NOTE TO USERS

Modeling and Evaluation of a Hierarchical Ring Interconnect for System-on-Chip Multiprocessing

Benjamin S. Kuo

Department of Electrical Engineering
McGill University, Montreal

A thesis submitted to McGill University
in partial fulfilment of the requirements of the degree of M.Eng

Your file Votre référence
ISBN: 0-494-06560-5
Our file Notre référence
ISBN: 0-494-06560-5

Abstract

This thesis proposes a software model for a multiprocessor system, which is targeted for SoC implementation. The proposed design is based on the two-level ring architecture adopted in the NUMAchine multiprocessor developed at University of Toronto. The proposed system re-considers the interconnect design alternatives for smaller design area and energy consumption. The system uses an alternative memory architecture to reduce logic complexity, as well as a different deadlock prevention scheme to reflect changes in memory architecture. The software model is implemented in the SystemC modeling language, which allows high-level behavior modeling to reduce both the development time and the simulation time. The model is also at a level of detail which reflects true communication characteristics on the interconnect network. Burst length, memory access latency, FIFO depth, and the operating frequencies for the rings are the four key SoC design parameters which have been identified and optimized for the system.

Résumé

Ce mémoire propose un logiciel qui émule un système à multiple processeurs dédié à la création de systèmes sur puce. La création proposée est basée sur une architecture en anneau à deux niveaux adoptée sur le système à multiple processeurs NUMAchine développé à l'Université de Toronto. Le système proposé réévalue les alternatives d'interconnexion pour une réduction de la surface sur la puce et de la consommation d'énergie. Le système utilise une architecture de mémoire alternative afin de réduire la complexité de la logique de même qu'un nouveau mécanisme de prévention des impasses reflétant le changement dans l'architecture de mémoire. Le modèle logiciel est implanté en utilisant le logiciel de modélisation SystemC, ce dernier offrant une modélisation à haut niveau permettant une réduction dans le temps de développement ainsi qu'une exécution plus rapide des simulations. Le modèle est conçu à un niveau de détail offrant les caractéristiques de communication réelles sur le réseau d'interconnexion. La longueur des fragments, le temps d'accès mémoire, la profondeur des mémoires tampons et les fréquences d'opération des anneaux sont les quatre paramètres clés du système sur puce qui furent identifiés et optimisés.

Acknowledgements

I would like to thank my supervisor Dr. Zeljko Zilic. He gave me the autonomy to pursue my research interests while providing me with relevant references. His guidance has been greatly appreciated since my undergraduate years and for my graduate studies in McGill.

I would like to thank Dr. Naraig Manjikian for graciously accepted to co-supervise my research. His knowledge in parallel computing and simulation model has proved invaluable in my research, I am especially grateful for his participation in the project.

Also a special thanks to Stephan Bourduas for supporting and participating in the project. I have benefited from ideas and solutions from the discussions we had. Stephan also integrated his power modeling framework into the project to allow optimization in the energy consumption area.

I would also like to thank STMicroeletronics for funding the project and providing the development tools which have became an integral part of the project.

For the years during my studies, I have always enjoyed the unconditional support from my family, I am grateful for their sacrifice and endurance to ensure that I can receive the finest education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To survive in the competitive environment today, profit-seeking organizations must be able to deliver the most up-to-date products to the market in relatively short time; more importantly, the cost for designing the products must be small to produce profit. Modern digital circuit designs are commonly implemented on Application Specific Integrated Circuits (ASICs). ASIC implementations are fabricated on silicon and packaged; the packaged device can then be placed on a board for system integration. A system is typically constructed with ASIC devices interconnected with wires on a printed circuit board (PCB).

With modern microelectronic fabrication technology, more transistors can now be packed into a given unit of area on silicon. A new trend in system design methodology emerges as a result of the improved fabrication technology. System-on-Chip (SoC) is a design methodology that places all system components on a single ASIC, and effectively moves system designs from board-level to chip-level integration. The goal of the design approach is to reduce cost and development time for system integration.

Devices designed with the SoC methodology can be seen as ASICs themselves. The main property that distinguishes SoC from ASIC is the more frequent reuse of existing designs [1]. The SoC methodology allows an IC design company to assemble a set of working components into a complete solution efficiently and minimize time-to-market and time-to-profit [2].

1.1 Motivations

When a system is implemented on a single chip, considerable cost savings can be achieved due to the smaller number of components that need to be packaged; as well, the physical dimensions of the final product can be significantly smaller. Performance and power improvements can also be achieved with the single-chip implementation. The SoC implementation is allowed to operate at a higher frequency for the following two reasons: First, signals do not need to be transmitted through high capacitance chip packaging. Second, the physical distance that the signal must travel is significantly shorter. The lower load capacitance from eliminating chip packaging also contributes to lower energy consumption.

Parallel computing systems are commonly known to supply computational throughput which single-processor systems cannot achieve, or to achieve the same level of performance of an expensive single-processor configuration at reasonable cost. Traditional parallel systems are computers connected through a network, or multiple processing units assembled at board-level. The concept of parallel computing can also be applied to SoC designs to improve the computing capability of the system. In addition to performance improvement, multiprocessor SoCs are believed to have numerous advantages over uniprocessor ones [1]. The following paragraphs discuss some of those advantages.

In a single processor system, all tasks are processed by the same processor. Multithreaded single-processor systems divide the computational resource to execute multiple tasks at the same time. While in a multiprocessor implementation, the overall task is divided and distributed to individual processors in the system. The goal of multithreaded processor is to utilize the computational throughput to allow the system to continue processing while some tasks waits for some resources to become available. For applications with high processor utilization, it is apparent that the processor in the single processor system must have more powerful processing elements and run at higher frequency to achieve the same processing throughput. In multiprocessor systems, the processors are smaller and require shorter global interconnect, which eliminates the effect of long latency on the global signals and allows the processors to

run at lower operating voltage.

Multiple instances of processors also contribute to better manufacturing yield. Manufacturing defects in one processor in multiprocessor implementation do not render the whole system useless. The embedded software can be redesigned to redistribute and execute tasks on the remaining processors at minimal cost [1].

Multiprocessor designs enable energy savings by lowering the operating voltage and frequency while maintaining the same processing throughput as single processor implementation. For example, two processors running at half the frequency and half of the supply voltage can save power by factor of four in dynamic power compared to a single processor running at full speed and supply voltage [1]. The 4x power saving in the example comes from simple calculation of the equation:

$$P = \frac{1}{2} f C V^2. \tag{1.1}$$

Development of hardware is often expensive and time consuming. The cost for modifications to a design grows at later stages of development. System modeling with software can be performed quickly and inexpensively to simulate the behavior of the hardware. When the system is modeled in software, developers can quickly modify the key aspects of the system and be able to test the changes by modifying the software code and recompiling the model. The system model is often used for exploration of different design parameters and allows the designer to observe the trade-offs of different architectures in timely fashion. Different algorithm implementations can be tested before the actual hardware development. The system model can also be used as an executable specification which is essentially a program that exhibits the behavior of the target system when executed [3]. The use of an executable specification is especially useful for hardware and software co-design which allows both the hardware and software of the system to be developed in parallel.

1.2 Thesis Overview

This thesis proposes a multiprocessor design targeted for on-chip implementation. The project described in this thesis is based on the architecture adopted in the NUMAchine multiprocessor system developed at the University of Toronto [4, 5]. The architecture is chosen for a list of properties that we believe is beneficial for chip-level implementation. This work has two objectives.

- First, this thesis studies the issues in migrating the board-level hierarchical ring multiprocessor organization to chip-level organization. The migration process includes identifying and modifying components in the original design to be better suited for on-chip implementation. Any potential issues that may be introduced by the modifications should be considered to implement necessary strategies to resolve the issues.

- Second, this thesis presents a software model of the proposed system. The goal of the software model is to verify the functional correctness of the proposed design. The model should be detailed enough to produce realistic traffic on the interconnection network, which can then assist in perform design space exploration on the modeled system.

The proposed system was developed to minimize the design area. The memory architecture of the NUMAchine multiprocessor is identified to have large design area requirements due to complex control logic and cache structure. The proposed design replaces the memory structure with a simpler architecture, which reduces the design complexity and area requirements. A different approach of data communication is implemented to reflect the changes in memory access method. Possibilities of deadlocks are re-examined for the new communication methodology, and proper mechanisms together with an embedded programming methodology are introduced to avoid deadlocks.

A SystemC [3, 6] software model is developed to simulate the proposed system. SystemC is a C/C++ library which provides constructs that can easily describe hardware behaviors. The model can be refined and eventually be synthesized for production. Design tools which can

be used to synthesize SystemC models include: ConvergenSC from CoWare, CoCentric from Synopsis, and Agility Compiler from Celoxica. The interconnection network in the model is developed at a cycle-accurate level of detail such that realistic network behavior can be observed. There are no standard testbenches for multiprocessor systems because applications have different approach for implementation on different multiprocessor architectures. For the proposed system, the types of traffic are categorized, a representative application and three synthetic testbenches are developed to execute on the software model. The testbenches are used for design space exploration and selection of optimal design parameters for the system. The design parameters evaluated in the work include: FIFO depth, data burst length, memory access latency, and relative operating frequency for the rings. Parameters for FIFO depth, data burst length and memory access latency are evaluated to give the best performance versus design area tradeoff, while the values for operating frequencies of the rings are evaluated to give the optimal performance versus energy consumption tradeoff.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides the background of multiprocessor system architectures and SoC interconnection properties, as well as the design issues related to migrating designs to chip-level. Chapter 3 presents the architecture and optimization of the proposed multiprocessor. Chapter 4 presents the implementation of the software model and simulation platform. Chapter 5 presents some parallel programs for the simulation platform, as well as simulations results that are used for selection of optimal setting for some design parameters. Finally, the conclusion and future research directions can be found in Chapter 6.

Chapter 2

Background

Before describing details of the project, this chapter describes the issues in SoC designs. The inefficiency of conventional SoC structure is presented together with the Network-on-chip (NoC) design approach which aims to reduce the cost and complexity of SoC design. A range of network architectures which could be implemented with the NoC design methodology are reviewed to illustrate the cost-performance tradeoffs between different topologies. Design considerations for on-chip network implementation are also described, and the types of network topologies are examined again for on-chip appropriateness. Then, potential issues for NoC designs which includes congestion handling and deadlocks are discussed. The approaches to resolve these issues are also reviewed. Finally, the different classes of multiprocessor architectures are reviewed.

2.1 On-chip Interconnect Properties and Design Considerations

Implementation of interconnect on SoC can have a large impact on designs in silicon. Performance, silicon area, and energy consumption of a design can vary significantly with different types of interconnect design approaches.

2.1.1 Interconnection Issues in Conventional SoC Designs

To reduce the component integration time, the interconnection used in SoC designs must have a standard interface such that integration of IP cores can be done simply and in a timely fashion [7]. Components or IP cores on the SoC must be able to communicate with each other at any time. For large systems, components will communicate simultaneously. The interconnect infrastructure must be able to support concurrent communication such that at any point in time, more than one component can communicate with other components through the interconnect [8].

Modern applications are becoming increasingly complex; SoC designs must incorporate more components and manage communications between them. The interconnections must be scalable to handle a large number of components, meaning that they must provide sufficient bandwidth to ensure reliable communications while maintaining small area. Conventional approaches in SoC design use *buses* or *ad-hoc* connections. A bus is a shared medium where the bandwidth is shared. Because all components connected to a bus are independent entities, initiation of traffic requires either synchronization for the components in addition to contention handling capability, or a bus arbiter to schedule accesses to the bus. The shared medium also has a higher power requirement due to its inherent broadcast-type communication. A message sent from one component is observable by all other components on the bus, which means that each component must drive the capacitive loads for all components connected to the bus, resulting in a higher power requirement. In ad-hoc connectivity, components can have an arbitrary number of connections to any components that communicate to them (no connections if the components do not communicate at all), from the source directly to the target. Without proper organization, the number of connections can be large and routing of the connections in layout can be complex and non-systematic. In addition, the components with ad-hoc connectivity do not have standard interfaces, and require the designers to have a clear understanding of all components so the connections can be made properly. Such an approach requires substantial development time and can be disadvantageous for SoC implementation.

In SoC implementations, the transmission of digital signals on wires is becoming increasingly unreliable, and data corruption can occur due to cross-talk as the chip implementation scales up in frequency while diminishing in die area [9]. To remedy this problem, it has been suggested that the use of packet communication for transporting data is effective at correcting communication errors [8].

2.1.2 Network-on-Chip Design Approach

NoC is the emerging design approach for SoC which satisfies the general requirements for efficient SoC implementation. NoC architectures have standard communication data types, therefore standard communication interfaces can be implemented on each component on the network [9, 10, 8]. A station in computer networks is the processing element that initiates data transmission as well as receives data from other stations through the network. Nodes in a computer network are responsible for forwarding packets, a node can also include a station which transmits and receives packets. A transmission of packets on a network typically passes through more than one node. Because components communicate through different parts of the network, the system can have communications between the components independently and simultaneously.

NoC is similar to general computer networks in that the basic transmission units between two stations are packets; data is formatted in packets before being transmitted through the network. Paths between the stations are divided into link segments where a segment is the path between two adjacent nodes. Most networks today are packet-switched networks where the nodes are switches that route packets to appropriate links. Packets are passed from one node to another node by switching the packet from a link segment to another, and a packet is switched based on the destination information stored in the packet header.

In a network, link segments are joined together to form communication paths between the stations; segments in one path can also be used to construct paths for other stations. Because link segments are shared by different paths, while a packet from one station is moving along a

path, only one segment in the path is active at a time. The segments that are not active can then be used to switch packets for other paths to provide higher communication throughput. The fact that link segments are shared between different paths also reduces the area requirement for the implementation due to a smaller number of connections.

The NoC approach is more desirable than the conventional ad-hoc and bus connectivity used in SoC designs. In general, the NoC approach is a more balanced approach in terms of scalability, reliability, and power tradeoffs; these qualities are not matched by the conventional approaches. Some of the more important properties regarding NoC are summarized below [9].

- NoC facilitates the use of standard interfaces to allow rapid system integration.
- NoC network architectures in general, have high communication throughput.
- Many NoC implementations use packets as the basic communication units, which is especially useful in guaranteeing reliable and efficient data transfer.
- Network traffic control and monitoring capabilities are common for networks, and can potentially be used to manage power and computational resources.

2.1.3 Physical Properties of On-chip Interconnect

For SoC implementation, pre-designed IP cores are used as components in the system. The primary consideration of the design then becomes the interconnect between the components. Migration to on-chip implementation for interconnection networks requires knowledge of both computer networks and the characteristics of digital circuit implementation. Certain properties of SoC can allow simplification of implementation, while others add constraints which must be followed to allow reliable operation.

Physical properties of the chip impose constraints on NoC implementations. Since SoC designs pack the whole system in a single chip, chip area limits the amount of logic which can be implemented. Complex logic typically requires larger area to implement, therefore it should

be avoided or keep to minimal usage in the design. One example of such a consideration is the implementation of retransmission protocol on the interconnect hardware. Retransmission protocol would require a significant number of logic gates, as well as the additional memory required for buffering packets. For example, packet reassembly and a *go-back N* algorithm in the retransmission protocol has large memory requirements. Implementation of such complex protocols in software can limit the design area at the cost of lower performance.

In computer network, retransmission protocol is used to retransmit lost or corrupted packets. Error correction algorithms such as forward-error-correcting scheme can be used to correct data corruptions which occurred during transmission, and would improve system performance because the packet does not need to be retransmitted. Error correction algorithms, however, can not replace packet retransmission protocol in the case of packet loss due to congestion. Mechanisms such as flow control can be implemented to eliminate packet loses due to congestion and the need to have complex packet retransmission logic. In a switched network, simpler switching logic can provide higher bandwidth simply because switching of packets can be completed in less time, which allows the network to switch packets at a higher frequency [11].

In addition to the area limitation, smaller buffers are desired for real-time applications. While a large buffer space is desirable for handling congestion, it increases the average latency for packets to go from one node to another. In congested locations, packets are buffered in a queue and wait for their turn; a large buffer allows more packets to be stored, thus increasing the average waiting time for a packet to get transmitted. In systems with large buffers, the time a packet spends in the interconnect network can vary significantly, therefore is not suitable for real-time applications. The latency variation can be reduced with smaller buffers in the interconnect network. The smaller buffers reduce the number of packets in the network, which also reduce the time that each packet needs to wait while in the network.

Computer networks have an inherently unreliable communication medium. Packets can be corrupted or even lost in transmission. Computer network applications typically are not timing critical, and do not necessarily require high performance communications; packet losses are

resolved with retransmission protocols such as TCP. On-chip implementation of a communication network, however, can provide better performance, and have a fixed network topology. For an on-chip implementation, a node in the network will not be randomly disconnected from the network and lead to loss of packets. The elimination of such non-determinism also removes the requirement of having a complex protocol to handle packet losses.

It is also important to note that NoC imposes energy constraints [9, 12, 13, 14]. Because all components of the system are implemented to fit on a single chip, the level of energy consumption must normally be at a reasonable level such that the system will not overheat and become unreliable. The energy constraint is also one of the primary concerns of embedded systems where SoC implementation is common. Based on the dynamic power equation in Equation 1.1, it is apparent that a system that must drive more capacitive loads would consume more power at a fixed clock rate.

Lastly, where the computer network allows the topology to be determined at later time, topology and architecture of NoC implementation must be specified and fixed at design time. Evaluation of interconnect network alternatives must be done at early stages of the design.

2.1.4 Network Topologies and On-chip Suitability

The concept of Network-on-Chip originates from computer networks [13]. Most of the computer network topologies can be used for on-chip implementation. As previously discussed, evaluation and selection of the on-chip interconnect network suitable for the application must be done in early stages of the design. The most common network topologies used in connecting network components are summarized below.

- A *linear* topology connects a node to two neighboring nodes with the exception of the end nodes which connect to only one neighbor.

