NoCSim: A Versatile Network on Chip Simulator

By

Michael Jones

B.ASc Queen's University, Kingston 2002

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

In

The Faculty of Graduate Studies

Electrical and Computer Engineering

The University of British Columbia

April 2005

NoCSim: A Versatile Network on Chip Simulator

ABSTRACT

The new network on chip paradigm that has been proposed involves radical changes to SoC design methodology. In this paradigm large numbers of heterogeneous IP blocks will by integrated together using a standard template. Each IP block is capable of sending and receiving data packets through an interconnect. The non-scalability of buses as on-chip interconnects forces us to select a more scalable alternative for communication.

Many different network-centric interconnects have been proposed for the large scale SoC domain such as *k-ary n-cubes*, *butterfly fat-trees, k-ary n-trees*, and *octagons*. With this network-on-chip (NoC) paradigm many new design challenges arise such as physical switch design, and network topology selection. Without any other means of predicting system performance, a network simulation tool is required to evaluate and compare networks.

Several network simulation tools exist, but fail to contain all functionality desired for NoC simulation (e.g. *wormhole switching* support). To fill this void we have developed *NoCSim*, an iterative *flit*-level network on-chip simulator capable of simulating networks under a wide variety of parameters and topologies. The tool was developed, tested, and verified against an established network simulator.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First of all, I'd like to thank my academic advisor, Dr. André Ivanov for his advice, motivation, and guidance throughput my time here at UBC. There were many times where the direction of my work was in question and it was then that he helped me the most.

I'd also like to thank Dr. Res Saleh for his feedback and support on technical matters as well.

I need to send a huge thank you to fellow members of Dr. Ivanov's research group, Partha Pande, and Cristian Grecu. Whenever I needed help they were there and willing to drop whatever they were doing to help me. I can't say enough about the appreciation I have for them, and without them this work would never have been done.

I'd also like to thank USC PhD student and member of the *SMART* Interconnects group, Wai Hong Ho, for his continued support on technical issues.

1.0 Introduction

The years to come will bring revolutionary changes to System on Chip (SoC) design methodology [28]. Complex SoCs consisting of billions of transistors fabricated in technologies characterized by 65 nm feature sizes and smaller will soon be a reality. At this physical size the number of semiconductor intellectual property (SIP) blocks could be in the hundreds. Each of these blocks is capable of performing its own unique function and is free to operate at its own clock frequency. However, in such a large system, integration of these heterogeneous blocks gives rise to new challenges. These problems include non-scalable wire delays, failure to achieve global synchronization, signal integrity issues and difficulties with non-scalable bus-based interconnects. It has since been established that the key to success for these large scale SoCs is the interconnection system [4].

The most frequently used on-chip interconnection architecture thus far is the shared medium bus where all communicating devices share the same transmission medium. While this topology has advantages for small networks, as the bus line increases in length to accommodate additional IPs the performance drastically declines. Intrinsic parasitic resistance and capacitance from the line and added IPs increase to extremely high levels causing propagation delay to be unacceptably high. A bottleneck also occurs as the large number of IPs must wait for their turn to use the only transmission medium [31].

It is for these reasons that several research groups have turned to the direction of a network-centric approach to integrate IPs in large scale SoCs [2,4,14,20]. In this approach the IPs are decoupled from the communication fabric removing the need for global synchronization. Independent IPs can then run on their own local clock frequency allowing IP specific clock trees to replace a system global tree.

Thus far, the suggested interconnects for large scale SoCs closely resemble interconnect architectures of high-performance parallel computing systems. The possible interconnect topologies come in many varieties, such as mesh or tree. There are also many configurable physical interconnect parameters such as switch buffer depth and size. Routing, collision handling, and deadlock are also issues that like the physical parameters, can greatly impact the overall interconnection system's performance [21].

As large scale SoCs are pushed into development, hardware designers will need to face the challenge of designing a complex interconnects. With so many interconnect options and configurable parameter permutations available, designers require a means of educating themselves of the impact the different interconnects have on overall system metrics, performance, power consumption, and die area. The ability to weight different networks against others on a unified platform is vital as only then will designers be able to make informed decisions regarding NoC design [21].

The above problem could be solved through one of a number of solutions. One is to create mathematical analytical models possible of describe all network options and retrieving performance measurements [17]. If it were possible to create completely accurate models of every interconnect topology and parameter configuration this would be the ideal solution. Once established, models could be used with very little time or effort to predict accurate system performance. Unfortunately mathematical models for large scale complex topologies have not been successfully developed when wormhole switching is the switching technique. *Wormhole switching* is the obvious choice for on chip network switching since it allows switches to have buffer sizes that are fractions of the size of *packet switched* buffers [31]. Thus, the added complications that wormhole switching bring to the mathematical analysis problem make this solution unrealistic, and unobtainable.

Another solution would be to gather the required performance statistics by measuring fabricated large scale SoCs. This of course cannot be done since no SoC with the scale discussed above has been developed. Even if one were to be fabricated, that could provide performance measurements for only one of many possible interconnect configurations. Therefore, educating large scale SoC designers through hardware testing is not an option.

The only remaining realistic solution is to emulate the behaviour of the on chip interconnect to produce performance statistics using a network simulator. With standard network performance defining statistics such as average message latency, and throughput,

designers can clearly weight interconnects against each other and make informed decisions [22]. These statistics would be as accurate as the models used in the simulator and the emulated traffic pattern used. A simulator tool would also be advantageous as it could allow designers to evaluate a large number of parameter permutations in their aim to design the optimal interconnect for a given NoC.

