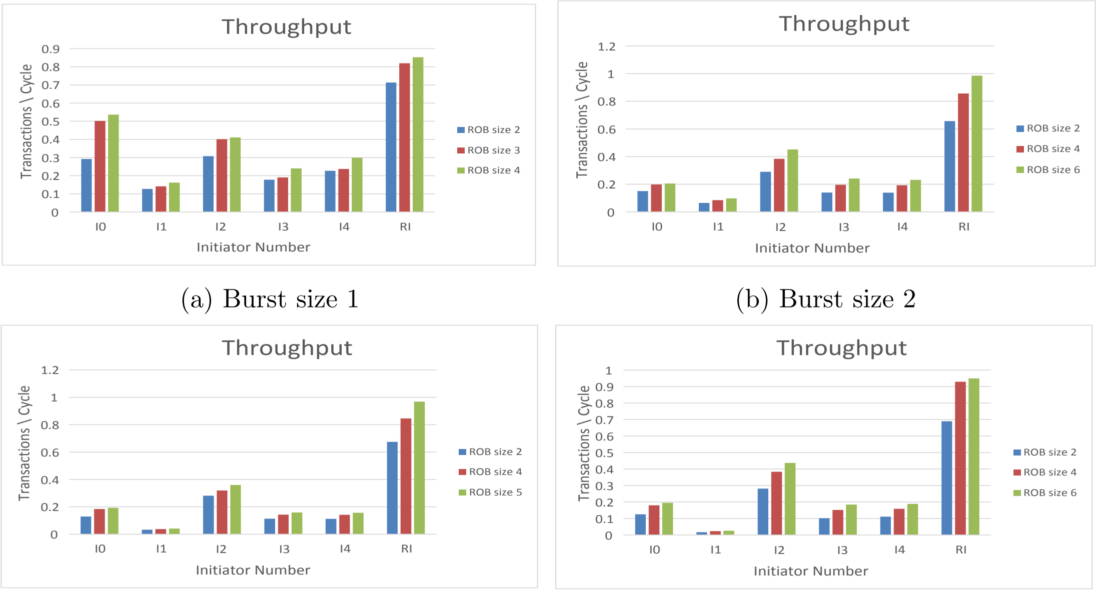
(c) Burst size 4

(d) Burst size 8

Figure A.4: Latency for different burst sizes

respectively in the high ring for the low ring. Figures A.6b and A.6c show that increasing the ROB size increases the throughput of the corresponding graph nodes.

3. **Burst size 8**. Figure A.6d shows that graph nodes 16 and 19 don't meet the required performance of the scenario. As indicated by tables 5.3 and 5.2, graph nodes 16 and 19 insert write requests in the low ring for targets in the low ring. Figures A.6b and A.6c show that increasing the ROB size increases the throughput of the corresponding graph nodes.



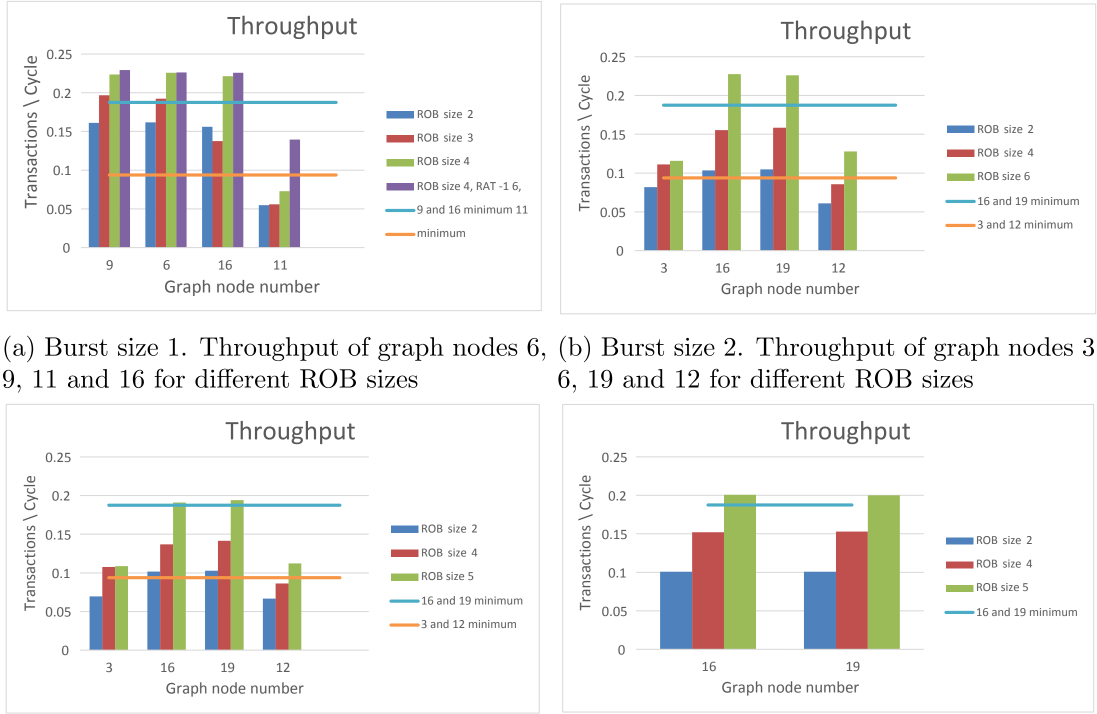
(a) Burst size 1

(b) Burst size 2

(c) Burst size 4

(d) Burst size 8

Figure A.5: Throughput for different burst sizes



(a) Burst size 1. Throughput of graph nodes 6, 9, 11 and 16 for different ROB sizes

(b) Burst size 2. Throughput of graph nodes 3, 6, 19 and 12 for different ROB sizes

(c) Burst size 4. Throughput of graph nodes 3, 6, 19 and 12 for different ROB sizes

(d) Burst size 8. Throughput of graph nodes 16 and 19 for different ROB sizes

Figure A.6: Throughput of graph nodes vs different burst sizes