- A *ring* topology can be considered as the linear topology with the two ends connected through a link. An extension of the ring network is to have the nodes connected with two

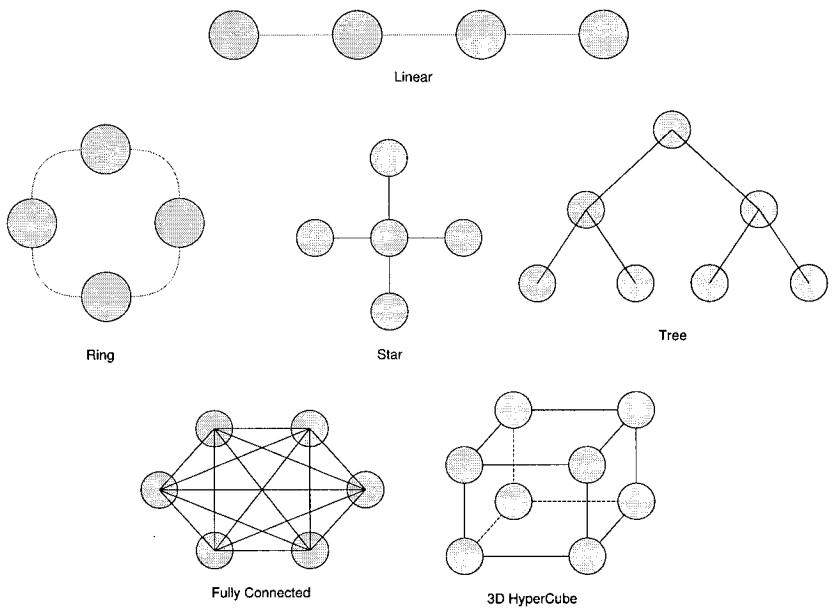

Figure 2.1: Network topologies

rings, each communicating in the opposite direction of the other.

- In a *star* topology, a central node is connected to all of the remaining nodes. In this network, each node can communicate to only the central node, and the distance from one node to another is fixed to two hops.

- A *fully connected* network connects each node to all of the remaining nodes in the station with dedicated links. In such a network, each node can communicate to another directly with minimal latency.

- A *tree* topology is an extension of the star network; it can be treated as a collection of star networks connected together in hierarchical fashion.

- In a *hypercube* network, each node is connected to n other nodes. A *1-n* hypercube is equivalent to a linear network, whereas a *2-n* hypercube is equivalent to a ring network. The hypercube network shown in Figure 2.1 is a *3-n* hypercube.

Similar to the tree topology, some of the networks described above can be implemented in a hierarchical fashion. The switch-box network [15], the butterfly fat-tree network [16] and the octagon network [17] are examples of hierarchical extensions of the basic network topologies described above. The work described in this thesis is based on hierarchical ring topology.

Not all networks are built equally; different network topologies have different performance and cost characteristics. The linear topology is the simplest network. *N* stations are connected with *N-1* links. The performance of communication between two nodes is, however, highly dependent on the location in the network. For the two end stations to communicate, packets must pass through all other nodes in the network. The problem becomes significant when there are large numbers of nodes in the network. The ring network improves the performance of a linear network by allowing the end nodes to communicate directly. For large networks, however, a packet still must pass passed through many nodes to get to the destination. In the star topology, packets only need to pass through the central node to reach the destination. The topology is simple and provides good performance, but the central node must have a number of interfaces equal to the number of leaf nodes in the network, therefore it is not physically feasible for large network. In addition, a single point of failure at the central node would render the whole system useless. The hypercube network is commonly used for parallel computing; the topology is flexible and allows performance improvement for larger systems by increasing the dimension of the hypercube. A higher dimension hypercube, however, faces issues of connectivity and higher interface complexity as the number of connections per node increases. The decision for selecting a suitable dimension for the hypercube that provides a good balance between performance and interconnect complexity is left to the system designer. Lastly, the fully connected network provides the best performance as there is only one dedicated path for a node to each node in the network. The cost associated with this performance is the large number of links and interfaces at each node, which are expensive to implement.

Previous discussions have indicated that NoC implementations offer a better balance between power requirements, interconnect complexity, and higher bandwidth. Of all the types

of interconnect architectures, however, some architectures have characteristics that are better suited for on-chip implementation.

Because NoC designs in general allow concurrent communications among the nodes, and use standard interfaces for ease of integration, selecting a suitable type of network then is based on the area requirements, implementation complexity, and performance goals of the given project.

It was discussed previously that the ring and linear networks have below average performance, but can be implemented easily and have low area requirement. The tree and star type of networks have good performance due to small number of nodes a packet must pass to reach the destination. Their implementation is simple, but the interface requirements at the forwarding (central) nodes are complex and expensive for a large network. The central node in the star network must also have large buffers to handle packet contention. Crossbar and fully connected networks have excellent performance because their links are dedicated, but the layout for these architectures can be difficult due to the large number of links. The large number of links also adds to a higher area requirement for the implementation. A hypercube network resembles the linear or ring network for lower dimensions. For a hypercube with dimension 3 or greater, the network becomes complex as the number of interfaces and links increase. Efficient routing algorithm also becomes non-trivial for hypercube with dimension greater than 2. For comparison purposes, the analysis on the relative performance and cost for the types of architectures discussed here are summarized in Table 2.1.

	Linear/Ring	Tree/Star	Crossbar/Fully connected	Hypercube 3-n+
Performance	below avg	above avg	excellent	avg
Area	small	avg	large	avg large
Complexity	simple	avg	complex	complex
Power	below avg	below avg	above avg	avg

Table 2.1: On-Chip Network Architecture comparison

Of the types of network described, all links have point-to-point connectivity. Assuming equal length, the energy consumption per link is the same in the networks. Total energy con-

sumption can then be assessed based on the number of links within the network. Linear, ring, tree and star networks all have a number of links equal to $(N-1)$ for N-node networks. Crossbar and fully connected networks have a large power requirement due to their large number of links. The number of links in hypercube networks grows with their dimension, but the upper bound of the number of links is still below crossbar and fully connected networks.

Based on the studies and comparisons of the various network topologies, we believe that the hierarchical ring network is suitable for on-chip implementation. The architecture preserves the properties of small area requirement and simplicity offered by the ring network, but improves the performance by dividing the nodes hierarchically into smaller rings connected through a central ring. The hierarchical nature also reduces the average distance between the nodes and improves the communication latency. More detailed descriptions on the characteristics of hierarchical rings and their suitability for on-chip implementation is discussed in Section 2.4.

2.2 Congestion and Deadlock Handling in NoC

2.2.1 Congestion

Congestion of traffic in a network is the condition in which there are large amounts of data in the network, while the network does not have sufficient bandwidth to handle a sustained level of traffic. How the network handles congestion can have a large impact on the performance and cost of the system.

Generally, congestion in a network is caused by having too many sources attempting to send data at too high a rate [18]. In a multi-hop switched network, a packet must pass through more than one switch to reach the destination. Each switch is responsible for routing the packet to the path that leads to the destination. Because the physical connections in the network are shared and used by different paths, collisions between packets can occur when two packets arrive at a switch simultaneously, and both packets are expected to be switched to the same output.

In a packet-switched network, the ideal behavior of a switch buffer is to store packets temporarily when they cannot be switched immediately. Implementation of such behavior, however, requires the buffers to have infinite capacity, which cannot be realized on any physical device. The size of buffers in NoC is limited. Problems then arise in a highly utilized network when the buffers are filled up at a rate greater than the rate at which they are drained. When the buffers are full, further incoming packets can cause the buffers to overflow, leading to loss of packets.

2.2.2 Approaches to Congestion Control

Congestion can be handled with one of two approaches [19]. The first approach allows packets to be dropped during congested periods and to be retransmitted as needed. The second approach requires the network to have lossless packet transmission, and congestion is handled through a flow-control mechanism which throttles the transmission rate at the sending side of the network. The first approach relies on the end-to-end protocol to recover the lost packets, allowing simple network implementations. TCP is one example of congestion control used commonly in computer networks. The drawback of packet retransmission is that the sending and receiving ends must support the protocol, which may be complex. The required acknowledgment packets add more strain to the network when it is congested. The retransmission of packets also adds traffic to the network. The retransmission approach is not ideal for solving the congestion issue because it adds more traffic to an already congested network where the cause of retransmission originates from an excessive traffic level in the network.

The lossless packet transmission approach uses a technique whereby a switch at the receiving end can inform the sending end of the busy status. Depending on the busy status, the sender can reduce the transmission rate or temporarily halt data transmission when the receiving switch is busy. When the receiving switch gets out of the busy state, another signal is sent to the sending switches to resume transmission. The mechanism for the congested switch to inform the sending switch of the busy status is called backpressure feedback. While the back-

pressure network guarantees lossless transmission of packet, it adds more complexity to the network implementation and is susceptible to a condition known as deadlock [19].

2.2.3 Deadlock

Deadlock is a condition in which the throughput of a network or part of a network goes to zero due to conflicts in resource acquisition [20]. Previous work has been done in identifying different types of deadlocks, and solutions have been proposed to resolve deadlock issues in computer networks [21, 19, 22]. This section summarizes the types of potential deadlocks in a network and lists some of the solutions.

2.2.4 Types of Network-level Deadlocks

Direct store and forward deadlock occurs when there are two stations each holding resources required by the other station and each station is in the state of acquiring the desired resource from the other station before it will release the resource it currently holds. The scenario is a generic deadlock example. In a network, the resources in question are the transmit buffers and receive buffers for the stations.

Indirect deadlock is a more general case than direct deadlock. The indirect deadlock scenario involves more than two stations. Each station involved in the deadlock waits for a resource from another station which has dependency on a resource held by a third station. The term indirect comes from the fact that the source of blockage in the network propagates through a chain of stations which ultimately halt the data flow. Indirect deadlock can be detected by finding the presence of circular dependencies on the resources in the network.

In some networks, data is transmitted through the network in units of packets, where data is divided into smaller pieces embedded in packets. When the packets arrive at their destination, the data portion of each packet is extracted and reassembled with others into the original data. Before data reassembly, all relevant packets are stored in a buffer waiting for all parts of the data to arrive. When all parts have arrived, the contents of packets are extracted and

reassembled. Once reassembly of data is complete, the relevant packets can then be removed from the buffer. Normally, a station in the network can receive packets representing more than one dissembled data items. If the reassembly of data on the receiving station are not properly handled, *reassembly deadlock* can occur. When the receiving station fills up the receive buffer with incomplete data, it will not be able to accept more data and throughput in this part of the network is reduced to zero.

Piggyback of a transmit acknowledgment on a packet going to the original sender is an approach to reduce the network traffic. *Piggyback deadlock* occurs when the packet that carries the acknowledgement is blocked at some point in the network, and the station that causes the blockage requires the acknowledgement to release the resource. The circular dependency can be observed by noting that the station blocks the acknowledgement packet but needs the acknowledgement to unblock the network.

In networks with virtual end-to-end links, reservation of resource for the link at each node on the path must be performed. The resource reservation is achieved by sending a special packet along the path that reserves resources on each node in the path. Because the reservation packets move in the network in the same way as data packets, if the virtual link is established dynamically, deadlock scenarios similar to the ones described before can occur regardless of the type of packet (either they are data packets or virtual path setup packets).

With all the deadlock solutions implemented in a network, the network can still be susceptible to deadlock. In networking terms, the interconnection between the stations is equivalent to the physical layer. A deadlock-free physical layer does not guarantee a deadlock-free system if there are deadlocks in the application layer. From the user's point of view, the network can be viewed as a medium responsible for transmitting data from one place to another reliably. At a higher abstraction level, the user does not need to understand how the network is constructed, but only needs to know that data will be transmitted to the destination. However, if applications on the stations contend for resources at the application level, deadlock can still occur if no precautions are taken. Deadlocks at the application level can leave unconsumed packets in

the network. Because the interconnect network has finite buffers, blockage of data flow can occur which ultimately leads to a complete halt of the system.

2.2.5 Deadlock Solutions

Research has been carried out to find deadlock solutions over the past decades and many solutions have been proposed for handling deadlocks in a network environment. However, solutions for deadlock can have drawbacks such as degraded performance or expensive implementation. Applications with high bandwidth requirements must consider interconnect architectures which provide higher bandwidth, as the lower bandwidth interconnects can be constantly over-utilized in such applications.

A *redundant network* is used to provide an alternative path for data from one station to another. Deadlock can be prevented by providing an alternative path for data transmission when congestion is detected in the primary interconnect network. Designers are required to explicitly choose the path by which the data is sent. A common use of such an implementation for request/response type of communication is to have request packets taking one path on the primary interconnect network, and response packets using another path on the alternative network. This method does not solve the problem, but rather it tries to defer the problem by adding more system resources. With one redundant network, the system is effectively doubled in bandwidth. When one network is blocked, the second network can continue to operate and transmit data. However, the fundamental problem is still unresolved: the same cause of deadlock on the first network can also cause deadlock on the second network. The second issue of using redundant network to solve deadlocks is that the network is likely to be under-utilized, and hence not cost efficient. As the network is not expected to be busy most of the time, the second network is only implemented for occasional usage when deadlock occurs in the first network.

The *Disha* deadlock recovery scheme [22] is a variation on using a redundant network where the possibility of having deadlock at the second network is removed. In Disha, a special

deadlock buffer is implemented at each switch in the network which forms a floating virtual channel. The purpose of the virtual channel is to provide an alternative path for the packets to by-pass the location of congestion. The virtual channel is predetermined and is made deadlock-free through predetermined priority switching or use of a token to provide exclusive access to the path to one source. The virtual channel is only used when network traffic congestion is detected. Although congestion in the network does not necessarily represent deadlock, it is a necessary condition for deadlock to occur at the network level.

For stations having only one type of buffer for both sending and receiving data, Store-and-Forward deadlock can happen when two stations try to swap data with total data size greater than the available buffers. In the described scenario, the shared buffers at both stations are filled up by outgoing data. Both stations cannot receive more data as the buffers are full, and no forward progression can happen as processes at both stations are waiting for the other station to free up space in its FIFO by removing transmitted packets. However, transmissions at both stations are blocked. Blocking of traffic continues as long as data cannot be drained from the buffers. The simplest approach to resolve such deadlocks is to have *separate transmit and receive buffers* for each station. With the separate buffers, transmission of data at a station is not affected by the receive side of the same station and vice versa. The independent send and receive buffer approach also has the advantage of simpler control logic for maintaining the packet information, i.e., monitoring of the incoming and outgoing packets.

A virtual path is a chain of switches with resources (buffers) reserved for the specific path from a source station to the target station. When the virtual path is established, the path is guaranteed to have all of the required resources reserved for the path. Once established, the virtual path ensures that packets can move through the network without having to wait for the resources used by other paths. Each switch on the virtual path has buffer space reserved for just that path. Data transmitted from the sender to receiver through the virtual path use only the reserved buffers on the switches. The connections between the switches are still shared between different virtual paths with appropriate scheduling scheme. When one of the virtual

paths is blocked, the scheduling scheme still allows other virtual paths to transmit data during the time which is allocated to them. Thus there is no blockage of data which can potentially cause deadlocks to occur.

A variation of the virtual path deadlock solution called *resource ordering* is to have different classes of packets. When forwarding packets, priority is given to a class with higher priority packets such that when lower priority packets are blocked in a deadlock, high priority packets can still move through the network and hopefully provide sufficient conditions to resolve the deadlock. With such an approach, packets which can potentially release resources are given higher priority. Acknowledgment packets and read response packets are typically classified for higher priority in such a deadlock resolution scheme. An alternative way to determine the priorities of the packet is based on the distance a packet has traveled [19]; packets are given higher priority when they have traveled a longer distance from the original sender. The resource ordering approach, however, requires that the priority information be embedded in the packet header, and thus requires more bandwidth to transmit a fixed amount of data.

Improvements to the resource ordering approach have been made in a solution proposed by Karol, Golestani and Lee [19]. The proposed deadlock prevention scheme is to be adopted in lossless back pressured networks where the backpressure mechanism is used to inform the potential senders to a congested area. When the backpressure mechanism is activated at one node due to congestion, nodes connecting to it will stop sending packets to the congested node. The method proposed in their study uses the distance that the packets have traveled to determine the packet priority. The backpressure mechanism is modified for the solution. Instead of embedding the priority information in the packet header, a *priority table* is implemented in each node of the network. Based on the level of congestion (e.g., the FIFO level), transmit feedback is sent from the receiving node back to the sending node. The sending node compares the transmit feedback with the priority table to determine which packet can be sent to the receiver. When the receiving node receives the packet, the priority table is updated. Based on the transmit feedback received from the third node, the sending node will send the lowest

priority packet (which still has higher priority than the received feedback) to the next node. With the use of transmission feedback and a priority table, the priority information is computed in the switches, thus removing the need to include the priority information in the packet header.

2.3 Types of Multiple Processor Architectures

Multiple processor architectures can generally be categorized into two groups: shared memory and distributed memory. In shared memory architectures, the system has one global memory space shared by all the processors. All processors have access to the whole memory space and writes performed by one processor are visible to all other processors. In shared memory architectures, data is not transferred until a processor tries to access it. Parallel programs in shared-memory multiprocessor systems are, in general, easier to code if performance is not the primary concern. Equipped with cache-coherent hardware [5], one can typically parallelize a single-processor program with minimal modification. This flexibility, however, does not guarantee good performance.

In distributed-memory systems, each processor has access to an independent memory space, and data is shared through explicit message passing from one processor to another. Because the message passing mechanism requires one receive operation to match each send operation, program developers are required to understand concepts of parallel programming to code efficiently.

There are also hybrid systems that combine both parallel programming models. The distributed shared-memory systems have locally-shared memory space for groups of processes, but all processes have shared memory globally. In such cases, processes will access locally shared memory with the shared-memory approach; when the processes try to access memory located remotely, communication is performed by exchanging messages [20].