A simulation tool should be able to evaluate different NoCs in terms of different parameters and conditions. Desirable features for an NoC simulator include:

- The ability to simulate different topologies structures and sizes.
- The ability to simulate under different traffic patterns and loads.
- The ability to vary switch parameters such as buffer depth, and number of virtual channels.
- The ability to implement different routing and collision handling schemes.
- The ability to implement *wormhole switching*

Several network simulators such as *OPNET* [36], *Flexsim1.2* [27], and *NS version 2.0* [25] have been developed both in academia and industry. While these tools are functional in their own domain, none provide all the required features for evaluating on-chip networks in a convenient and unified platform.

Without a simulator tool containing this compete feature set, the problem of uniformed interconnect design remains unsolved. It is for that reason that we have developed a flit-level network simulator called *NoCSim* to fill the void. *NoCSim* has all the above stated

features which are described in detail in Chapter 3.0. The development of *NoCSim* is the main contribution, and topic of this thesis.

The main goal of the thesis is to educate about the need for the simulator in the NoC domain, and then to describe the simulator's feature scope, describe its functionality and to then validate it where possible with results.

The thesis is organized in chapters with this general introduction to the problem and possible solutions being the first. Chapter two describes some of the background information and related work. Chapter three describes the feature set of *NoCSim*, and how those features can be used. The fourth chapter details the functional behaviour of *NoCSim* and discusses simulator performance and limitations. In chapter five possible simulator outputs will be discussed. Results validation will be done by comparing *NoCSim* results to results of Flexsim1.2. Finally in chapter six conclusions will be drawn and future work will be touched upon. The appendices contain the *NoCSim* source code and some documentation regarding how the code is organized.

2.0 Background and Related Work

2.1 Background Information

One of the most major problems that arise from the technology advancement is the non-scalability of global wire delays. Global wires carry signals across a chip, but these wires do not scale in length with the ITRS roadmap for technology scaling. While gate delays scale down with technology, global wire delays typically increase or remain constant by inserting repeaters. However, repeaters also have their inherent problems such as their need for the use of an even number of inverters, he many via cuts, and, above all, the additional silicon area and power consumed. It is estimated that non-repeated wires with practical constraints result in delays of about 120-130 clock cycles across the chip in the 50 nm technology node [40]. In ultra deep-sub micron processes, 80% or more of the delay of critical paths will be due to interconnect.

Another important problem associated with global wires is that such wires are typically implemented in top-level metal layers, with routing performed automatically in later stages of the design cycle. These wires end up having parametric capacitance and inductance that is difficult to predict before hand.

The goal of global synchronization of all IPs is therefore left unrealized due to impossibility in sending signals from one end of the chip to another within a single clock cycle. Instead of aiming for that unobtainable goal, an attractive option is to allow self-

synchronous IPs to communicate with one another through a network-centric architecture [40].

Existing on-chip interconnect architectures will give rise to other problems with scaling. The most commonly used on-chip interconnect architecture is the shared medium arbitrated bus. In such an interconnect, all communicating devices share the same transmission medium, usually a group of wires. The achievable operating frequency of the bus depends on the propagation delay in the interconnection wires. This propagation delay depends on the number of IP cores connected to the wires. Each core attached to the shared bus adds a parasitic capacitance, thus degrading performance with system growth [39]. For SoCs consisting of hundreds of IP blocks, this bus-based interconnect architectures will lead to propagation delays that exceed one clock cycle, therefore making it impossible for IPs to reliably communicate with each other.

To overcome these stated problems, it has been proposed the use of a network-centric approach to integrate IPs in complex SoCs [40]. Research groups have proposed interconnect solutions that use mesh, torus, fat-tree, and octagon network topologies [2,4,14,20,40]. In these models, IPs are connected to closely neighboured switches which are connected to neighbouring switches. Global signals which would span significant portions of a die in a more traditional bus-based architecture, now only have to span the distance separating switches. In this scenario, global wires will only consist of top level interconnects between switches. The specifics about such interconnect can be known at

early stages of the design process, enabling a better prediction of the electrical parameters of the interconnect, and overall system performance [40].

Several on-chip network proposals for SoC integration can be found in literature. One is *Sonic's Silicon Backplane* is a bus-bases architecture in which the IP blocks are connected to a shared bus through specialized interfaces called *agents* [49]. Each core communicates with an agent using the *Open Core Protocol (OCP)* [10]. Agents communicate with each other using *TDMA (Time Division-Multiplexed Access)* bus access schemes effectively decoupling the IP cores from the communication network. The basic interconnect architecture is still bus based and will hence suffer from performance degradation trends common for buses [40].

MIPS technologies has introduced an on-chip switch integrating IP blocks in a SoC[50]. The switch called *SoC-it* is intended to provide a high-performance link between a *MIPS* processor and multiple third party IP cores.

Kumar [28] and Dally [30] have proposed mesh-based interconnect architectures. These architectures consist of an m x n mesh of switches interconnecting IPs placed one at each switches. Each switch is thereby connected to four neighbouring switches and one IP block. The mesh topology is described in detail in Chapter 3.

In [2,3,4,39,40] the SoC group at UBC has described and interconnect architecture for a networked SoC, as well as the associated design of required switches, addressing mechanisms, and dealing with inter switch wire delay problems.

In SoC environment where die area is an issue, switches need to consume as little area as possible. In *wormhole switching* the packets are divided into fixed length flow control units (*flits*) and the input and output buffers need only to be able to store a few flits only. This feature ensures the buffer space requirement in the switches will be small relative to packet switching [31].