The boundary between shared memory and message-passing models becomes less clearly defined as software and hardware advance. Message-passing operations can be supported on

most shared memory machines through shared buffer storage. The global address space used in the shared memory model can be constructed in software on a message passing architecture by implementing remote memory accesses with a set of message exchanges between remote stations [20].

2.4 NUMAchine Multiprocessor

The NUMAchine multiprocessor has been developed at the University of Toronto as a part of research project on hardware and software for parallel computing. The platform is a non-uniform-memory-access, distributed-shared-memory system that consists of 16 stations connected by a hierarchical ring interconnect [4, 5]. The NUMAchine multiprocessor system was implemented with commodity components; control logic was implemented in programmable logic to allow modifications to hardware without any physical changes to the printed circuit boards.

The work described in this thesis is based on the NUMAchine multiprocessor architecture, with the primary focus placed on on-chip implementation. The architecture was chosen because it has a planar topology with point-to-point connectivity, it offers the possibility of supporting a flexible clocking scheme, and it satisfies previously discussed requirements for an SoC interconnect including concurrent communication and can be implemented with standard interface.

The topology is planar; the network connectivity can be drawn on paper with no lines crossing other lines. This property simplifies layout, and the layout area can also be reduced because the links can be routed efficiently.

The point-to-point connectivity between nodes simplifies routing and reduces layout area; this property is also beneficial for lower energy consumption in the interconnection network. Because each link only connects two nodes, the capacitive load for each link is small, which can potentially reduce the power requirement or allow the interconnect network to operate at a

higher frequency.

The hierarchical ring architecture is composed of rings connected together by a central ring. Each ring can be clocked differently so IP cores can run at the frequency optimized for them without any modification. Not only can the development time be reduced by efficient component reuse, but different clock frequencies in the system can reduce energy consumption because some components are allowed to run at a lower frequency.

The ring architecture is a packet-switched network that provides properties beneficial to SoC designs. The architecture uses packets as the basic communication units to allow reliable data transmission, components within the network can communicate concurrently, and a standard network interface is used to allow faster integration.

2.5 Related Work

There are various examples of research on multiprocessor implementation with an NoC design methodology [23, 16, 24]. STMicroelectronics has developed a multiprocessor system targeted for network processing applications [23], but the primary focus has been placed on processor utilization. Characteristics of the interconnect network are modeled with parametrical latency, which may not reveal the true latency of the network introduced by various applications. The butterfly fat-tree interconnect architecture proposed by the SoC Research Lab at the University of British Columbia [16] is a hierarchical network consisting of tree networks connected with a mesh at higher level. The network was designed to resolve global synchronization issues caused by long global wire delays. Issues such as multicast and communication performance with respect to silicon area are also addressed. The MicroNetwork proposed by Sonics Inc. examines the latency and area advantage of a bus network and made improvements suitable for SoC integration. An SoC design flow and a simulation platform are also proposed in the paper [24]. A study on packetization and routing analysis of an on-chip multiprocessor network [13] contains a comprehensive analysis of NoC implementation, congestion handling

with a dynamic routing algorithm, and energy/performance analysis for a shared memory multiprocessor system implemented with a mesh interconnect network.

In this thesis, a design flow for a multiprocessor SoC design with a two-level hierarchical ring interconnect network is described. The system design includes considerations for both the interconnect network and the processor IP cores. System-level issues such as flow-control and deadlock resolutions are also considered. A simulation platform is also implemented at a sufficient level of detail to illustrate functional correctness of the design. The platform is also used to optimize some of the system design parameters.

Chapter 3

Architecture

The architecture adopted in this thesis is based on NUMAchine multiprocessor. The proposed architecture is targeted for on-chip implementation, and the architecture differs from the NUMAchine multiprocessor to optimize for on-chip implementation.

3.1 Two Level Hierarchical Rings

The hierarchical ring architecture used in the NUMAchine multiprocessor consists of four local rings connected by one central ring. Each local ring is composed of four stations, giving a total of 16 stations in the system. Each station is made up of 4 generic processors, memory, and control logic. The stations are connected to the local rings through station-ring interfaces, while the rings are connected to the central ring through inter-ring interfaces. Figure 3.1 illustrates the interconnect interfaces of the architecture.

The hierarchical ring network uses packets to communicate between the stations. Unidirectional rings are used such that there is only one unique path between each station. The unique path preserves the ordering of packets on the receiving end of the network. The architecture uses slotted ring approach for packet transmission. Each ring connection point (i.e. station-ring interface or inter-ring interface) is a slot. A set of registers is used to hold a packet at the transmitting end of a link during a clock cycle. At the beginning of each clock cycle, the

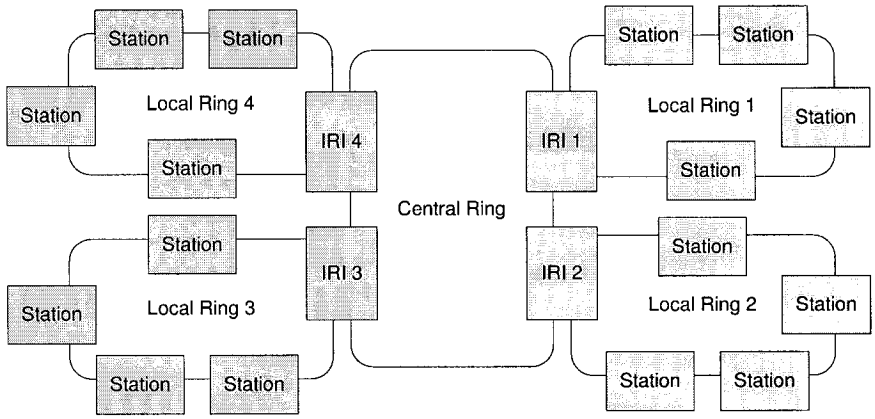

Figure 3.1: Two-level hierarchical ring interconnect

content of the register is replaced with the next packet to be transmitted.

Addressing of the stations is achieved through use of bit-masks. In the two-level hierarchical ring network, the addressing mode consists of two sets of masks: local-ring mask bits and central-ring mask bits. Each local ring is assigned a bit in the central-ring mask, and each station is assigned a bit in the local-ring mask. The mask bits have size equal to the number of nodes in the rings, in the implementation, 4 local rings are matched by 4 bits in the central-ring mask and 4 stations in each local ring are matched by 4 bits in the local-ring masks. The use of bit-masks can uniquely identify each station in the network by specifying the location of the local ring and the position of station in the local ring. Multicast of packets to multiple stations can also be applied with the bit masks with minimal addition of routing logic. Figure 3.2 illustrates the addressing bit-mask for the stations. Multicast packets differ from the regular packets in the mask bits. Instead of using one bit to identify the location of the station on a local ring, multiple bits can be used to identify multiple stations. Similarly, multicast to multiple local-rings can be performed by setting multiple bits in the central-ring mask bits. In the multicast case, one packet can be distributed to the destination without the need to transmit multiple copies at the transmitting station; such approach can significantly reduce the bandwidth requirement.

Congestion of traffic is handled in the architecture with lossless data transmission. The hierarchical ring architecture prevents packet loss due to FIFO overflow by implementation of

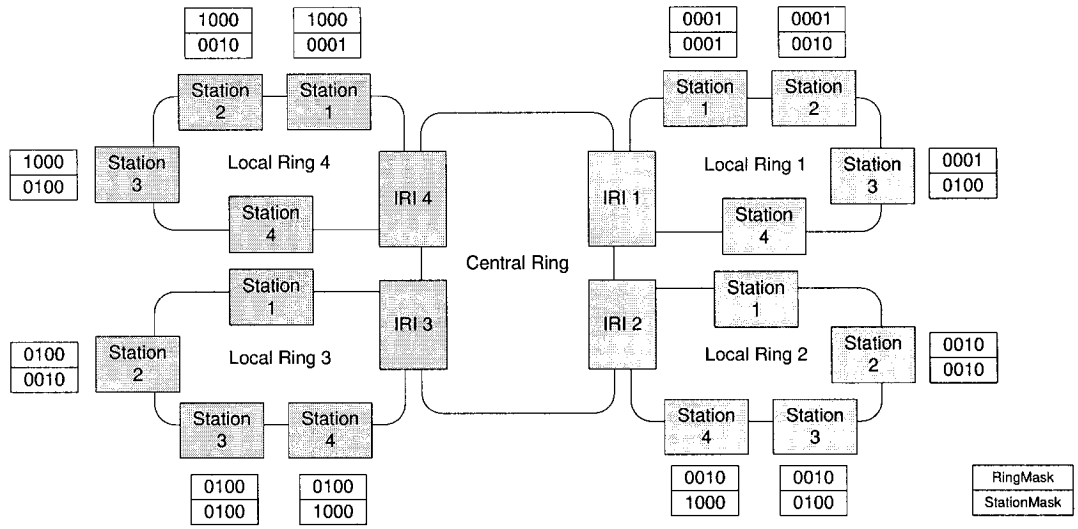

Figure 3.2: Station addressing with bit-masks

backpressure mechanism which inform the potential senders when the FIFO is approaching its capacity.

Based on the description of the hierarchical ring architecture from NUMAchine multiprocessor, it was determined to be suited for on-chip implementation for the reasons below which are described in more detail in the following sections:

- Data integrity,
- Layout simplicity,
- Natural grouping of components for locality,
- Potential for lower energy consumption

3.2 Data Integrity

Data integrity is good for network type of interconnect that uses packets as the basic communication units. Error correction algorithms have been applied on modern computer networks and have been observed to be quite reliable to ensure data integrity. The unique paths between the

stations which ensure proper ordering of the packets arriving at the receiving end also relieve the designer from having to implement complex algorithms to reassemble packet data.

3.3 Natural Grouping of Components

As clock-skew becomes more pronounced, synchronizing every component on a chip to a global signal will become impossible [9, 25]. The solution is to partition the system into multiple clock domains resulting in locally synchronous clock domains which are globally asynchronous with respect to each other. The two-level hierarchical ring topology described in Section 3.1 is well suited for such partitioning; each ring can be controlled by a local clock signal. Synchronization between clock domains is achieved through the use of asynchronous buffers between rings.

In a study by L. Benini and G. De Micheli [9], properties of hierarchical networks are explored. The introduction of hierarchies in the design of interconnect network allows grouping of tightly coupled computational units so that local traffic is maximized while minimizing global traffic. The overall effect is to reduce the total bandwidth required for the interconnection network [9, 25]. The grouping of components with different clock frequencies can also reduce the development effort and time. Since some of the IP cores need to run at their native frequency, local clock frequency can further reduce the effort for reuse of previously designed components.

3.4 Power Considerations

The hierarchical clocking scheme is also beneficial for low-power devices which are common in SoC implementations. Energy consumed by the interconnect network is directly attributed to the operating frequency. Optimization can be performed by operating rings where components have less communication activities at lower clock frequency. Real-time power optimization can also be made possible by clocking non-utilized portion of the interconnect network at a

slower rate.

3.5 Layout Considerations

The primary goal of system-on-a-chip implementation is to pack all components of the system into a chip. SoC implementations have larger silicon area because more components need to be placed on the IC, as well as larger number of connections required to connect the components. SoC can potentially be expensive to fabricate because production yield decreases as the IC silicon area increases. The hierarchical ring architecture is suited for SOC implementation as interconnections do not have complex routing, and it optimizes both performance and layout area.

An important consideration for chip fabrication is to minimize the area of the chip. ICs with large silicon area are prone to fabrication defects and do not utilize the area on wafers due to their larger geometry. Both fabrication defect and wafer area utilization contribute to lower yield and increase cost per IC.

Wire routing in VLSI layout can have a significant effect on performance and chip area of the system, efforts have been made in researching efficient routing algorithms. Inefficient routing can increase the wire length unnecessarily which not only increase the chip size, but also decrease performance. Efficient routing is especially important for designs in SoC, although advancement in chip fabrication allow designer to scale down the size of transistors for smaller logic implementation, but interconnection wires cannot be scaled at the same level as the transistors [1].

The hierarchical ring architecture described in the work has a planar structure such that the interconnections are local to the rings, and the interconnections do not cross each other. Layout is simplified as interconnects are mostly local and can be routed systematically. Figure 3.3 below is an example of layout of the hierarchical rings.

Figure 3.3 illustrated the simplicity of layout for the hierarchical ring architecture. Each

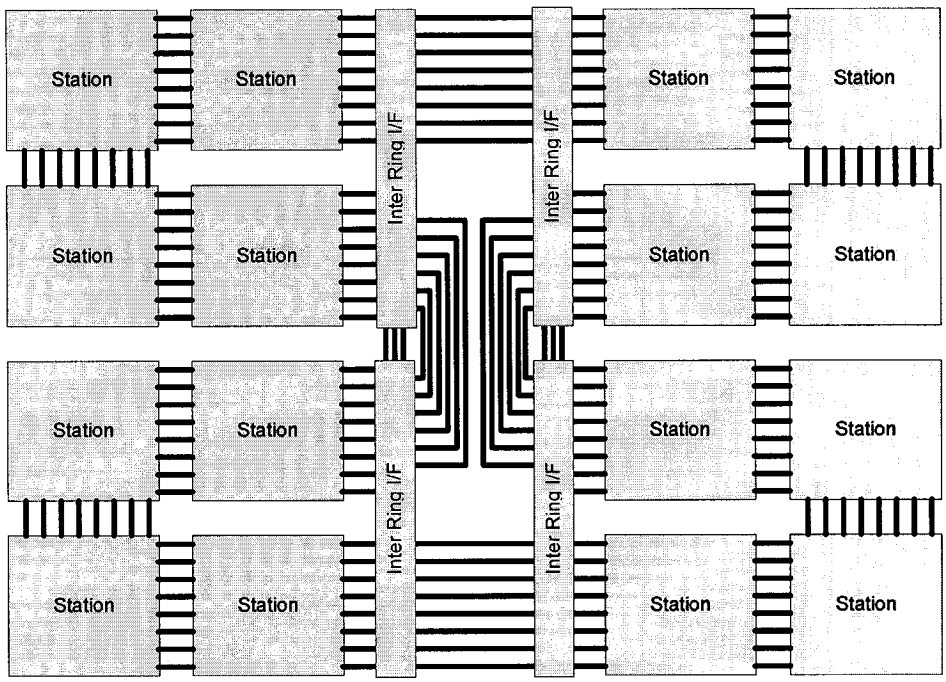

Figure 3.3: Hierarchical rings and layout

station is laid out and connected with others to form a local ring. The local ring can be placed as a macro cell, and local rings can be instantiated and connected together to form the central ring. The complexity of layout is simplified as all connections in the interconnect network are point-to-point connection to their neighbor.

3.6 Migration to On-chip Implementation

Although previous discussions have indicated that the hierarchical ring architecture from NUMAchine is suited for on-chip implementation, further optimization can be made to migrate the architecture to SoC. Moving from shared memory model to message passing model can reduce the complexity and size of the interconnect implementation. As a result of using message passing model instead of shared memory, the FIFO size on the inter-ring interfaces and station-ring interfaces can be significantly reduced. The change in communication method also requires modifications to handle deadlock differently.

3.6.1 Distributed Memory Model

The NUMAchine multiprocessor took the shared memory approach in order to ease the burden of migrating single processor applications to the platform. Local caches are used to reduce bandwidth requirement. These caches introduce data coherence issues where copies of block of data can exist at multiple locations, and modification to the data at any one of the copies must be reflected to copies at all locations. NUMAchine resolves the data coherence issue by introducing a directory cache coherence protocol where the directories are maintained at both the memory level and the network level.

The proposed system architecture uses distributed memory model for parallel processing. In the proposed architecture, each individual processor has an independent address space and its own dedicated physical memory. Communication between processors relies on minimal hardware support for message-passing send/receive operations that are explicitly initiated by the embedded software running on each processor.

Distributed memory model implementation provides better performance for the following reasons: embedded software has matched send and receive pair such that a sender only sends data that are no more than required by the receiver. In the shared memory with local cache approach, a cache miss would require a full cache block of data to be transmitted. A cache coherence protocol also adds to the bandwidth usage on the network to resolve cache conflicts. Since the embedded software designer needs to have a more detailed understanding of the architecture, parallel programs that use message passing approach can be implemented more efficiently. Lastly, it was discussed in Section 2.3 that the shared memory type of applications can be implemented in software or OS level if desired. The design approach taken for the proposed system is to have simple and effective framework which eases the effort necessary for system integration and offloads the hardware complexity to software level.

3.6.2 Congestion Handling

As described previously, the on-chip network does not exhibit the non-determinism of packet losses due to disconnection of nodes. The retransmission protocol approach for flow-control then becomes less attractive for the on-chip implementation since the only source of packet drop in such approach is due to congestion. Retransmission of packets only adds more traffic to the network which may cause further packets to be dropped. The order of packet arrival can also be disrupted because some packets are dropped and retransmitted. In addition to the issue of reduced performance due to retransmission and packets not arriving in-order, complex logic for the retransmission protocol must be implemented at the station interfaces, and large buffers are required for the protocol implementation. The approach taken in the NUMAchine multiprocessor and the work described here implements lossless communications which uses a simple backpressure mechanism for handling network congestion while avoid packets being dropped.

Backpressure mechanisms in a two-level hierarchical ring structure are implemented at each level. A backpressure signal is required to control traffic between the local rings, in the local ring level, backpressure mechanisms are needed to control the rate of traffic into and out from each local ring. The implementation differs from the conventional per-hop backpressure, a backpressure signal is shared between the interfaces in each ring. All interfaces in the ring can take appropriate measure as soon as the status of backpressure is changed, in per-hop backpressure approach, each interface must wait for the backpressure to propagate through the network. At each hierarchical level, backpressure is triggered when one of the FIFO in the ring is reaching capacity. The mechanism is necessary to prevent new packet from being placed on the ring that can potential overflow the FIFO and cause packet loss, it is disabled when the packets in the offending FIFO is drained to a specified level.

3.6.3 Flow Control

The flow control mechanism implemented in the work is based on NUMAchine [4], and is extended for distributed memory system architecture. Three levels of flow-control are implemented: the Central-Ring level, Local-Ring level and Station level flow-control.

At the Central-Ring level, the ringStop backpressure is activated when the central ring is congested and has risk of packet loss if more packets are moved from local-rings to the central ring. The condition for such situation occurs when one of the downward FIFO at the inter-ring interfaces is reaching capacity. Any new packets placed on the central ring have risk of overflowing the FIFO. The solution is to stop packets from ascending from local rings on to central ring by signaling all inter-ring interfaces with the ringStop backpressure. At the time when the ringStop backpressure is triggered, packets already on the central ring is allowed to continue to travel among the inter-ring interfaces until they are removed (i.e. descended to local rings). The ringStop signal is de-asserted when the FIFO level drops to one forth of the total capacity.

The Local-Ring level flow control is triggered when a local ring cannot send more packets to central rings due to congestion in the central ring, or when the stations cannot receive more packets because the stations cannot process the received packet fast enough. A stopUp signal is triggered when the up-ward FIFO in the inter-ring interface reaches a specified level, a stopDown signal is asserted when receiving FIFOs on one of the ring interfaces has reached the limit. The stations are prohibited to send new packets into the local ring when either stopUp or stopDown signal is asserted. Both stopUp and stopDown signals are de-asserted when the respective FIFO level drops to one forth of the total capacity.

The third level of flow control is triggered when the out-going buffer at the ring-interface is full due to congestions in the rings. When this happens, any new data transmission from the station will be blocked until there is room in the buffer.

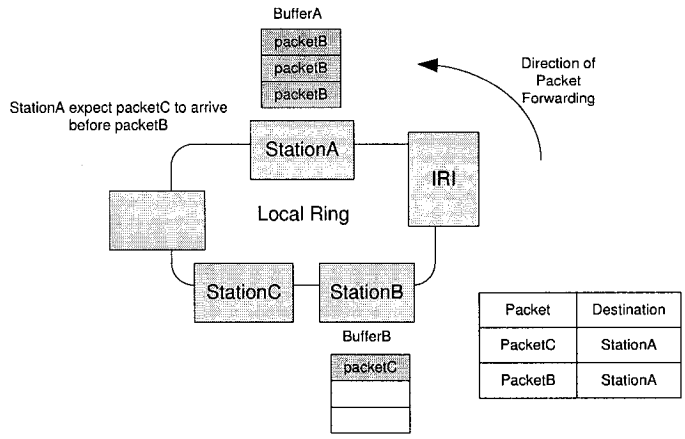

Figure 3.4: General store and forward deadlock at network level

3.7 Deadlock Handling

Section 2.2.4 described a list of potential deadlocks in the network environment. The following discussion will illustrate the scenario of the deadlocks in the hierarchical ring network, and the measures taken to prevent them.

3.7.1 General Store and Forward Deadlock

The following scenario illustrates an example of general store and forward deadlock: Consider in the ring network where three stations are exchanging data, the network arrangement is illustrated in Figure 3.4.

Prerequisite: Station A blocks while waiting for packets from station B and station C, and station A must process packets from station B (packet B) before packets from station C (packet C). Packet B can go directly to station A, and packet C must go through station B to get to station A.

Condition: Station A receives packet B first and filled up the receive FIFO, and packet C is being passed along to station A.

Result: Packet B will not be processed because station A has not received packet C. On the other hand, packet C can not get to station A because there are no empty slots in the receive

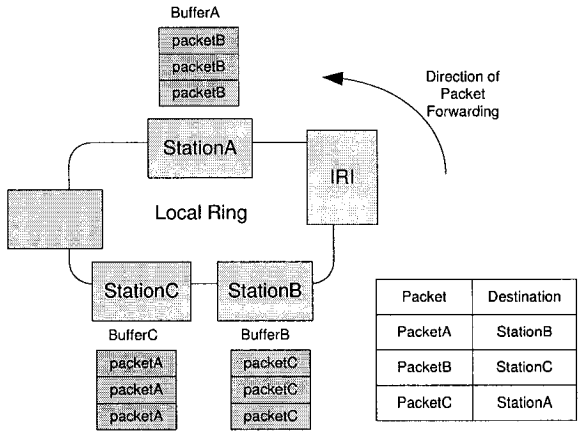

Figure 3.5: Indirect store and forward deadlock at network level

FIFO on station A. The circular dependency then leads to deadlock.

NUMAchine resolves the issue with the shared memory architecture where the data is pulled from the remote station as required. In NUMAchine, the scenario described will be different: packet B and packet C will be requested when station A tries to access the data and experience a cache miss. In such a case, station A will request packet C first then request packet B when it has finished processing packet C and sees another cache miss for packet B.

The proposed architecture confront the potential store and forward deadlock by providing a non-blocking path which stores the data content directly into the desired memory location on the stations as soon as the packets arrive. The approach is to use direct memory access (DMA) mechanism, which handles incoming packets by removing them from the receive FIFOs and place the packet data into appropriate memory locations regardless of the status of process on the processor. With the DMA in the system, arrived packets are moved from the FIFO into memory in the station even when the process is blocked. It then becomes clear that blockage of packets imposed by the limited FIFO size on the station-ring interface can no longer exist, so that new packets can always reach their destinations.

3.7.2 Indirect Store and Forward Deadlock

Figure 3.5 illustrates a possible scenario for indirect store and forward deadlock. The scenario is described as follows:

Prerequisite: station A blocks while waiting for packet C, station B blocks while waiting for packet A, and station C blocks while waiting for packet B.

Condition: All stations have initiated transmission of packets and that all receive FIFOs at the stations are filled with packets from its direct neighboring station.

Result: Packet from station B cannot be moved to station A because buffers in station A are full of packets that need to be transmitted to station C; the packets from station A may not be able to be moved to station C for the same reason, similarly, packets from station C cannot be moved to station B because the packets from station B are still waiting to be moved to station A. The loop of dependencies then causes indirect store and forward deadlock.

The potential of the described deadlock is identified to be caused by single buffer type and is described in Section 2.2.5. The suggested solution is to use two independent FIFOs for staging send and receive packets. The approach was adopted in NUMAchine multiprocessor and was kept in the proposed architecture.

3.7.3 Acknowledgement Packet Related Deadlocks

Section 2.2.5 has identified potential deadlock caused by blockage of acknowledgement packets. The situation is sometimes referred to fetch deadlock [20] where a packet in the FIFO which cannot be processed, and would cause total blockage of all the remaining packets in the FIFO. The issue is resolved in NUMAchine by categorizing packets into sinkable which do not require returning packets (such as write access packets) and non-sinkable which expect the receiving station to send packets back to the sender (such as acknowledgement and read data) packets, the two categories of packets are handled differently such that sinkable packets cannot be blocked by non-sinkable packets while the station is generating the responses to the non-sinkable packets. The acknowledgment packet related issues do not have much effect in

message passing multiprocessors such as the proposed architecture. Messages are explicitly initiated by the sender rather than having request-and-acknowledgement type of communication that must be processed differently while they are in the buffers.

3.7.4 Network-level Deadlock Prevention Measures

The rules followed in the NUMAchine multiprocessor project to prevent deadlocks were discussed in Loveless's thesis [4] are:

- A non-blockable downward path in the hierarchy must always exist for sinkable transaction, even if the upward path is stalled indefinitely.
- The number of outstanding non-sinkable requests that can exist in the ring hierarchy at any given time is bounded, and the value of the upper bound can be determined.

The first rule is to ensure that stations can continue to receive while sends are blocked due to congestion, which guarantees that the network can make forward progress. When no new packets can appear on the network while the ones exist in the network can continue to be removed, the source of congestion will eventually be resolved. The second rule is related to the size of buffer used to hold the non-sinkable packets, and is not relevant to the proposed architecture.

In hierarchical rings, the non-blockable downward path is made possible by a special switching scheme used in the backpressure mechanism. While the backpressure flow control mechanism can prevent stations from adding more packets on the rings, packets already exist on the network (not including the ones on FIFO) are still allowed to move on the rings until they have reached their destinations. The assertion of backpressure signals does not immediately stop network traffic, and must handle the packets already on the network.

Implementation of the non-blockable downward path in the two-level hierarchical ring network is described as follows: In the central-ring level, when ringStop signal is asserted by one of the inter-ring interface, no new local-ring (located on the up FIFO) packets can be moved

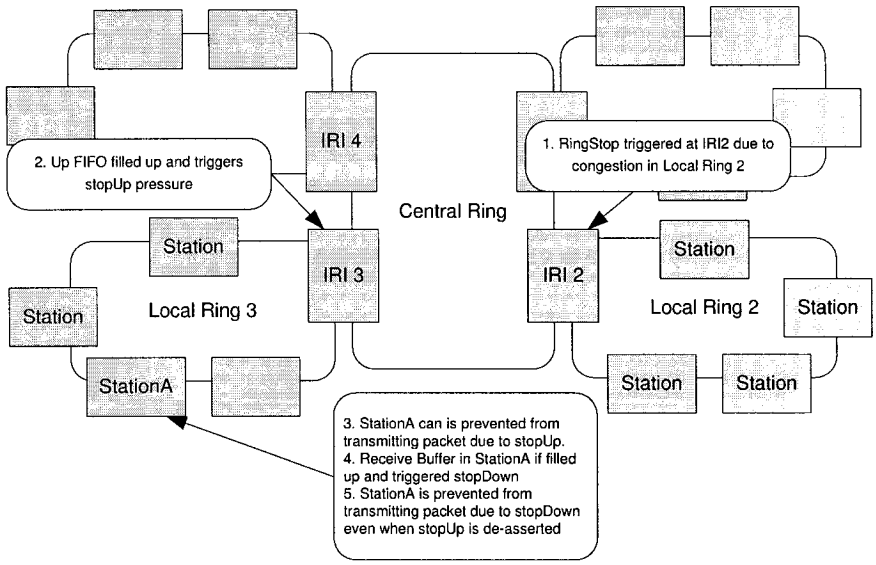

Figure 3.6: Software caused deadlock propagated to network level

to the central ring, but packets can descend from central ring to local rings. In the local-ring level, stopUp signal which indicates that the central ring cannot take more packets prevents the stations to place new packets on the local ring. However, packets already on the ring may still move along the local-ring until they are removed at their destination. It is then apparent that while congestion occurs in the upward traffic, downward traffic can continue to move and drained at their destination which makes forward progress possible.

3.7.5 Deadlocks Propagated From Software Layer

While the potential for deadlock has been considered at the network interconnect level, deadlock can still occur in the software level and propagate down to the network level. Figure 3.6 illustrates one possible scenario of such deadlock.

The scenario first starts with a ringStop in the central ring due to congestion at another local ring. The backpressure is propagated to the local ring and triggers assertion of the stopUp signals which prevents new packets from appearing in the local ring. When station A tries to send data, the transmission will be blocked waiting for the stopUp signal to clear. While

the station waits for the packet transmission, packets from other stations are allowed to reach station A as part of the non-blocking downward path implementation. When the transmission in station A is in progress, the station is unable to process the arrived data, then the input FIFO at station A starts to accumulate, and eventually causes stopDown backpressure. The stopDown backpressure then prohibits station A from transmitting, even with absence of the stopUp backpressure. Since station A is still trying to transmit and unable to transmit due to stopDown backpressure, and station A is still unable to process the arrived packets, the stopDown signal will not be cleared, a circular dependency can then be observed which causes deadlock in the local ring.

The source of the deadlock starts out from the fact that the embedded software on the station is unable to process received data while data is being transmitted. The blockage of the station from processing the received data in turn causes congestion on the network and eventually can cause deadlock. Typically, the limitation of a station not being able to process received data while transmission is in progress is imposed by the software API. The API can be modified to interleave processing of data transmission and processing of received data. However, such an implementation is inefficient and causes confusion to the software designer, especially when the type of deadlock can only occur in rare cases.

The measure taken to resolve such a deadlock issue is to use a DMA device which allows removal of received data in the receive FIFO while transmission is in progress. When transmission and reception of data can occur simultaneously, the circular dependency is effectively broken, which eliminates the potential of such deadlock.

Chapter 4

Simulation Model

This chapter discuses a software model of the proposed multiprocessor system for verifying the functionality of the design as well as observing the behavior of the system. The model is also used for design space exploration of the system such that the optimal design parameters can be determined. In this chapter, modeling environment, as well as the implementation details for the model of the proposed hierarchical rings are described.

4.1 Modeling Languages and Libraries

The behavior of a system can be modeled in software. Different aspects of the system can be evaluated in software before the actual hardware implementation. For the model to be efficiently used in a design project, the model must be able to be developed quickly, and should properly reflect the behavior of the system.

4.1.1 SystemC

C and C++ are among the most popular programming languages used today. Because designers are familiar with these languages, it makes sense to use them to build executable specifications. C and C++, however, are generic programming languages and lack the constructs that are

necessary to describe hardware behaviors.

SystemC is a C++ library that provides modeling constructs similar to those used for RTL and behavioral modeling within VHDL or Verilog [6]. Highlights of SystemC are described in the SystemC User Guide [3] and are summarized below.

- SystemC supports the description of hardware, software, and interfaces in a C++ environment,
- The SystemC syntax for modeling hardware is similar to that typically found in hardware description languages (HDLs). Constructs such as modules, processes, and ports are supported for describing hardware behavior.
- Multiple clock signals of arbitrary phase relation that are common in an SoC environment can be modeled easily in SystemC.
- SystemC supports cycle-based simulation for fast and effective simulations.
- Multiple abstraction levels are supported in SystemC, where the levels of abstraction range from high-level functional models to detailed cycle-accurate RTL models.
- Lastly, the SystemC library supports signal-trace capabilities which are essential for verification purposes. SystemC models can generate waveform output in various industry standard formats such as VCD, WIF and ISDB.

SystemC provides support for an HDL-like design methodology while allowing the design to be implemented at various level of abstraction. This flexibility allows for fast validation and optimization of the design, as well as the ability to perform exploration of various algorithms [3]. The SystemC library is freely available to the general public; it works with most of the C++ development environment and tools. Because the SystemC libraries are implemented in C++, standard C++ development tools can be used at low cost as compared to the cost of standard HDL development tools. There are no requirements for expensive licenses and the development environment can be installed on relatively inexpensive computers.

The SystemC library provides support functionality commonly found in HDL development tools as well as syntax similar to HDL languages. The SystemC core consists of an event-driven simulator that works with events and processes [26]. Modules and ports are used to represent structure, while interfaces and channels are used to describe communication [26]. Modules provide the ability to partition a system into functional blocks and are composed of processes, ports, internal data, channels, and other modules [26]. Processes belonging to different modules communicate through channels and ports. Events are triggered in the same fashion as other HDLs such as Verilog and VHDL.

System models can be built in different abstraction levels which range from high-level behavior models to HDL-like cycle-accurate models. For fast development and evaluation of a new component, high-level behavioral modeling can be used such that the functionality can be verified. A component modeled at a behavioral level does not need to reflect the hardware implementation and must generate the proper response with given input. The interior of the components can be designed in pure C++ syntax. For code reuse, the component can be encapsulated into a module and connected to other components through ports, interfaces, or channels. In SystemC, the module can be treated as a C++ class, while instantiation of a module is equivalent to instantiating an object of the class. For more detailed simulation, SystemC allows cycle-accurate design with the use of HDL-like constructs. Similar to designing in HDL, the content of a component is divided into processes that are triggered with specified signal events. The signal events can be either system clocks or signals originating from another component connected through the input port.

The goal of SystemC is to build system models that can be refined into synthesizable RTL, then synthesized into netlist, and ultimately can be used to fabricate the designed system. When the technology becomes available, the overall design flow can be greatly simplified. In the current method of designing an IC, a system-level model is designed independent of the hardware. Once the system-level model is developed, a designer manually converts the model into an HDL where the design is simulated again and synthesized for production. With the con-

ventional design methodology, inconsistencies between the software model and the hardware design become an issue. First of all, conversion from the system-level model to HDL is tedious and error prone. Secondly, once converted to HDL, when design flaws are discovered and fixed in hardware, updates to the software model are often neglected to reflect the modifications. Finally, test-benches used for system-level models are typically not compatible with the HDL model, and it is often necessary to convert the test-benches in languages such as C into the HDL environment [3].

With the proposed SystemC approach, inconsistency is no longer an issue. SystemC adopts the refinement methodology where the system is divided into modular components. Each component model is initially written at the highest level of abstraction, typically to reflect the behavior of the model. The design can then be slowly refined for each component to add more detail such as hardware and timing constraints. When all components are modeled in the cycle-accurate level of abstraction, the model then is complete, and can be converted into synthesizable RTL with relatively less effort. Designers can model in SystemC from the system-level to RTL level without having to know multiple languages. Unlike the conventional approach, SystemC testbenches can be reused for both system-level models and RTL models, which lead to considerable savings in development time.

4.1.2 StepNP

StepNP is a network exploration platform developed by STMicroelectronics that provides simulation tools and SystemC modules targeted for network processor modeling. The modules provided by StepNP includes processors, memory, and interconnect modules [27]. Supplied with the platform is a collection of cross-compiler tools for generating binaries to be executed on the processor modules. Other tools provided by the platform include SystemC macros that can be used for the introspection of signals during simulation as well as a graphical user interface that can be used to interface with the simulator for debugging and data collection purposes. A high-level model of standard interface is also supplied with the StepNP library to allow faster

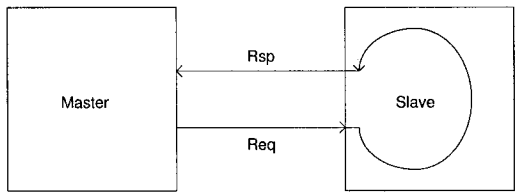

Figure 4.1: Transaction-level communication

system integration and more efficient simulation of the system model.

4.1.3 Transaction-Level Communication and SOCP

The transaction-level model is introduced in SystemC as part of a high-level modeling approach to improve simulation performance. Transaction-level channels implement the communication behavior of bus protocols without timing information. Communication through channels is modeled by events that are typically specified by the type of operation. Events in the transaction-level are categorized into requests and responses. Bit-level signals and other physical level components are not included as part of the transaction-level model.