The performance of large scale SoCs will depend on the throughput of the network which depends on the *flow control mechanism* [31]. Flow control determines the allocation of channel and buffer resources to packets as it traverses its routed path. In switch based interconnect architectures, buffers are associated with physical channels and messages are buffered at the input and output of each physical channel and are commonly operated as *FIFO (First-In, First-Out)* queues. Therefore once a message occupies a buffer for a particular channel, no other message can access the physical channel, even if the message is blocked [9].

It is possible that the channel remains idle, while there are packets in the network waiting for it. This problem is a unique problem associated with an interconnection network using wormhole switching technique and impacts the SoCs overall performance substantially [9].

To solve this throughput degradation problem it has been propose to use the concept of virtual channels in the SoC environment [9,40].

The concept of virtual channels was introduced by Dally [9] and serves to decouple buffer resources from link transmission resources. This decoupling allows unblocked messages to pass blocked messages using transmission resources that would have otherwise been left idle. It has been shown that this increased utilization opportunity can lead to substantially higher throughput. The exact amount of performance increase depends on topology and many other network parameters.

The decoupling of buffer and link resources involves adding extra buffers on the input and output end of a link (see Figure 1). Flits stored in an output buffer do not own the link as well; rather they must compete with other flits being stored in the other output buffer of this link. This competition is done in the form of round robin so that each virtual channel receives an equal share of the link bandwidth. Once a flit is granted access to the link, it can traverse and must be stored in an available buffer at the input end. If an input buffer is not available, the flit would not have been granted use of the link.

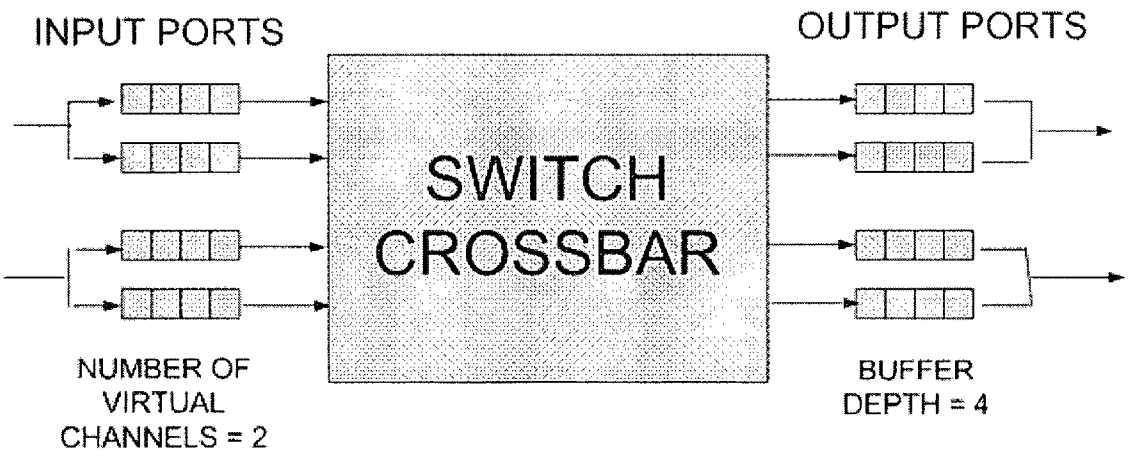

Figure 1: Example switch using two virtual channels

No message can occupy more than one virtual channel at a port's input or output port at a given time. Once a free virtual channel is populated by a header flit, the virtual channel is reserved for the use of only that message until the reservation is cleared by that message's tail flit.

2.2 Related Work

Other network simulation tools have been developed, both commercially and academically. As described below, their feature capabilities fall short of what is required for NoC simulation and analysis.

2.2.1 *FlexSim*

FlexSim1.2 is a simulator for flit-level simulation k-ary n-cube networks developed by the *SMART* Interconnects group at University of Southern California (USC) [27]. The

tool was developed with the purpose of studying the performance impact of various routing algorithms.

FlexSim has the ability to vary a number of network parameters such as virtual channels, traffic pattern, load, switch delays, message lengths, and buffer depths. A deadlock detection module is included that builds a dependency graph and located cycles. It also has the functionality to characterize the deadlocks in terms of frequency and size. Various routing adaptive routing algorithms are available as well as standard dimension ordered routing.

The tool collects statistics throughout on latency and throughput and produces standard outputs (throughput, average message latency).

The main restriction of *FlexSim* is its inability to simulate networks other than the mesh or torus. Another limitation is the size of each dimension along the cube is restricted to be a power of two, preventing irregular sizes such as a 50 x 50 network. While *FlexSim* can give users a good idea of the performance of adaptive routing algorithms in these two topologies, there are many more topologies for NoC designers to consider. This drawback is the main reason *FlexSim* by itself is not enough to give designers the information they require to make a completely informed decision about on-chip network interconnection.

2.2.2 *IRFlexsim0.5*

IRFlexSim0.5 is another flit-level simulator developed by the *SMART* Interconnects Groups at USC [26]. The tool is based on *FlexSim1.2* and shares most of its functionality. The main purpose of *IRFlexSim0.5* is to study the performance of various routing algorithms over networks with arbitrary topologies.

The parameters that could be varied with *FlexSim* (virtual channels, traffic pattern, delays, etc.) are still there. Static and adaptive routing algorithms can be chosen and routing tables can be input through external files. Deadlock detection is also present in this version.

The ability to create a regular k-ary n-cube topology is lost in this version, as are the topology specific routing algorithms. In its place, *IRFlexSim0.5* has the ability to create irregular topologies arbitrarily when given a specified number of switches and links. Other guidelines like the minimum and maximum degree of each switch and whether or not multiple links are allowed between links can also be set.