Figure 4.1 illustrates how transaction-level communication is performed. The communication is initiated by the master device sending a request operation. The operation is interpreted by the slave device and the corresponding task is performed. When the slave device completes the task, a response is sent back to the master device [28].

The SystemC Open Core Protocol (SOCP) communication channel is the SystemC abstraction of the Open Core Protocol (OCP) standard [29, 7]. The goal of OCP / SOCP is to provide a standard configurable interface for IP cores that is capable of handling all types of communication. The standard interface specification allows components to be integrated with minimal effort. With a standard interface such as SOCP, the designer no longer needs to follow a particular input/output specification and to try to connect the components together manually through their input and output signals. SOCP offers the high-level abstraction of the OCP standard. SOCP have no bit-level signals, polarities, clock cycles, or detailed timing information. Communication is modeled at the transaction level; the requests and responses to requests are

modeled and can be considered functionally identical to the OCP. The higher-level abstraction allows better performance in simulation; more details can be added to SOCP channels in later stages of the design cycle to conform to the timing requirements specified by the OCP standard.

4.2 Components

The two-level hierarchical ring multiprocessor system described previously is modeled with SystemC and StepNP libraries. The system is composed of 16 stations communicating over a two-level hierarchical ring interconnect. Each station contains a processor with a dedicated memory component and relevant control logic. To allow expandability in the future for more processors, a shared memory is connected to the processor through a configurable access controller the can supports up to four processors. Transmitter and receiver blocks are used to handle communication between the processor and the interconnect module.

Following the SoC design methodology proposed in a study by Cesario et al. [30], the implementation of the multiple-processor system is divided into two distinct categories: the interconnect module and the station module. The functionalities of each module are defined so that the modules can be developed independently of the other. The interface between the modules adopts the SOCP communication channel interface. Communication specification for operations between the interconnect network and station modules is also defined with a list of expected requests and responses to the requests so that there is no ambiguity in integrating the components.

The interconnect module forms the physical layer of the communication protocol. The module itself can be viewed as an independent IP core and the primary function of the interconnect module is to relay data from one of its endpoints to another. Given that the interconnect network and station modules are implemented with a standard interface, the interconnect module can be replaced easily with a different architecture. Similarly, the station modules can also be replaced with other types of modules that are not restricted to multiprocessor implementa-

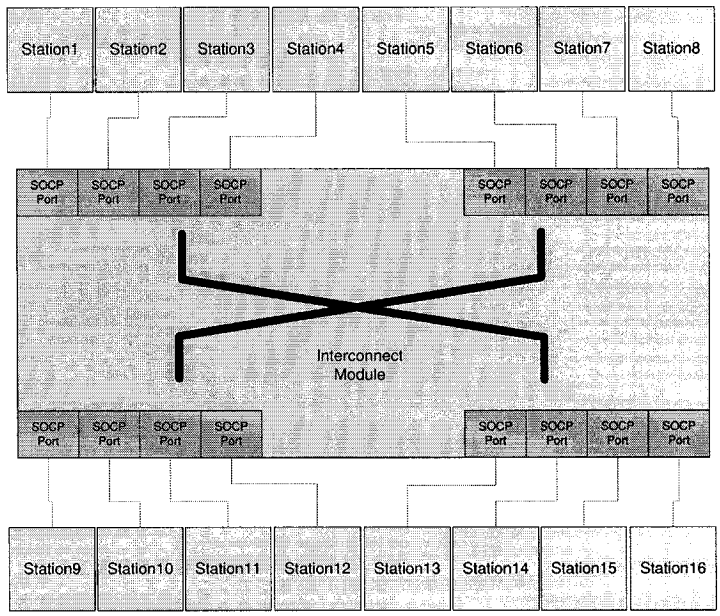

Figure 4.2: Simplified component integration with SOCP

tion. In the proposed two-level hieratical ring multiprocessor system, the stations are instances of the same station module. However, it is not required that all stations are made equal; stations of different implementation can be used. For example, some stations may be responsible for I/O processing, while another maybe responsible for computation such as execution of algorithms on the input data.

The division of components into different IP cores reduces the complexity of system integration. This ability is especially useful in simulation models, as the design space can be easily explored to determine the feasibility of an experimental architecture in terms of performance or cost, or trade-off of different algorithms.

Figure 4.2 illustrates the connectivity of the interconnect module and the station modules. With SOCP, the interconnect network can be replaced with different architecture implementations. Stations can also be individually replaced with modules with different implementations.

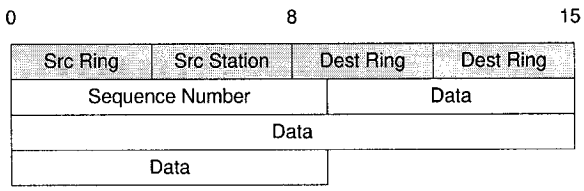

Figure 4.3: Packet structure

4.2.1 Interconnect

In this section, the implementation of the interconnect module is described. The module implements the hierarchical ring architecture described in Section 3.1. The multiprocessor system consists of four local rings with four stations for a total of 16 stations. Each local ring is connected to a central ring with an inter-ring interface that contains two packet FIFOs, one for each direction to or from the central ring. Each station is connected to a local ring with a station-ring interface that has a FIFO to store packets received from the local ring, and a transmit FIFO for staging packets that will be sent to the local ring.

The links connecting the inter-ring interfaces or station-ring interfaces are bit-parallel signals which represent one packet of data, i.e., each bit represents one bit of the packet content. The packet has a size of 57 bits and the format is illustrated in Figure 4.3. The SrcRing and SrcStation fields form the bit mask used to identify the station that created the packet. The DestRing and DestStation fields form the bit mask used to identify the intended destination of the packet. The two fields of a packet can be modified by the switches. The bits after sequence number until the end of the packet are used for storing data that the packet encapsulates. The Data field of the packet can be expanded to contain more data in the future if higher bandwidth is desired.

4.2.2 Inter-ring Interface

Figure 4.4 illustrates the implementation of the inter-ring interface block. The inter-ring interface is connected as part of the central ring and connects to the two ends of a local ring. The functions of the inter-ring interface are outlined below.

- The inter-ring interface switches packets from local rings (destined to stations on another ring) to the central ring.
- The inter-ring interface also is responsible for forwarding packets on the central ring that are destined to another local ring.
- Packets targeted for the attached local ring are switched at the inter-ring interface from the central ring to the local ring.
- In case of switching multi-cast packets, one copy of the packet is forwarded to the next inter-ring interface and a copy is forwarded to the local ring. Both copies of the packet have their ring-mask adjusted accordingly.
- Inter-ring level flow-control is achieved through implementation of FIFO monitoring logic integrated with the switching algorithm. The details were described in section 3.6.3.

There are two inputs to the inter-ring interface, *localInput* takes input from the local ring attached to it, where *centralInput* takes input from the previous inter-ring interface connecting to it. There are also two outputs from the inter-ring interface, *centralOutput* and *localOutput*, connecting to the next inter-ring interface and the local ring respectively.

When switching packets, priority is given to packets on the same hierarchy level. Packets switching from one level to another have lower priority. The packet on centralInput will have priority over packet at localInput when there are packets arriving on both inputs, both expected to be switched to centralOutput. In this case, packet at localInput is buffered in the Up FIFO (also called localCentral FIFO) until there are no more packets on the centralInput. Similarly, when there are packets arriving on both localInput and central input both expected to be switched to localOutput, the packet at localInput has priority over the packet at the centralInput, and packets in from the central ring are stored in the Down FIFO (also called the centralLocal FIFO) while there is traffic from the local ring passing through the inter-ring interface. The switch controller block in the figure is a generalization of the control logic within the inter-ring interface module.

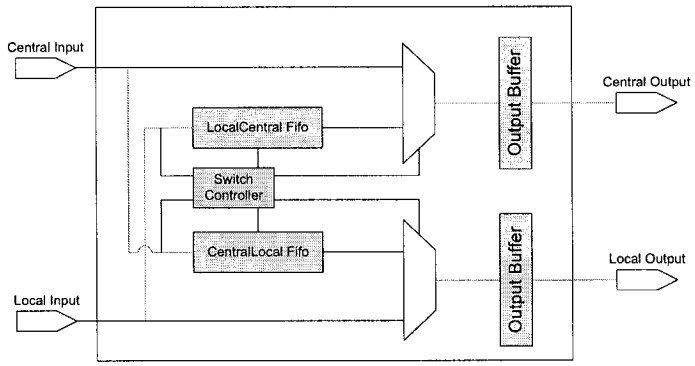

Figure 4.4: Inter-Ring interface implementation

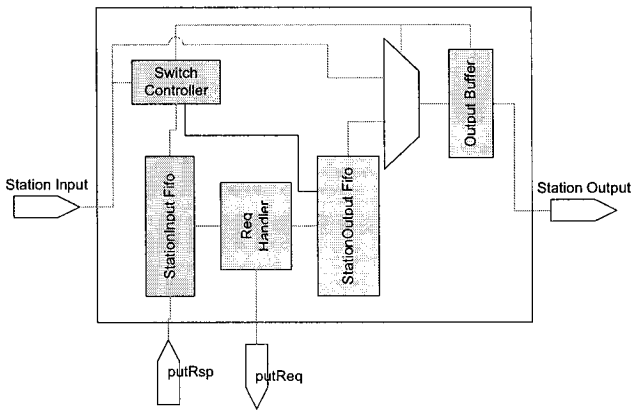

Figure 4.5: Station-Ring interface implementation

4.2.3 Station-ring Interface

The station-ring interface is the bridge between the interconnect module and the station modules. Figure 4.5 illustrates the implementation of the Station-Ring interface block. The station-ring interface connects to its neighboring interface through the stationInput and stationOutput ports. Connection to the station module is achieved through the masterPort and slavePort with SOCP interface. The functions of the station-ring interface are listed below:

- Packaged data is transmitted from the station into a packet and stored in the stationOut FIFO.
- Packets that are not destined to the station are forwarded to the next interface in the local ring.

- At each clock cycle, the interface transmits data from the stationOut FIFO to the next interface in the ring when the interface is not busy forwarding packets.

- Packets that are destined for the attached station are removed from the network at the station-ring interfaces. Received packets are stored in the stationIn FIFO.

- In case of switching multi-cast packets, one copy of the packet is forwarded to the next station-ring interface and another copy is stored in the stationIn FIFO. The station mask of the current station is removed from the forwarded packet.

- Mechanisms for the station to query transmission status are also provided for the component connected to it through a SOCP interface.

- Ring-level and station-level flow control is achieved through implementation of FIFO monitoring logic integrated with the switching algorithm. Details are described in section 3.6.3.

The switching algorithm is similar to the inter-ring interface. Priority is given to existing traffic on the ring. When there are packets on both the station input and the stationOut FIFO, the packet on station input is transmitted first. Packets on stationOut FIFO can only be transmitted when there are no packets on the stationIn port.

The SOCP interface connects the station-ring interface and the station module. A putReq remote function is implemented on the station-ring interface to service requests from the station module.

The putReq function implemented on the station-ring interface allows the attached station module to query the status of the stationIn FIFO and the transfer of packet data. The main features implemented on the putReq function are listed below,

- At every clock cycle, the station-ring interface inquires if there are data available in the stationIn FIFO in the interconnect module.

- When there are data available from the interconnect module, the station-ring interface acquires a packet worth of data from the interconnect module and place in the specified memory location.

- The station-ring interface also implements a protocol to inquire the overhead information (packet header) of the first packet in the stationIn FIFO.

When an operation is requested by the station module, the putReq function first decodes the operation code in SOCP request, and then performs the corresponding operation. The types of operations are divided into read and write requests and are described in Table 4.1.

Operation	Op Code	Description
Read	0	query the stationIn FIFO status
	1	read input overhead (mask bit representation)
	2	read input overhead (station number representation)
	3	read data field
	4	read output FIFO status
Write	1	Update target station mask bits
	other	transmit data to the target station

Table 4.1: Operation descriptions

A read operation with operation code 0 is used to query stationIn FIFO status. If the FIFO is not empty, 1 is placed on the data field. SOCP returns data back to the station module together with a data-valid flag. If the FIFO is empty, 0 is returned with a data-invalid flag to indicate that the FIFO is empty.

A read operation with operation code 1 and 2 are used to fetch the packet overhead information. The overhead bits from the first packet on the receive FIFO is placed on the data field of the SOCP communication channel and transmitted back to the station as a response.

A read operation with operation code 3 is used to fetch the data field of the first packet in the stationIn FIFO. When requested, the first packet on the stationIn FIFO is copied to the packet FIFO and the packet entry is popped from the stationIn FIFO.

A read operation with operation code 4 is used to query the stationOut FIFO status. The number of free slots in the stationOut FIFO is returned to the station module. When stationOut

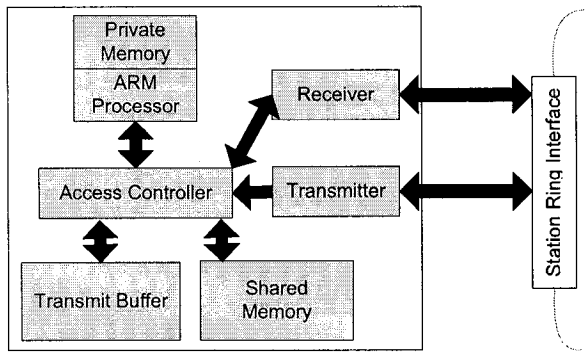

Figure 4.6: Station content

FIFO is full, 0 is returned to the station module together with an invalid code in the SOCP response.

A write operation with operation code 1 is used to update the value of the target ring-mask and station-mask. The data field of the SOCP request specifies the mask bits of the target station. When the request is processed, the mask bits are copied into the targetMask register that will be used in creating the packet header. The register maintains the last updated value of the target station mask bits until the next write request with operation code 1.

Write operations with operation codes other than 1 are processed as data transmission requests. The data field of the SOCP request contains the data that will be sent to the station specified in the targetMask register. When the request is received, the data included in the SOCP request is placed on the data field of the packet, the targetMask register content is placed in the target ringMask and target stationMask field and the sending station mask bits is stored in the source ringMask and source stationMask field. A sequence number is incremented and assigned to the seqNum field. Once prepared, the packet is placed in the StationOut FIFO. If there is insufficient space in the stationOut FIFO, the packet is dropped. The simulator will output a warning message, but it is Station module's responsibility to check the stationOut FIFO status before performing a data write request.

4.2.4 Station

Each station model consists of an ARM processor with its own private processor memory, a shared memory block to support multiple processors in each station, a transmit buffer for staging outbound packets, and transmitter/receiver units for packet transmission to/from the interconnect network.

The contents of each station in the system model are illustrated in Figure 4.6. The ARM processor module simulates the execution of code and accesses of data in the attached private memory module. If additional ARM processors are added to the station, they each have their own private memory, but they may access common information stored in the separate shared memory in the station through the access controller.

Communication to access the received data is performed through the SOCP interface on the station receiver module. Embedded software must configure the receiver to process arrived packet from the station-ring interface. When a processor wishes to send a message to another station (whether on the same local ring or a different local ring), the message is staged in the transmit buffer or in the shared memory. When the processor triggers the transmission, the transmitter fetches the message into the transmit buffer if it is not already staged in the buffer. In each clock cycle, a word is fetched from the transmit buffer and sent to the station-ring interface.

4.2.5 Direct Memory Access Usage

Direct memory access is a mechanism for off-loading data transfer functionality from the processor and having the device controller transfer data directly to or from memory without involving the processor [20]. The effect of DMA in the multiprocessor implementation described in this thesis not only improves performance, but DMA can also be used to prevent the occurrence of certain deadlock-related issue.

The transmitter and receiver modules are implemented using a modified form of DMA. The implementation of DMA described here differs from the conventional DMA implementations

for parallel computers. There are two differences. First, the DMA component in the transmitter adds another layer of protection for data corruption. While DMA data transfer is in progress, no new DMA transfer request will be accepted to prevent write-after-write type of data corruption. Second, received data is copied directly to the memory location specified by the embedded software. The use of a receive buffer is eliminated.

DMA allows sending data to another station by means of non-blocking writes. It is important to note, however, that non-blocking writes can introduce inconsistency in data transmission. When consecutive requests for transmission occur, the processor must make sure that data has been transferred completely before it triggers the next send operation. If the processor tries to send a second set of data before the first has completed, the first set of data will be interrupted and will not be received completely at the receiving station. In the implementation of the system model, the second request is rejected and a return code is returned to the processor. The embedded software must check the return status of the transmission request and take appropriate action.

DMA for the receiver in conventional message passing systems typically transfers the received data into a buffer where they are queued until the target process performs a matching receive. The data is copied into the address space of the receiving process only after the read request is received. The approach taken here for the described architecture eliminates the use of a receive buffer because the functionality is already provided by the receiving queue on the station-ring interface. The embedded software must specify in advance the location to which the received data should be stored. When data is fetched by the DMA mechanism, they are stored directly to the specified location.

4.2.6 Transmitter Module

The transmitter and receiver modules are the bridge between the processor on the station and the interconnect module. The functions of the transmitter module are outlined below.

- The transmitter module can query packet information from the shared memory module as

well as the out-going FIFO status on the station-ring interface. The information includes packet header information and availability of FIFO space.

- The transmitter can also configure the station-ring interface for packet transmission based on the transfer information fetched.

- Data that are to be transmitted can be copied from the shared memory module into the transmit buffer.

- The transmitter can transmit data of specified size and prevents potential data corruptions that are caused by consecutive transmit requests.

Transfer information is the information that is used to generate the packet header. In the implementation, the transfer information contains the target ring mask, target station mask and length of data to be transmitted. Registers are used to store information necessary for control and monitor of data transmission. The register names and their functionalities are listed in Table 4.2.