Topologies can also be taken from a user created input file. The input file contains the number of nodes and links in the network followed by a list of links in the form *<node>-<node>*. This manual process is limited to use only bi-directional links, but with enough manual effort, any topology can be input into the simulator. The major drawback of this functionality is that no matter what topology is manually input, the network is still treated as an arbitrary graph of nodes. This means that if a

multidimensional octagon was manually input, it would not be routed as one, but rather as an arbitrary group of nodes.

Another limitation of the topology creation is that the process is limited to direct networks. Intermediate switches in topologies such as trees cannot be created as each node in the network contains a traffic source and sink.

2.2.3 *NS version 2.0*

NS version 2.0 is an object-oriented discrete-event simulator for packet switched local and wide area networks [25]. The Network Research Group at the Lawrence Berkeley National Laboratory first developed *NS version 1.0*. Version 2 is now part of the *Virtual Inter Network Testbed* (*VINT*) project at USC.

NS version 2.0 is suitable for small scale simulations of queuing algorithms, congestion control protocols, and multicast analysis. It has the capabilities to implement network protocols such as *TCP* and *UDP*, produce application based traffic source behaviour, and use various queue management mechanisms and routing algorithms. It provides standard outputs that the USC tools also provided such as throughput, and latency.

NS version 2.0 does not inherently support wormhole switching however. Although some insight to NoC behaviour could be taken from using this tool's packet switching functionality, it is not sufficient to simulate wormhole networks under packet switching

conditions. Many added complexities and dependencies arise when wormhole switching is introduced, and to get accurate simulated results [32][33], a tool with wormhole switching capabilities is required. *NS version 2.0* would require a significant amount of modification to support wormhole switching and so falls short of fulfilling all requirements for a NoC simulator.

2.2.4 *OPNET*

OPNET (Optimized Network Simulation Tool) is a very powerful piece of commercially developed simulation software [36]. It provides a development environment for simulation and analysis of communication networks. The tool has capabilities for creating topologies of static or dynamic nodes to include satellites and wireless devices. Nodes are connected together by either packet streams of statistic wires to transfer packets or numerical signals. Logic control and behaviour are modeled using process models which are written in a language called *Proto-C*.

OPNET Modeler is an environment for network modelling and simulation. It provides the ability to create hierarchies of network objects such as nodes and links. This feature makes the engine scalable to large networks. Fully parallel discrete event simulation is possible. Graphical environments model the simulated networks if specified by the user. Standard output statistics are available, as are more specialized measurements.

Although *OPNET* is a very powerful and encompassing tool, its focus is on communication networks at a higher level than on-chip. Wormhole switched simulations may be possible after some degree of customization. Comparing network topologies would involve creating each network from scratch as no automatic network creation is provided. These drawbacks cause developers to be well versed in *OPNET's* modelling language in order to evaluate newly proposed topologies.

A tool specialized for NoC simulation would be more beneficial for designers since the ability to quickly and easily compare topologies of different regular types and sizes is absent with *OPNET*.

2.2.5 *Stuttgart Neural Network Simulator*

SNNS (Stuttgart Neural Network Simulator) is a software simulator for neural networks developed at *The Institute for Parallel and Distributed High Performance Systems (IPVR)* at the University of Stuttgart [41]. The goal of the *SNNS* project is to create an efficient and flexible simulation environment for research on and application of neural nets.

Since network on-chip interconnects borrow ideals from large scale parallel processing and neural networks are closely linked to parallel processing it is possible that a neural network simulator can shed some light on a network on chip interconnect performance. Neural networks do have the fundamental difference that wormhole switching is not

supported. It is for that reason that neural network simulators cannot be used for our purpose.

2.2.6 *The cnet Network Simulator (v2.0.9)*

The cnet Network Simulator (v2.0.9) was developed at The University of Western Australia [42]. *cnet* enables users to vary data-link later, network later, routing and transport layer networking protocols in networks consisting of point-to-point links and *IEEE 802.3 Ethernet* segments. This simulator is used mostly as an educational tool, rather than a design aid. As its focus is centered on protocol use at higher than physical layers of the network model, it fails to provide the requirements needed for an NoC simulator.

2.2.7 *QualNet Version 3.8*

QualNet is a software tool used to design and test communication networks, including ad-hoc wireless networks as well as other wireless and wired networks [43]. It supports real-time simulation of a large number of nodes (10-10000). *QualNet* also has a feature that enables users to view 3D graphical models of their networks. The tool is widely used in industry and academia. While many different protocols and routing schemes are available at many layers of the network model, the simulator fails to support wormhole switching at the physical layer. Since the use wormhole switching surfaces many unique communication problems, a tool that does not allow one to investigate wormhole switching performance is less than satisfactory.

2.2.8 *REAL 5.0 Network Simulator*

REAL is a network simulator originally intended for studying the dynamic behaviour of flow and congestion control schemes in packet-switched data networks [44]. The simulator takes as input a description of network topology, protocols, workload, and control parameters. It produces as output statistics such as the number of packets sent by each source of data, the queuing delay at each queuing point, and the number of dropped and retransmitted packets. Unfortunately, *REAL 5.0* does not support wormhole switching or physical switch parameters and so cannot be considered as a NoC design tool.

2.2.9 *MaRS Maryland Routing Simulator*

MaRS (Maryland Routing Simulator) is a discrete event simulation test bed for evaluating routing systems [45]. The physical network is somewhat limiting and it consists of link components and node components. A node component models the "physical" aspects of a store-and-forward entity and is characterized by parameters such as buffer space, processing speed, packet queuing discipline, and failure and repair distributions. A link component models a transmission channel between two nodes. A link component connecting node A and node B represents two one-way channels. Each one-way channel is modeled by a queue of packets. A link component is characterized by parameters such as bandwidth, propagation delay, and failure and repair distributions. By connecting link

components and node components, the user can specify a network of arbitrary topology. However communication is limited to packet switching and virtual channels are not supported.

2.2.10 Boppana, Chalasani, and Siegel: *Wormhole Network Simulator*

R.V. Boppana, S. Chalasani, and J. Siegel developed a network simulator which supports wormhole switching. The code is provided as is with no technical support [46]. The simulator allows for the selection of physical network parameters such as virtual channels, flit size and buffers per lane. Routing and traffic load can also be configured. The main limitation of this software is its failure to support topologies other than mesh and torus. This fact would prohibit designers from fully investigating all options and as a result this software falls short of the requirements.

2.2.11 *Simured Multicomputer Network Simulator*

Simured is a multi-computer network simulator that was developed at the University of Valencia [47]. This simulator supports only k-ary n-cube tori and meshes. Flow control is limited to wormhole switching. It has several routing functions: deterministic, adaptive, with dead-locks support. *Simured* also allows the user modify the number of virtual channels of the network. Notably, *Simured* has been referenced in the text *"Interconnection Networks"* of José Duato [31]. However, due to topology restrictions, this simulator does not meet the requirements.

3.0 *NoCSim*

NoCSim is a flit-level network-on-chip simulator developed in C++. The main purpose of the tool is to provide a common platform to study the performances of various network topologies. The tool allows for the simulation of wormhole switched direct and indirect networks of several NoC proposed topologies. All parameters that characterize a network have default values, but can also be user-defined.

The simulator was developed using Microsoft Visual Studio, and coded in *C++*. *C++* was chosen because the popularity of the language makes expanding or changing the program easier for future developers. The code is modular so that new topologies and routing or collision handling algorithms can be added without complications.

This chapter details the features and options available in *NoCSim*. All options are chosen in the main menu shown in Figure 2. Topology characterization is discussed first below, followed by switching and routing concepts. Descriptions of the parameters and options available with *NoCSim* follow. Finally, output options and statistics are discussed. Chapter 4 describes the simulation engine and Chapter 5 goes into more detail about output, illustrating results obtained from running simulations.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
CURRENT NETWORK/TRAFFIC PARAMETERS
1 - Butterfly Fat Tree
2 - Number of IPs: 256
3 - Buffer Depth (flits): 1
4 - Virtual Channels: 4
5 - Source Queue Length (messages): 100
6 - Simulation Duration (cycles): 1000000
7 - Reset stats at time: 2500
8 - Traffic Type: Uniform
9 - Load: 1
a - Message Length (flits): 16
b - Packets dropped when source queue overloads
c - Control flits not used
d - Message info not dumped
e - Port info not dumped
f - Adjacentcy list not dumped
g - Average Queue Length calculation is ON
h - Stat update interval: 2500
i - Average Active Messages calculation is OFF
j - Header Collisions handled by: Port Order
k - 1 interation per cycle
l - Not using trace file