RegisterName	Description
SourceMask	masking bits which correspond to the station itself
TargetMask	masking bits which correspond to the target station(s)
MemAddr	memory address of the beginning of data content that is to be transmitted
TxSize	length of data (in number of 32-bit words) that will be transmitted
TxTrigger	Write access to the register initiates DMA packet transmission
TxStatus	Read access to the register returns status of transmission. busy = 0, idle = 1

Table 4.2: Transmitter control and monitoring registers

A memory range in the shared memory block is reserved for the transfer info. The transfer info is fetched first after detection of write access to the TxTrigger register. After the transfer information is fetched, the target station mask and ring mask are combined to form the 8-bit mask that is described in Section 3.1. The conversion is necessary due to the fact that the processor writes one word at a time, and processing the bits on embedded software can lead to inefficient use of processing cycles. Implementation in logic allows the operation to be

performed within one clock cycle, while it may take multiple cycles if the equivalent operation is performed on the processor.

To provide more options to embedded programmers, the transmitter allows the embedded software to specify a station through both bit masking or simply with a station number, i.e., a station can either be represented by bit masking as "0010 1000" or simply as station number 8. Representing a station with a station number relieves the programmer from having to know the system architecture details and allow more systematic programming of parallel applications.

The transmitter is responsible for handling data transmission of different sizes through the DMA mechanism. While the DMA transmits a large chunk of data, new requests for data transmission cannot occur; this protection mechanism is implemented with a critical section in the embedded software. Because memory used to store data can be overwritten by the processor while data is being transmitted, a transmit buffer is used to store the data while transmission is in progress. Access to the transmit buffer is also guarded by a critical section so that data integrity is maintained.

Transmission process begins by setting a mutex "TRIGGERED" to indicate that transmission is in progress. All packet-related data are transmitted to the station-ring interface through request on the SOCP communication channel interface. At the beginning of the DMA transmission, the information necessary to construct packets is transmitted. Then, in each clock cycle, the transmit FIFO status is checked, and if the FIFO is not full, one word of data is sent to the station-ring interface. The transmission process is temporary halted when there is no more space in the transmit FIFO. The process of data transmission continues until it has transmitted packets equal to the amount specified in the TxSize register. At the end of transmission, the "TRIGGERED" mutex is released to allow more transfer requests. Note that transmit requests can be issued at any time, but no operation will be performed as long as the TRIGGERED mutex is asserted. It is up to the programmer to check the TxStatus register before initiating any DMA transfer.

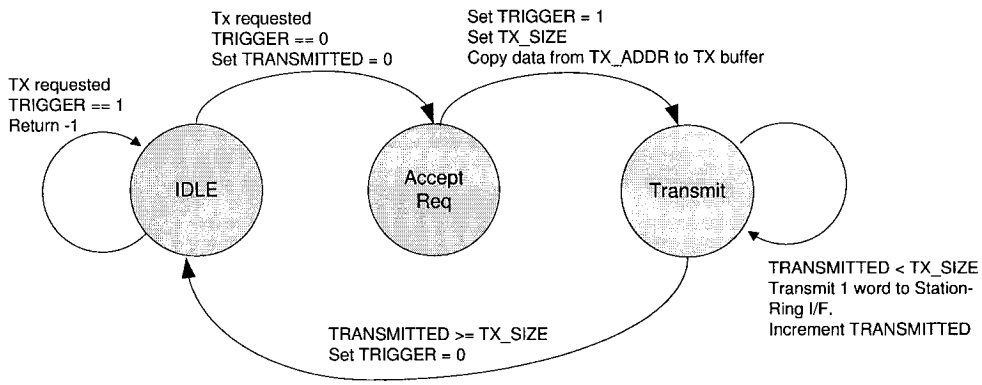

Figure 4.7: State diagram for the transmitter module

Figure 4.7 illustrates the state diagram for the logic implemented on the transmitter. Figure 4.11 later in this section contains a flow chart that illustrates the operation for data transmission with DMA.

4.2.7 Receiver Modules

In message-passing parallel programs, data reception is achieved through an explicit call to a receive function in the embedded program. As opposed to its name, the receiver module does not actually receive data from the interconnect network. Instead, the module acquires received data from the stationIn FIFO on the interconnect module. The state diagram in Figure 4.8 illustrates how the receiver module operates, and Figure 4.12 later in this section contains the flow chart that illustrates the operation for data Reception with the DMA.

Similar to the transmitter, the receiver module has a set of registers that are used to store information necessary for control and monitor of data transmission. The receiver registers and their functionalities are listed in Table 4.3.

RegisterName	Description
MemAddr	Memory address where the received data will be stored
RxSize	Length of data (in number of 32-bit words) that will be received Write access to the register initiates DMA data reception
RxStatus	Read access to the register returns status of reception. busy = 0, idle = 1

Table 4.3: Receiver control and monitoring registers

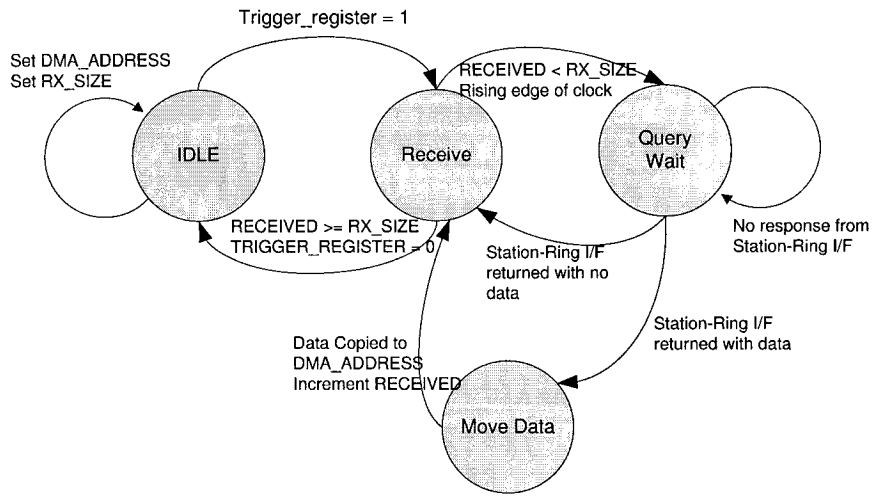

Figure 4.8: State diagram for the receiver module

When a receive operation is initiated by the embedded program, the software sends the address of memory location in which the received data will be stored. Embedded programs also must specify to the receiver the length in the number of words that it is expecting. The DMA receive operation is initiated when a write access is detected at the RxSize register. The MemAddr register must be specified before a write to RxSize to guarantee proper data transmission. Reception of data is handled by a DMA module which periodically checks the receive FIFO for newly-received packets. After being initiated, in each clock cycle, the DMA sends a request with an operation code to the station-ring interface indicating that it wants to receive data. The data field on the first FIFO slot is returned as the response to the receiver module. The receiver will process the returned operation code. If the operation code indicates that data is available, the data field in the SOCP response is copied to the memory location previously specified. If the operation code indicates that the FIFO is empty, no action will be taken in the receiver. The DMA component continues to fetch data from the station-ring interface until it has fetched the specified amount of data, at which point RxStatus is cleared to indicate the end of transmission. The embedded program must write to the RxSize register each time it wants to receive data.

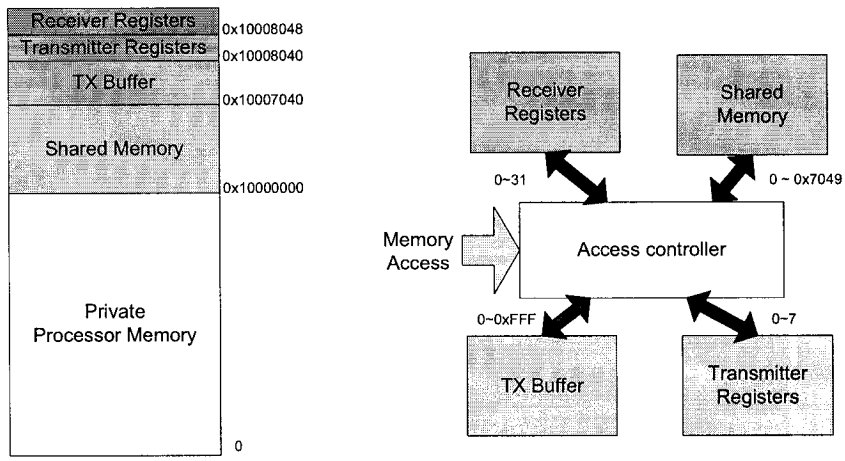

Figure 4.9: Station memory map

4.2.8 Memory Map and Access Controller

Modules in the station are categorized into master and slave groups and are connected through SOCP interfaces. Each slave device is assigned to an address range; access to the device can then be made through the memory map. The access controller connects all the devices in the station, and it is responsible for decoding memory accesses and forwarding requests and responses to the corresponding device. Figure 4.9 illustrates the memory mapping of the components in the station. The private processor memory can be accessed by the processor directly without going through the access controller. Access to components including the transmitter, receiver, shared memory, and the transmit buffer must go through the access controller.

The processor module is the heart of the station that coordinates other devices to perform specified tasks. The processor is capable of communcicating to other components in the station through memory access. A memory address decoder on the processor module allows the processor to perform reads and writes to its internal memory space. Based on the accessed memory address, the address decoder looks up the address in the memory map file and performs necessary operations. In case of access to other devices, the decoder sends access request through the SOCP interface to the access controller, which forwards memory access to the corresponding components in the station. The decoder is configured to allow other types of operations such

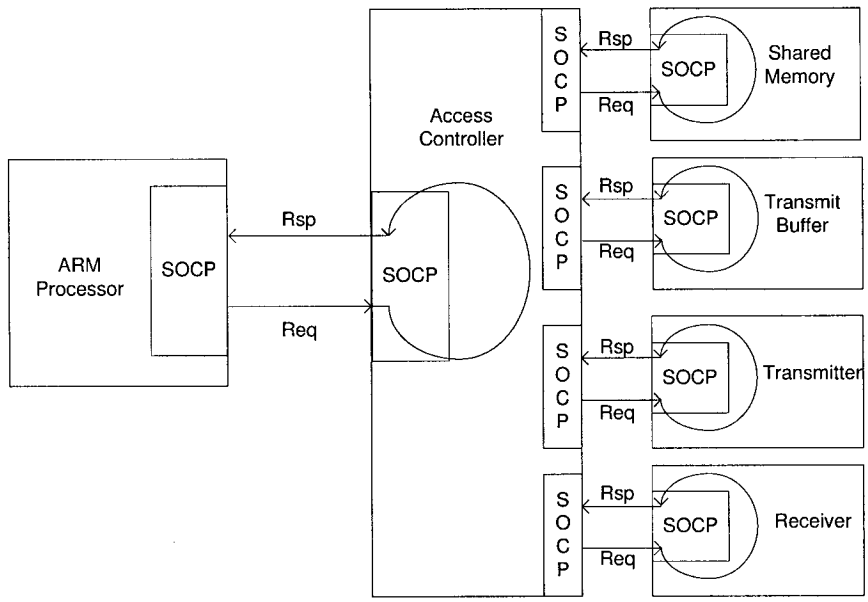

Figure 4.10: Transactions-level communications between the components

as program termination.

Figure 4.10 illustrates how the devices are connected through SOCP interface. The processor module acts as the master device in communications between the processor and the access controller. The task performed by the access controller is to forward the request from the processor to the corresponding slave devices. In the process of forwarding requests, the access controller becomes the master device for communications to the slave devices.

4.2.9 Partitioning of Software and Hardware

At one point in the design, a decision must be made regarding where to draw the line between the software and hardware. In the described multiprocessor architecture, the boundary is located at the station module. The operations necessary for controlling basic message passing include transmission and reception of data through the interconnect network. The logical partition for the DMA implementation described is to have the software performing operations that set up the DMA mechanisms and allow the hardware to handle the actual data transmission. Figure 4.11 and Figure 4.12 illustrate how the software and hardware are partitioned for

transmiting and receiving data.

4.3 Embeded Software API

The application programming interface (API) is the bridge between the embedded software and the hardware components. The API provides a layer of abstraction for the system to perform low-level operations. Each operation in the API performs some specified task where the details of the operation are hidden to the application software designers. The goal of API is to provide ways for applications at a higher layer of abstraction to perform specified functions independent of the hardware specific details. Because users do not need to understand the low-level details, programs can be made portable as long as the API functions are implemented properly for the hardware.

The API package developed for the multiprocessor platform contains functions for sending packets from a station and receiving packets at a station. The API function developed for basic message passing are summarized in Table 4.4. The API implements the software portion of the processes illustrated in Figure 4.11 and Figure 4.12. Note that the send operations provide the software designer the flexibility of specifying the target station address in either decimal notation or in bit-masking notation. The conversion from decimal to bit-mask is performed by the transmitter logic.

Operation	Type	Description
Send	Normal	Send one word of data to a specific station
	Burst	Send multiple words of data to a specific station
MSend	Normal	Send one word of data to multiple stations specified by TargetMask
	Burst	Send multiple words of data to multiple stations specified by TargetMask
Receive	Configure	Configure the DMA to receive number of words of data as specified in the RxSize register and store results in MemAddr

Table 4.4: message passing API for the embedded software

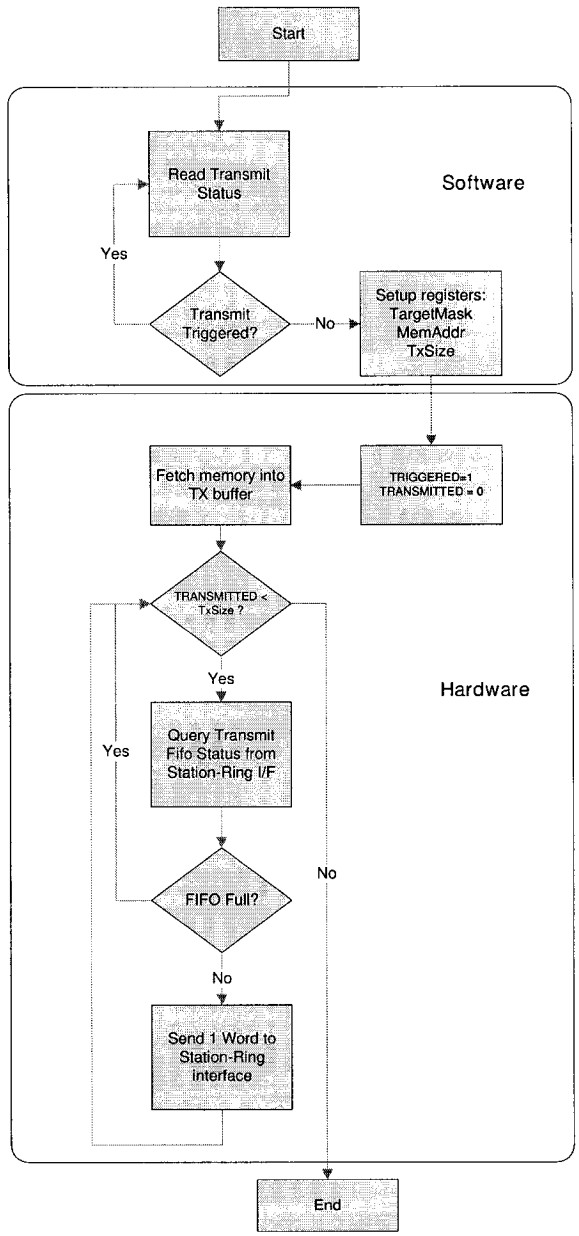

Figure 4.11: Software and hardware partitioning in transmitter

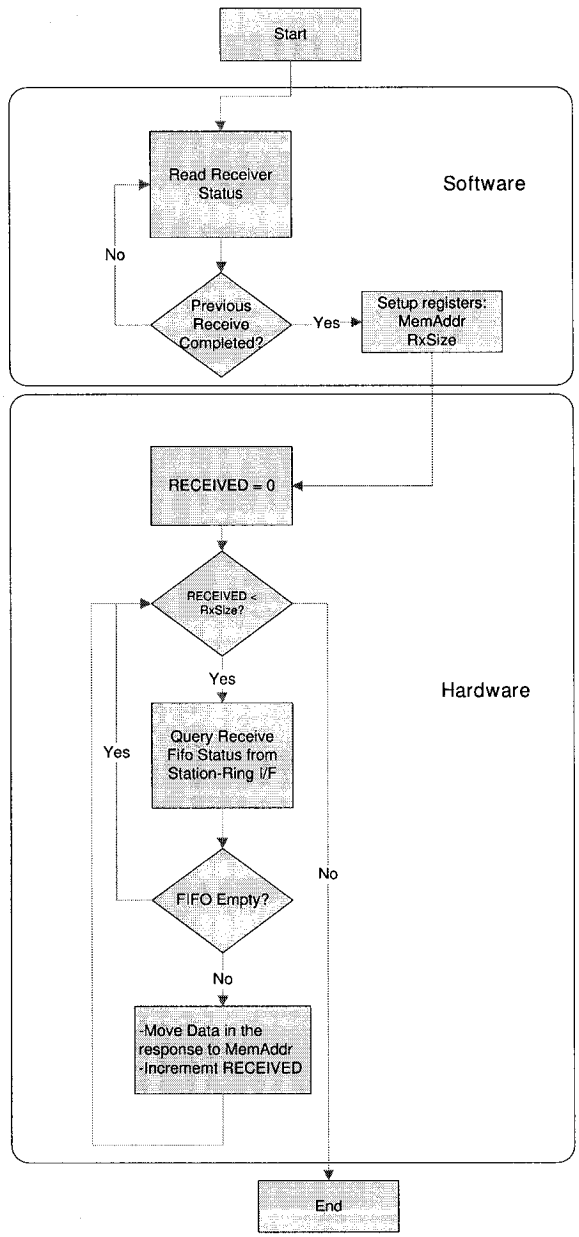

Figure 4.12: Software and hardware partitioning in receiver

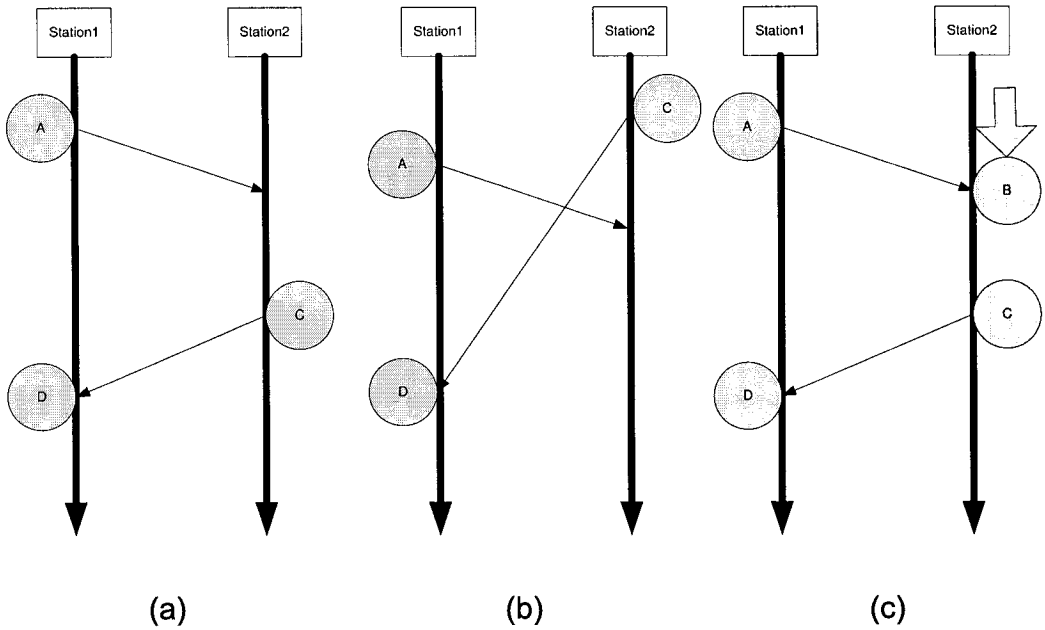

Figure 4.13: Program synchronization

4.4 Simulation Environment

Parallel programs written for the system can be compiled and executed on the SystemC model simulator. Program code written for the system must first be compiled with the cross-compiler to produce the binary file that uses the ARM processor instruction set. The simulator loads the binary file into the ISS simulator in the ARM processor SystemC module and begins program execution. All stations in the model execute the same program code, but use branch instructions to direct program flow on different stations to execute different sections of the program.