Derived Parameters:
Switches: 120
Nodes: 376
Levels: 4

COMMANDS: Run - r Setup Sweep - s Quit - q
Enter number to change parameter, or choose a command to run simulation:
```

Figure 2: *NoCSim* Options Menu with Default Settings

3.1 Network Topologies

Networks topologies have been previously classified as they have been widely studied in parallel processing [31]. Since so many varieties of topology exist, these classifications instantly give a better understanding of topology characteristics when considering a new topology. *NoCSim* offers simulation of a wide variety of topologies; some of the classifications of said topologies are described below:

- Shared-Medium Networks
- Direct Networks
- Indirect Networks

Available topologies in *NoCSim* are *k-ary n-trees* (fat trees), *k-ary n-cubes*, *butterfly fat trees (BFT)*, and *octagon*. For *k-ary n-cubes*, the number of dimensions, and then number of IPs along each dimension are accepted thus making it possible to simulate a *2 x 5 x 7* network if so desired. Also, whether or not wraparound links are used is also to be specified. For *k-ary n-trees*, *k* and *n* must be input creating a network of *k*k* switches at n levels. Details and formal definitions of *k-ary n-trees* and *k-ary n-cubes* follow in the subsections below.

All networks require the appropriate number of IPs to be set as well. This value can be arbitrary as the simulator will create the smallest topology possible that will incorporate this number of IPs. E.g., if a 100 IP BFT is selected, a 256 IP BFT will be created with 156 dormant IPs. Dormant IPs do not impact network statistics such as throughput.

3.1.1 Shared-Medium Networks

Shared-medium networks are the least complex interconnect structure. They are made up of one transmission medium which is shared between all communicating devices [31]. A major drawback of these networks is that only one device is permitted to use the medium at any given time. As the number of devices sharing the medium increases, the medium becomes a bottleneck as many collisions arise and devices are forced to wait for their turn to use the medium. To handle these collisions, a hardware arbiter is required to deal with requests.

An inherent benefit of shared-medium networks is their ability to support broadcasting. Since all devices connected to the medium can monitor the network activities one device can reach all other devices with one effect by simply including the addresses of all destination devices in the message header, or using a broadcast flag.

3.1.1.1 Bus

Bus systems fall under this category and are the first suggested topology for SoC environments. Buses are shared medium networks where the medium is one or more wires that stretch across a series of communication devices. Examples of on-chip bus networks are *AMBA* [5], *CORECONNECT* [6], and *WISHBONE* [7].

As stated above, buses have been used for smaller SoCs, but as we advance to the deep sub-micron era, buses fail to provide adequate performance for any application. One of the major drawbacks of bus systems is the transmission medium (bus line) becomes a bottleneck as IPs requesting its use become backed-up with unsent packets. The problem only gets worse when additional IPs are added.

Buses also have drawbacks in the form of propagation delay and parasitic capacitance. Intrinsic parasitic resistance and capacitance from the line and added IPs increase to extremely high levels causing propagation delay to be unacceptably high. High

propagation delays lead to longer clock cycles, higher latency, and lower throughput. A theoretical example of a bus topology is shown in Figure 3.

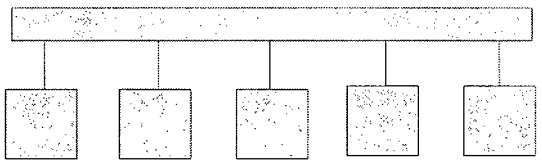

Figure 3: Simple Bus Structure

3.1.2 Direct Networks

Direct networks topologies consist of a set of nodes which are connected to each other by channels. Each node contains an independent functional unit, or IP in the NoC domain, as well as a router component that is common amongst all nodes. Because of these characteristics, direct networks are also called router based networks, or point-to-point networks. The routers are connected to neighbouring routers via bi or unidirectional links. Each router also has internal channels proving paths to connect a node's external channels together. Direct networks have been generally described using graphs containing nodes N, and channels C, $G(N,C)$ [31].

An ideal direct network would have all nodes connected to all other nodes directly without intermediate nodes in the way. A network like this is called fully connected where each router must contain N external links where N is the number of nodes. For any substantial N, the cost of such a network is prohibitive due to the overwhelming wiring complexity. Therefore even though performance may suffer when less links are

used, such sacrifices are necessary to create a scalable and cost efficient network. Many different topologies have been suggested that attempt to balance this trade-off of cost and performance.

Direct networks have the inherent feature that they scale well to large number of IPs. Each new IP added brings with it a new router component, thus the overall bandwidth of the network increases with the number of IP blocks. This is a key advantage when designing a network which is to contain a large number of IP blocks.

As messages traverse nodes to reach their destinations, the routing algorithm used by each node determines the path it will take. Efficient routing is a critical factor of the performance of the interconnection network. Routing schemes can be specific to topologies but share the same fundamentals such as keeping travel paths short, and deadlock avoidance. Routing and deadlock issues are discussed in depth in Sections 3.4 and 3.5, respectfully.

3.1.2.1 *K-ary N-cubes*

K-ary n-cube topologies fall under the direct network category. Each IP in the topology contains a router component that connects it to neighbouring routers their IPs [20][28]. These networks contain K nodes along N dimensions for a total of K^N nodes. Below is the formal definition of a *k-ary n-cube*:

A *k-ary N-cube* has k nodes along each dimension n. Each node X is identified by n coordinates $(x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$. Two nodes X and Y are neighbours if and only if $y_i = x_i$ for all i, $0 \leq i \leq n-1$ except one, j, where $y_j = (x_j \pm 1) \bmod k$.

Rings fall under the *k-ary n-cube* definition with $n=1$. Other examples are shown in Figures 4-6.

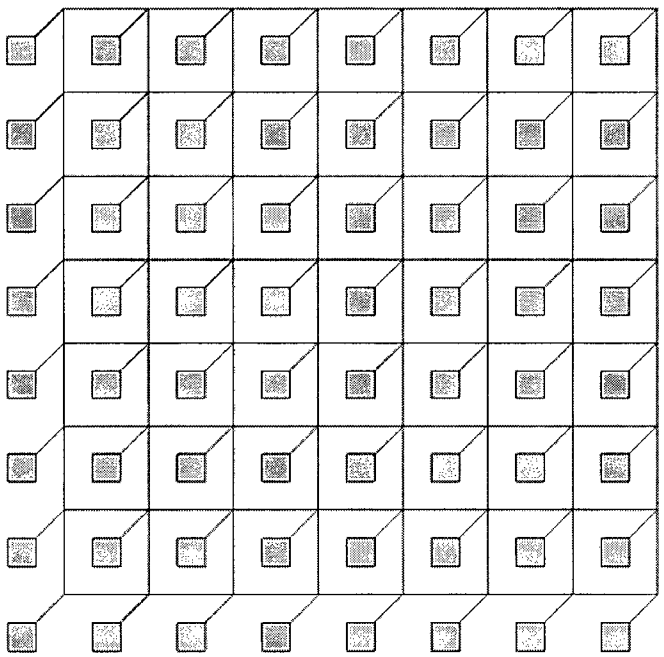

Figure 4: A *8-ary 2-cube* without wraparound links, also called a *mesh* topology.

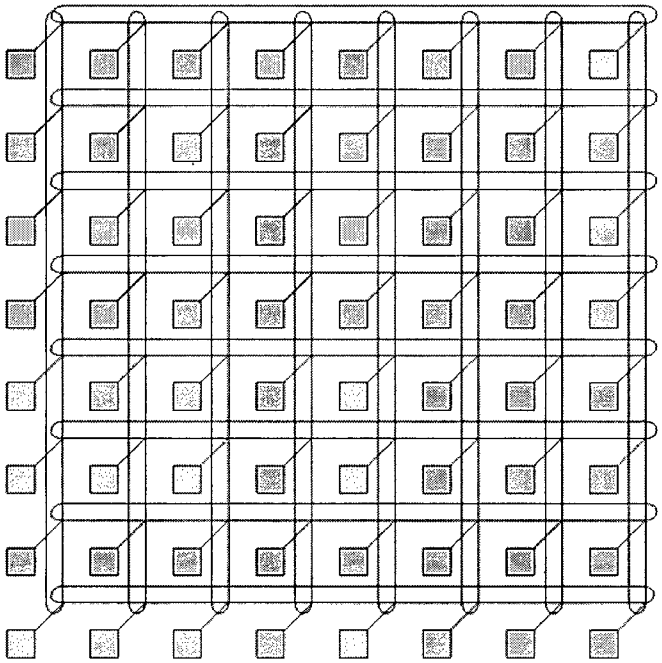

Figure 5: A *8-ary 2-cube* with wraparound links, also called a *torus* topology.

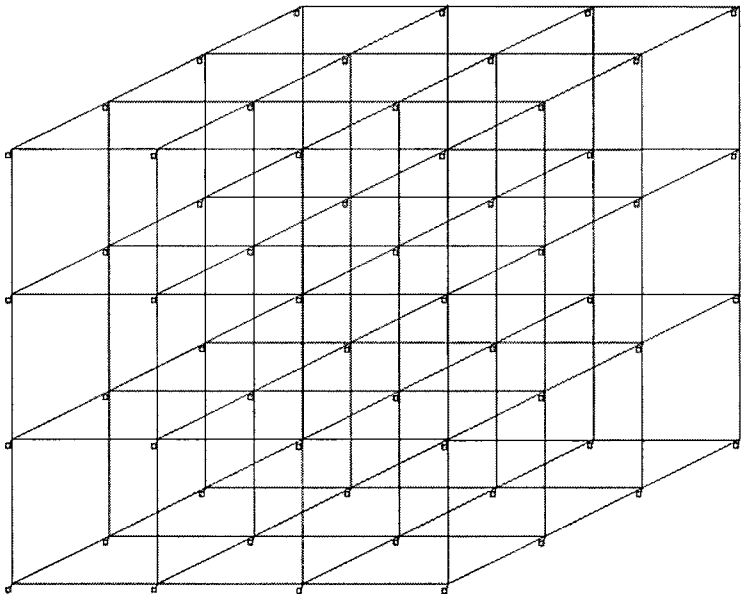

Figure 6: A *3-ary 3-cube*.

A two dimensional cube as defined above is also referred to as a *torus*. A *torus* is a conventional mesh with the addition of wraparound links for each dimension. The formal

definition of a mesh is the same as the *k-ary n-cube* with the simplification that two nodes X and Y are neighbours if and only if $y_i = x_i$ for *all* i, $0 \leq i \leq n-1$, except one, j, where $y_j = x_j \pm 1$. Leaving out the modulus k removes the wraparound links making the mesh irregular and without symmetry.

The folded torus topology is graphically equivalent to a torus topology, the only difference being the way in which the IPs are laid out [14]. IPs are 'folded' over and interleaved on a row by row basis to dispose of a long and power inefficient wrap-around link. As a result, short wire lengths increase, but the need for a long 'wrap-around' link is alleviated reducing the overall energy consumption for flit transmissions. The difference is illustrated below in Figures 7 and 8.

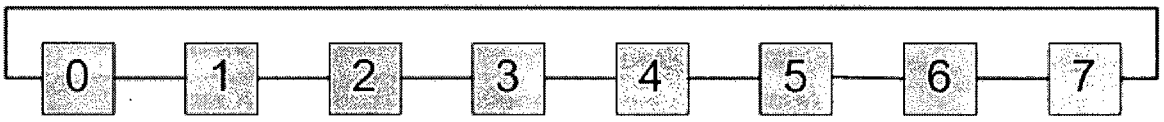

Figure 7: A conventional 8 IP, *8-ary 1-cube* with wraparound link, also called a *ring*, or *torus*.

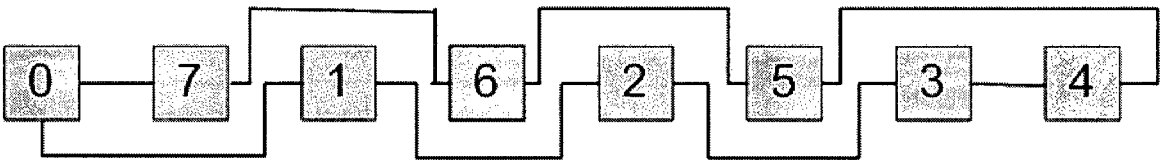

Figure 8: A *'folded' torus* of 8 IPs. It is still an *8-ary 1-cube* with wraparound link. Each node keeps the same neighbouring nodes, but the node order is changed to remove the long wraparound link.

3.1.2.2 *Octagon*