The programs running on the described platform require special attention in handling program termination. Because all processors execute one instance of the same program locally, synchronization is required to maintain of the program states. Program synchronization on message passing systems is performed through send-receive pairs. Each word sent in one station is matched with a receive operation in the target station. Station-to-station synchronization can be achieved with blocking reads on the receiving station where the receiver blocks while waiting for an expected message. Blocking reads can be implemented by looping the

instructions that check the receive status register. The loop terminates when the status register indicates all data have been received. Similarly, system-wide synchronization can be achieved through the use of broadcast messages coupled with blocking reads that receive one message from each station. The process is illustrated in Figure 4.13.

Figure 4.13(a) shows the desired behavior: Station1 sends a message at A, Station2 receives the message from Station1 and sends the processed message at point C back to Station1. Figure 4.13(b) shows a possible scenario which may occur and cause problem. As each station run independent of another, Station2 may reach point C before Station1 reaches Point A, then Station1 will get erroneous data at point D. To prevent the above-mentioned behavior, the two stations must synchronize in order to ensure that the data is sent at the right time and under the right conditions [31]. Figure 4.13(c) shows how synchronization can be achieved through send-receive pairs. A blocking read is placed on station2, the blocking read operation will force station2 to wait for the message from station1, and then the right response from station2 will be received at station1.

The ARM processor SystemC module from stepNP was designed to run in single-processor simulation environment. When the program terminates, it is assumed that the simulator can be terminated. In the multiprocessor simulation platform, not all instances of the programs terminate at the same time. Termination of an instance of the program can confuse the processor module because it indicates to the ISS emulator that the program has completed. ISS emulator would terminates when an instance of the program terminate while there are still instances of the programs that continue to require the ISS emulator.

To handle the above-mentioned issue, the embedded software is required to signal the station module for program termination by writing to a reserved register, and then keeping the program in a while loop. A static variable in the station class is used to keep track of the number of stations that have terminated their programs, and the simulator terminates when all stations have written to the register.

Chapter 5

Results

This chapter describes the details of a parallel program running in the simulation platform. Embedded software are designed to demonstrate proper operation of the system, which includes the interconnect network, station cores, and the software API. The model is also used to evaluate design parameters for the design. Three synthetic testbenches were developed to simulate applications with different traffic characteristics. Then, design space exploration is performed on some of the design parameters for the system. The synthetic testbenches as well as one of the parallel programs developed are used as the reference applications for the design space exploration. Finally, analysis of testbenches and simulation results which contribute to finding optimal design parameters are also presented.

5.1 Parallel Programs

The general concept of parallel processing is to divide a large task into smaller and more manageable sub-tasks. The sub-tasks can then be executed simultaneously across multiple processors to give higher processing throughput. In the ideal case, a parallelized program would take less time to execute until completion than the same program running on a single processor. In reality, there are communication overheads that are associated with parallel processing, and may result in negligible performance improvements or even worse performance depending on

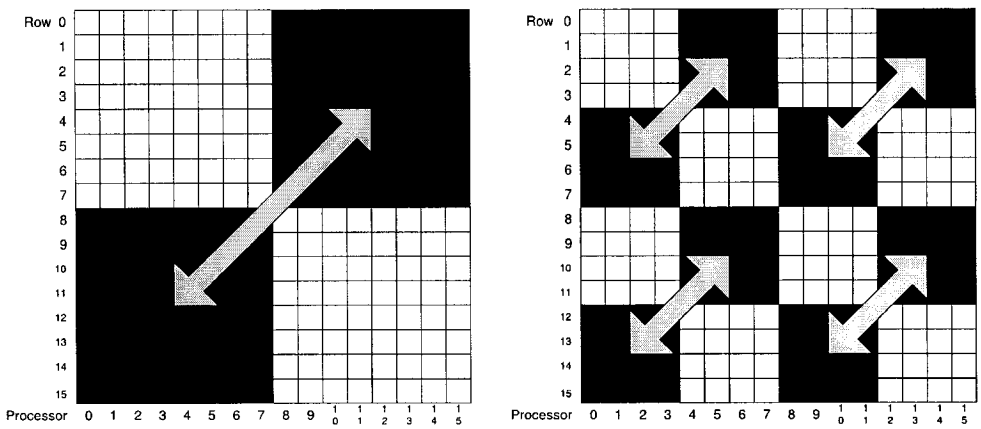

Figure 5.1: Transpose algorithm

the type of the application.

The process of partitioning an application into sub-tasks for parallel execution is generally non-trivial. Normally, parallel programs would minimize the amount of communication and have most of the processor cycles processing the sub-tasks that were assigned to them.

5.1.1 Matrix Transpose

The parallel application discussed in this section performs matrix transposition using the algorithm presented in [32]. The algorithm is a representative application which can be used to stress the functional accuracy of the interconnect network with above-average inter-processor communication. In the algorithm, each processor holds one column of the matrix. At each iteration of the algorithm, the matrix is divided into four sub-blocks where the lower left block in the matrix is then swapped with the upper right block. A synchronization point is placed at the end of each iteration to make sure that all processors have successfully updated their respective columns. Each of the sub-blocks is then processed in the next iteration. The algorithm continues until each sub-block can no longer be sub-divided (i.e., each is a 1×1 matrix). The process is illustrated in Figure 5.1. Because the system that is modeled in this thesis has up to 16 processors, the original algorithm [32] has been modified to make each processor handle

multiple columns when the matrix size is greater than 16×16.

The matrix transpose program has above-average communication overhead because the main function of the program on each processor is to swap data with all other processors. The amount of data exchanged in each iteration for a station amounts to half of the data the station is holding. Therefore, the communication overhead can be significant for a large matrix. The program is especially useful during development of the ring interconnect. In the algorithm, each station will exchange data to each of the remaining stations for at least once so all possible communication links can be tested. The correctness of the result can be easily observed by simply verifying that the initial row data appears in each column of the resulting matrix. The traffic characteristic of the program can also be manipulated by varying the maximum size of data that each station will transfer at each send operation.

The transpose parallel program execution can be divided into three distinct phases: the initialization phase, the communication phase, and the computation phase. In the initialization phase, the allocated memory space is initialized to default values. The initial values of the matrix are also generated in the initialization phase. In the communication phase, data is swapped between the stations. Data to be transmitted is prepared in a buffer and transmitted to the destination station. Data received from the other station is also processed, and it replaces the corresponding location in the column of the matrix held by the station. Because each station may be responsible for multiple columns of the matrix, in the computation phase, each station performs transpose operations within the local memory and does not need to communicate through the interconnection network.

5.2 Design Space Exploration

The system model that is implemented in SystemC is used to assist in finding optimal design parameters for the on-chip ring interconnect. Because design specifications cannot be modified after production of the SoC design, the system must be optimized in early stages of the

design. The system model provides an inexpensive alternative to evaluate the impact on performance for different design parameters. Design parameters that are explored for the two-level hierarchical ring design are: *maximum burst length* recommended for the embedded program, *maximum memory access time* for moving received packet contents into memory, *FIFO depth* for FIFOs in the interconnect network, and *relative operating frequencies* for the rings in the system. The importance of each design parameters is discussed in more detail in later sections.

The design parameters are most relevant in applications with high communication-to-computation ratio, where the program is composed primarily of communication components. Applications with high computation-to-communication ratio do not utilize the interconnect network enough to have significant impact to the overall performance. Because the communication overhead is considered negligible in computation-intensive applications, it is not the primary focus for optimization in those applications. The following subsections describe the testbenches used in simulations, and they also present simulation results and analysis of the findings.

5.2.1 Testbenches

Three synthetic testbenches were implemented to assist in the selection of optimal design parameters. Each testbench transmits a specified amount of data with a configurable maximum burst length for each transmission. The testbenches are named *SyntheticHigh*, *SyntheticMid* and *SyntheticLow*. Each testbench has different communication characteristics. In the SyntheticLow testbench, each station transmits a stream of data to a corresponding station in the farthest ring. In the SyntheticMid testbench, each station *multicasts* the stream of data to all stations in the farthest ring. The SyntheticHigh testbench *broadcast* the stream of data to all stations in the system. All three testbenches must receive complete data before termination of the program. The total execution time is measured at the end of the testbench when the program terminates.

The purpose of the SyntheticLow testbench is to simulate communication characteristics of one-to-one communication between the stations. The testbench stresses the communication

network by having all stations communicating to one other station in the system. The SyntheticMid testbench places more emphasis on local-ring utilization, but the amount of global traffic on the central ring remains the same as the SyntheticLow testbench. In the SyntheticMid testbench, the number of inter-ring packets remains the same as the SyntheticLow testbench, but each packet is duplicated at the local-ring level and multicast to all stations in the ring. The SyntheticHigh testbench provides more stress from the amount of traffic by duplicating each packet in the inter-ring level to broadcast to all local rings. Each packet is duplicated again in each local ring so a copy is received by each station. The testbenches are configured to run at 100 MHz clock frequency and transfer 400 words of data from each station. Unless specified otherwise, the FIFO depths are configured to be 16 packets for all FIFOs in the system. All rings runs at the same operating frequency, and memory access latency of 1 clock cycles.

5.2.2 Data Burst Length

The burst length is the number of words a station would transmit at one time with each DMA send operation. Because there are configuration overheads associated with setting up the DMA hardware, embedded programs must select a reasonable data burst length where the configuration overheads are not excessive. To send a fixed amount of data across the network, a larger burst length would have lower configuration overhead because the embedded program does not need to configure the DMA component as frequently. The larger burst length, however, compresses the communication portion of the embedded software so that the instantaneous traffic level is large, which can potentially cause more backpressure in the network and reduce communication performance. The configuration overhead and the higher congestion level are two opposing forces which affect the overall execution time of the embedded software. The designer's goal is then to determine the burst length that best fits the application. Because each transmit operation is matched with a receive operation in message passing systems, data burst length also affects the size of memory which must be allocated to stage data for transmission or to store the received data. Selecting a large burst length would mean that a large amount of

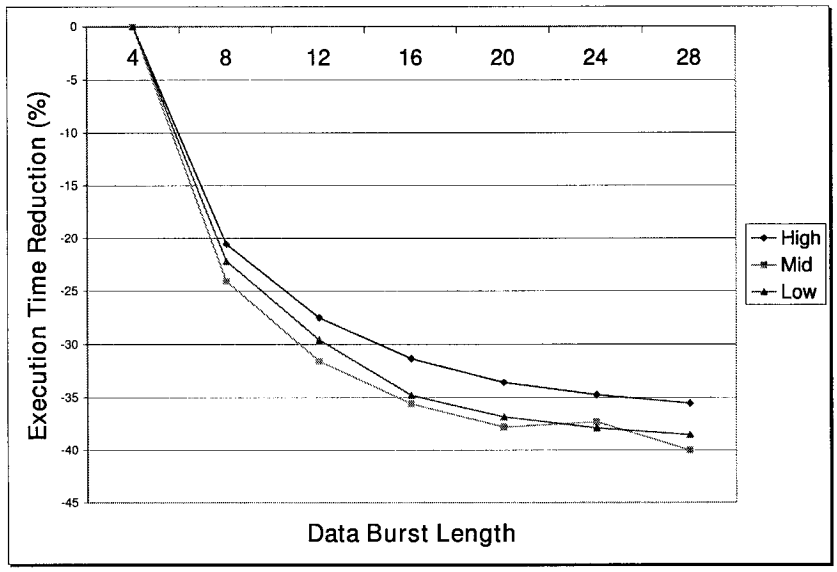

Figure 5.2: Burst length Vs. Execution time (Memory Access Latency = 1 cycles)

memory is reserved unnecessarily.

Figure 5.2 illustrates the effect of different data burst length for the three different testbenches. It can be observed that increasing the burst length can have a large impact on the execution time for a small burst length, then the reduction diminishes as the length is increased. For the three testbenches, *incremental execution reduction becomes small for burst sizes larger than 16*. Further increases in burst length have negligible effect in performance, but introduce higher memory requirements for staging the data for transmission and for storing the received data. The effect of data burst length is investigated further in Section 5.3 with the matrix transpose program which represents a more realistic testbench.

5.2.3 Memory Access Latency

When the DMA mechanism on the station module attempts to move a received packet from the receive FIFO at the station-ring interface, depending on the type of memory component or control logic used, the process may take one or more clock cycles. System performance can vary depending on the memory access time. To ensure good performance, the access time

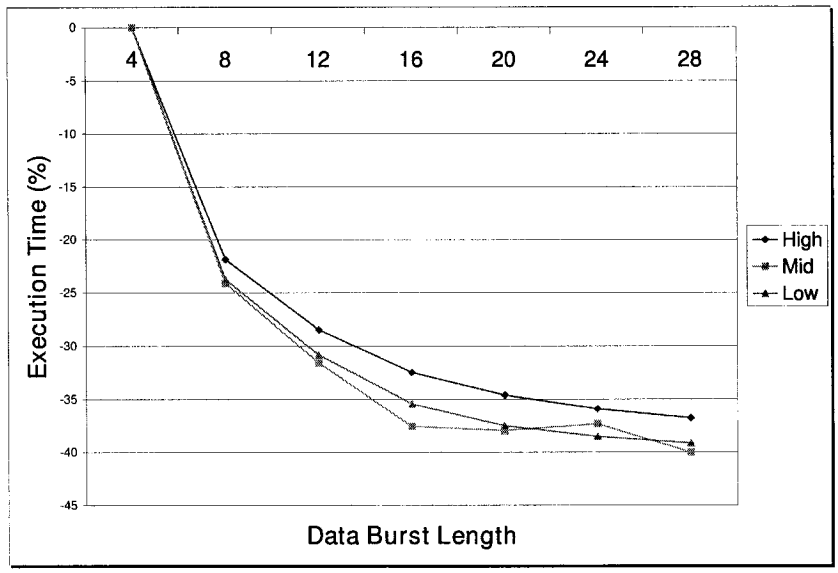

Figure 5.3: Burst length vs. Execution time (Memory Access Latency = 2 cycles)

constraint must be placed in selecting or implementing the memory and memory controller modules. The goal of selecting suitable components such as memory is to reduce the design cost: high performance components may be expensive to obtain or require more complex control logic which requires more space on silicon. A less expensive component can be used as long as it can meet the timing constraints.

To study the effects of memory access time on communication performance, more simulations have been performed with varying burst length for the memory access latencies for the three synthetic testbenches. Results are shown in Figure 5.3 and Figure 5.4. The results indicate that similar behavior is obtained for different memory access latencies. The improvement in execution time diminishes at a burst length of 16 words. The results are summarized in Figure 5.5 to allow a more detailed evaluation of the impact of different memory access latency. It can be observed that the execution times of the three synthetic testbench are not affected for memory access latency less than or equal to 2 clock cycles, and the incremental increases in execution time grows linearly with the access latency. Programs with higher network utilization are more sensitive to increase in memory access latency. The increase in communication time is caused by the station not being able to absorb the received packets fast enough. As a

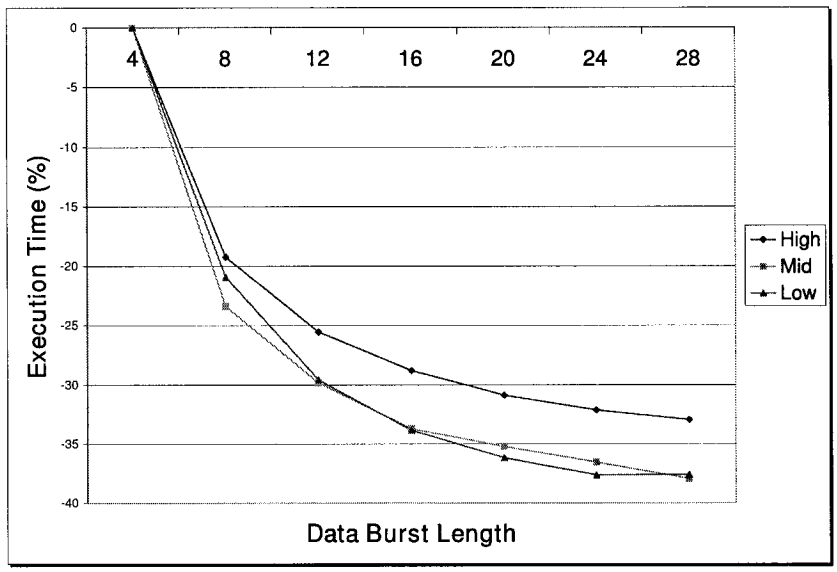

Figure 5.4: Burst length vs. Execution time (Memory Access Latency = 3 cycles)

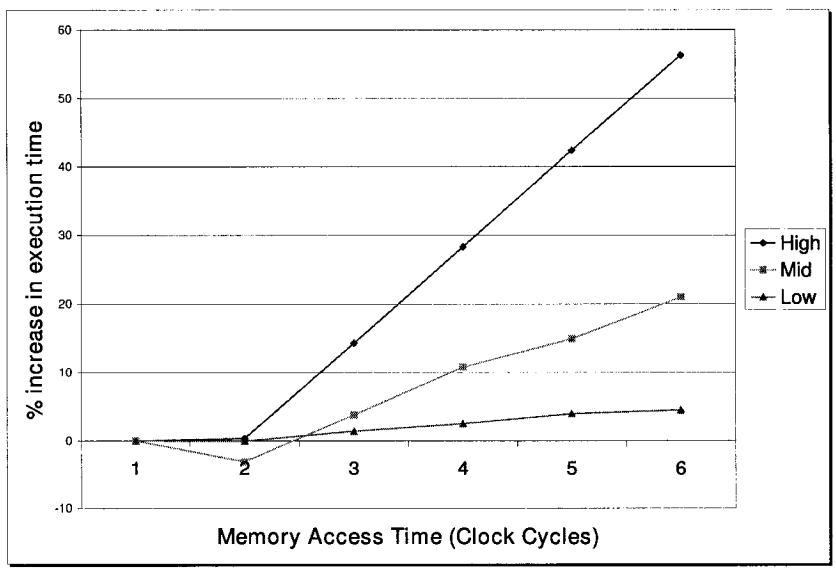

Figure 5.5: Memory access latency vs. Execution time (Burst length = 16 words)

Access Time ($clockcycles$)	Execution Time (ns)	percent increase (%)
1	729810	0
2	729450	-0.05
3	740610	1.48
4	743130	1.83
5	740970	1.53
6	776070	6.34
7	768150	5.25

Table 5.1: Effect of memory access time on communication performance

result, number of packets in the receive FIFO grows faster for systems with higher memory access time, which causes more backpressure that propagates throughout the network.

In addition to the synthetic testbenches, simulations have been performed with the matrix transpose program described in Section 5.1.1. Table 5.1 shows the how memory access time can affect communications in the matrix transpose program. Because we are interested only in the communication portion of the program, the processing parts of the program (i.e., the local transpose operations) are removed to evaluate only the execution time for the actual inter-station communication.

The result for matrix transpose agrees with the results obtained with the synthetic testbenches. Memory access latency less or equal to 2 clock cycles has no effect to the execution time. It is then sufficient to say that the only limitation for memory and controller implementation is to ensure that a memory access can be completed within 2 clock cycles. It should be noted that the clock cycle time is measured based on the local ring operating frequency, where the station IP core may be running at a slower frequency.

5.2.4 FIFO Depth

The depth of FIFOs located on the switches can affect both performance and design area of the system. Design consideration must be made in selecting a FIFO depth that provides good performance at reasonable design area requirement. Table 5.2 summarizes the effect of the FIFO

FIFO Depth ($words$)	Execution Time (ns)			
	High	Mid	Low	Trans
12	644670	466860	412530	740610
16	646290	463500	414420	740970
20	647760	469170	413790	741150
24	647610	466860	409860	740970
28	647550	462180	411690	740970
32	645450	467280	405240	746370
36	645510	472690	410910	746190
40	644670	442290	407760	746190
44	645660	442290	406290	746190
48	645240	442290	406290	746190
	High	Mid	Low	Trans
Max	647760	496860	414420	746370
Min	644670	442290	405240	740610
% difference	0.47	6.87	2.27	0.78

Table 5.2: Effect of FIFO depth on performance

depth on the communication performance for the synthetic testbenches, as well as the matrix transpose program. The performance improvement achieved through increasing FIFO depth is limited for most of the testbenches, while only the SyntheticMid testbench shows a sizable reduction in execution time. Nevertheless, it is still not a good trade-off considering the amount of extra design space required to implement the FIFOs. The simulation results have indicated that the FIFO depth does not have a significant impact on communication performance in the two-level hierarchical ring network. This phenomenon can be explained by noting that a small FIFO size causes more backpressure to be triggered when the network is congested, whereas an increase in FIFO size reduces the number of backpressure occurrences, although it takes longer to recover for those. These two opposing forces for the hierarchical ring are observed to be equal in strength for the applications tested, and results in relatively little variation in performance with different FIFO depths, *Based on the simulation results, it is believed that the FIFO size should be kept relatively small in order to reduce the design area while maintaining satisfactory performance.*

5.3 Collaborative Work on Power Optimization

In addition to the individual efforts undertaken for this thesis, a collaborative project on SoC power optimization is also currently under development. A power modeling framework developed at McGill University by Stephan Bourduas has been integrated with the multiprocessor system model described in Chapter 4. The project adds capability for the model to monitor the simulated energy usage of the interconnect network for the execution of a particular application program. The power model is not intended to provide exact power estimation, but the results obtained by the estimation should be in the same order of magnitude as expected from the physical device. The point of the power modeling is to provide an estimation of relative energy usage on a device with different design parameters.

The matrix transpose program was used to obtain the execution time and the power estimation of the interconnect network for transpose of a 128x128 matrix. The first part of the simulation experiment finds the optimal burst length for best performance. The second part of test finds the best configuration for the clocking scheme to provide optimal power characteristics for the architecture.

The results of simulations performed with different data burst lengths for a 128x128 matrix transpose are shown in Table 5.3. It can be observed that short data burst lengths have longer execution times. The execution time decreases until a burst length of 16 words, then execution the time starts to increase. The higher execution time at a lower burst length results from the fact that a station must configure the transmitter and the receiver. The overhead for data transmission becomes dominant if a station needs to configure the transmitter and receiver for each word it transmits. Higher burst lengths also increase execution time from congestion caused by higher utilization of the interconnect network. For burst transfer of data, one packet can be transmitted at each clock cycle. Hence, network utilization increases with the data burst length. Congestion results from higher network utilization and may cause flow-control to be activated more often and cause the execution time to increase. Flow-control stops the transmission of new packet when the receive FIFO for a station is nearing its capacity. Stations

Burst Length (words)	Execution Time (ms)
1	20.72
2	15.17
4	12.49
8	11.26
16	10.96
32	11.31
64	12.60

Table 5.3: Interconnect performance for different burst lengths (128x128 matrix transpose

in the same ring are prevented from sending new packets to the network even when the upward path is not blocked.

Frequencies of the ring clocks can be varied to optimize the power requirement of the interconnection network. In order to compare the system performance for different interconnect speeds, the following two parameters were evaluated.

- The *local divisor* is the factor by which the *local rings* are slowed down compared to the global clock.
- The *central divisor* is the factor by which the *central ring* is slowed down compared to the global clock.

For example, a local divisor of 2 would indicate that the local rings are running at a clock speed which is half of the global clock frequency. Simulations were performed with burst a size of 16, which was previously established to give the best performance. The results of simulations with varying ring speeds are listed in Table 5.4 for matrix transpose with a 128x128 problem size. It can be observed that the ring speeds have less impact on the execution time, but can achieve substantial power savings with slower operating frequencies. The overall execution time can be longer because the latency increases with lower ring operating frequencies. The execution time, however, is not affected significantly by the ring operating frequencies due to the fact that the interconnect network is still able to transmit data with good throughput. Because the interconnect network can hold multiple packets, a pipelining effect can be

Local Divisor	Central Divisor	Exec Time (ms)	Power (mW)
1	1	10.96	42.68
2	1	10.97	23.57
2	2	11.05	21.47
2	3	11.13	20.80
2	4	11.15	20.44
3	1	10.99	17.21
3	2	11.06	15.09
3	3	11.17	14.39
3	4	11.15	14.07
4	1	11.04	14.03
4	2	11.10	11.91
4	3	11.19	11.21
4	4	11.29	10.86
5	1	11.07	12.12
5	2	11.12	10.00
5	3	11.20	9.29
5	4	11.25	8.95

Table 5.4: Effect of varying ring speeds for burst size of 16 words

observed. In the simplest case, if there is only one station sending packets to another station in the network, the station can still send one packet per clock cycle even when the network is operating at lower frequency. Similarly, the receiving station can receive one packet per clock cycle. The performance impact of lowering the central ring frequency can be observed to be more significant than lowering the local ring frequency. The result can be explained by the fact that the central ring is the throughput bottleneck because it is responsible for all of the global traffic from the four local rings. In terms of power saving, we can observe a large reduction in power usage by simply slowing down the local rings. The reduction in power usage comes from savings in dynamic power from lower operating frequency. The power savings from reducing the local ring operating frequency diminishes with a higher local divisor because the static power portion becomes dominant. From results listed in Table 5.4, we believe that a local divisor of 2 and a central divisor of 1 is the optimal clocking scheme for the hierarchical-ring multiprocessor.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis has discussed the process of modeling and evaluating a multiprocessor system targeted for chip-level implementation, with an emphasis on the interconnection network. The interconnect architecture that is studied in this thesis is based on the hierarchical ring network that was implemented in the NUMAchine multiprocessor platform. This choice is based on advantages of the original ring network such as the simplicity and modularity that would also be beneficial for chip-level implementation. To study the chip-level adaptation of a ring-based hierarchy, a model of the architecture has been implemented in the SystemC modeling language. Simulation experiments have been conducted to verify proper system behavior, as well as perform design space exploration to study the effects of some design parameters.

Requirements for successful SoC development are identified as follows:

- Short development time: IP reuse reduces the design and verification complexity, and allows SoC to be developed within short time frame. Modular components together with standard interfaces can also assist in reducing the time required for development of SoC.

- Lower cost: Integration of system components into a single chip reduces the packaging cost. However, the size of design can have large impact to the production cost. Large

designs have higher silicon area requirement, as well, production yield reduces as the design area increases.

- Low power: SoC is common in embedded applications which have strict power requirement.

The proposed system is designed to fulfill the stated requirements. Here is a summary of the development:

- A simpler memory structure was implemented to lower the expected silicon area requirement for the interconnect network and the buffer size in the stations. The distributed memory structure implemented in the design uses message passing communication and relay on minimal hardware support. Memory management tasks are moved to the software level in order to reduce the control logic complexity.

- The potential of deadlocks are reconsidered to reflect changes in the communication characteristic introduced by modification to the memory structure. An improved direct memory access hardware implemented with the software API is used to enhance performance of communications between the stations as well as removing the potential for network deadlocks introduced in software-level.

- A software model was developed to verify the functional correctness of the proposed design. The model was implemented in SystemC for its flexible refinement design approach. When the technology becomes available, the modeled system can eventually be synthesized into gate-level net list and fabricated on silicon. Following the SoC design methodology, the software model is partitioned into two modular components so maximum IP reuse can be achieved to reduce the development time. The implemented model consists of approximately 4000 lines of C++ code (excluding components provided by the StepNP library).

- An estimate of the silicon area required to implement the interconnect network on chip is performed. The interconnect network component is composed of switches which include the inter-ring interface and the local-ring interface together with connection wires. The switch is implemented in Verilog and synthesized with Synopsis design compiler. The synthesis tool estimated the area of each interface to be $0.225mm^2$ which gives total design area of $4.5mm^2$ using TSMC 0.18μ technology library.

- Design parameters which can affect the design size and performance are identified. Because communication overheads are considered negligible for computation intensive applications, testbench of different traffic characteristics are developed. Design space explorations are performed for various design parameters and the optimal settings are determined based on the simulation results.

6.2 Future Work

Improvement for higher performance can be achieved by additional computational units in the system. The multiprocessor station core developed for the proposed system can add processing throughputs for parallelizable applications. Current station core is designed with one ARM processor but allow more processors to be added. Inter-processor communication can be performed through mutex locks and variables located in the shared memory within each station. For the proposed system, packet corruptions affect not only the data transmitted, corruptions to the packet header can result to packets being transmitted to invalid destination. In such case, the packet will need to be retransmitted. Because the design approach of the proposed system places emphasis on minimal hardware and simple control logic, the retransmission protocol should be implemented in software.

Bibliography

[1] International technology roadmap for semiconductors, 2001 edition. Avaliable at `http://public.itrs.net`.

[2] Eric Y. Chou and Bing Sheu. System-on-a-chip design for modern communications. *Circuits and Devices*, pages 12–17, November 2001.

[3] *SystemC Version 2.0 User's Guide — Update for SystemC 2.0.1.* Available at `http://www.systemc.org`.

[4] K. Loveless. The implementation of flexible interconnect in the NUMAchine multiprocessor. Master's thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 1996.

[5] R. Grindley et al. The NUMAchine multiprocessor. In *Proc. 29th Int'l Conf. on Parallel Processing*, pages 487–496, Toronto, Ontario, August 2000.

[6] *Functional specification for SystemC 2.0.* Available at `http://www.systemc.org`.

[7] The important of sockets in soc design. available at `www.ocpip.org`.

[8] W.J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the Design Automation Conference*, 2001.

[9] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *IEEE Computer*, 35, January 2002.

[10] Vijay Raghunathan, Mani B. Srivastava, and Rajesh K. Gupta. A survey of techniques for energy efficient on-chip communication. In *Proceedings of the 40th conference on Design automation*, pages 900–905. ACM Press, 2003.

[11] Dongho Yoo and Inbum Jung. Multistage ring network: A new multiple ring network for large scale multiprocessors. In *International Workshops on Parallel Processing*, pages 290–294, Japan, September 1999.

[12] Cesar A. Zeferino, Márcio E. Kreutz, Luigi Carro, and A. Susin. A study on communication issues for systems-on-chip. In *Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design (SBCCI 02)*, 2002.

[13] Luca Benini Terry Tao Ye and Giovanni De Micheli. Packetization and routing analysis of on-chip multiprocessor networks. *Journal of Systems Architecture*, 50, 2004.

[14] L. Benini and G. De Micheli. Powering network on chips. In *Proceedings of the* 14^{th} *International Symposium on System Synthesis*, pages 33–38, 2001.

[15] A Brinkmann, J-C Niemann, I Hehemann, D Langen, M Porrmann, and U Ruckert. On-chip interconnects for next generation system-on-chips. In *ASIC/SOC Conference*, pages 211–215, 2002.

[16] Andr Ivanov Partha Pratim Pande, Cristian Grecu* and Res Saleh. Switch-based interconnect architecture for future systems on chip. In *Proceedings of SPIE, VLSI Circuits and Systems*, pages 228–237, 2003.

[17] S. Dey F. Karim, A. Nguyen and R. Rao. On-chip communication architecture for oc-768 network processors. In *Proceedings of the Design Automation Conference*, pages 678–683, 2001.

[18] James F. Kurose and Keith W. Ross. *Computer Networking, A top-Down Approach Featuring the Internet.* Addison-Wesley, 2003.

[19] Jamaloddin Golestani Mark Karol and David Lee. Prevention of deadlocks and livelocks in lossless backpressured packet networks. In *Transactions on Networking*, volume 11, December 2003.

[20] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture*. Morgan Kaufmann Publishers, Inc., 1999.

[21] Klaus D. Gunther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communications*, 29(4):512–524, April 1981.

[22] Sy-Ye Kuo Shih-Chang Wang, Hung-Yau Lin and Yennun Huang. A simple and efficient deadlock recovery shceme for wormhole routed 2-dimentional meshes. In *Procedings of Pacific rim International Symposium on Dependable Computing*, pages 210–217, December 1999.

[23] Pieere G. Paulin, Chuck Pilkington, Essaid Bensoudance, Michel Langevin, and Damien Lyonnard. Application of a multi-processor soc platform to high-speed packet forwarding. In *Procedings of Design, Automation and Test in europe Conference and Exhibition Designers' Forum*, 2004.

[24] Drew Wingard. Micronetwork-based integration sof socs. *DAC*, June 2001.

[25] T. Theis. The future of interconnection technology. *IBM Journal of Research and Development*, 44(3):379–390, May 2000.

[26] *SystemC 2.0.1 Language Reference Manual, a hardware/software approach*. Available at `http://www.systemc.org`.

[27] P. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A system-level exploration platform for network processors. *IEEE Design and Test of Computers*, 19(6):17–26, November/December 2002.

[28] *SystemC Version 2.0.1 Master/Slave Communication Library.* Available at `http://www.systemc.org`.

[29] OCP-IP. *Open Core Protocol Specification.* available at `www.ocpip.org`.

[30] Wander O.Cesario, Damien Lyonnard, Gabriela Nicolescu, Yanick Paviot, Sungjoo Yoo, Lovic Gauthier, and Mario Diaz-Nava. Multiprocessor soc platforms: A component-based design aproach. *IEEE Design & Test of Computers*, pages 52–63, 2002.

[31] Jane W.S. Liu. *Real-time systems.* Prentice Hall, 2000.

[32] Ian Foster. Matrix transposition. `http://www-unix.mcs.anl.gov/dbpp/text/node126.html`.