Another direct network topology that has been proposed for NoC purposes is octagon [19]. At the lowest level, the topology consists of 8 nodes and 12 bi-directional links connecting each node. Each node is connected with the node directly across from it, its clockwise neighbour, and its counter clockwise neighbour as shown in Figure 9.

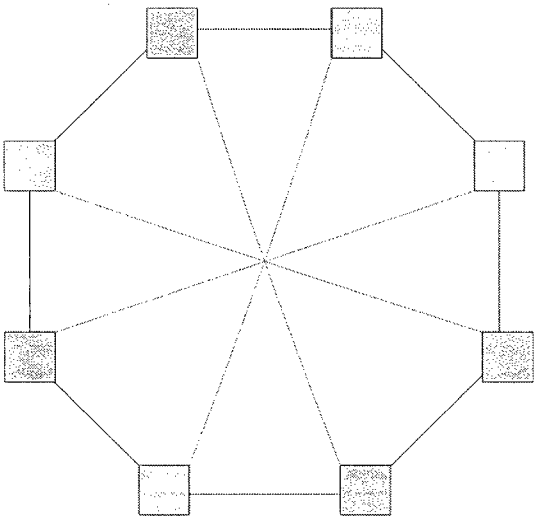

Figure 9: An 8 node *octagon.*

For networks with greater than 8 IPs, the octagon scales by extending into multidimensional space. A 64 IP example is shown in Figure 10. If each node is indexed with the ordered pair $(i, j) i, j \in [0,7]$. For each $i = I \in [0,7]$, an *octagon* is constructed using nodes $\{(I, j), j \in [0,7]\}$. These *octagons* are then connected together by linking corresponding I nodes according to the octagon configuration. Thus, each node could be thought to be in two different octagons. The first consisting of nodes $\{(I, j), j \in [0,7]\}$, and the second consisting of nodes $\{(i, J), i \in [0,7]\}$. This pattern can be continued any

number of dimensions, giving a maximum number of IPs of $N = 8^d$, where d is the number of dimensions.

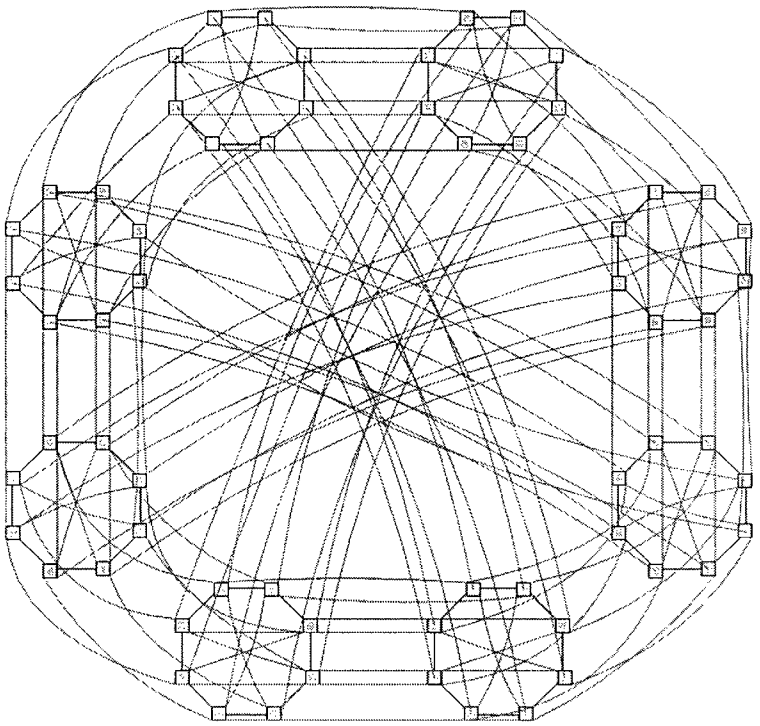

Figure 10: A 2-dimensional 64 IP *octagon* topology

This strategy maintains a low worst case hop count between IPs, but comes at the cost of high wiring complexity. At the lowest level, packets have to travel at most two hops to reach their destination. That number only increases by two for each added dimension keeping the connectivity close to full even as N increases to thousands. This connectivity is extremely costly as the wiring demands increase exponentially. For example, for a 512 IP three dimensional octagon, 2304 links are required, while for a 64 IP two-dimensional octagon, only 192 are required.

3.1.3 Indirect Networks

Indirect networks are similar to direct networks but differ in that instead of IPs being connected directly, they are connected though intermediate switches. These switches act like the router components of direct networks. Each contains a number of ports which contain input and output links. Because of this distinction, indirect networks have been called switch-based networks, while direct networks are called router-based networks.

The topology of an indirect network is determined by how the switches are connected together through links. Like direct networks, indirect network topologies can also be modelled by a graph $G(N,C)$, where N is the set of switches and C is the set of links between the switches [31].

In regular indirect networks, the switches are usually identical and are organized as a set of stages. Each stage is only connected to the previous and next stage using regular connection patterns. Input/output stages are connected to functional nodes as well as to another stage in the network. These networks are referred to as *multistage interconnection networks* (*MIN*) and have different properties depending on the number of stages and how those stages are arranged.

3.1.3.1 K-ary n-trees

K-ary n-trees, or *fat-trees* have been proposed for on chip network architectures [15][18]. *Fat trees* are in the indirect network category because they have IPs connected to

switches at the leaves of a tree. Those leaves are connected to other switches further up the tree. The formal definition is shown below:

Definition 1: *A fat-tree is a collection of vertices connected by edges and is defined recursively as follows:*

- *A single vertex by itself is a fat-tree. This vertex is also the root of the fat-tree.*
- *If* $v_1, v_2, \ldots, v_i$ *are vertices and* $T_1, T_2, \ldots T_j$ *are fat-trees, with* $r_1, r_2, \ldots r_k$ *as roots (j and k need not be equal), a new fat-tree is built by connecting with edges, in any manner, the vertices* $v_1, v_2, \ldots, v_i$ *to the roots* $r_1, r_2, \ldots, r_k$. *The roots of the new fat-tree are* $v_1, v_2, \ldots, v_i$.

The above definition is very general and covers regular trees, and fat-trees with variable sized switches and multiple connections between vertices and irregular constructions.

K-ary n-trees are a specific class of *fat-trees* where all switches are identical, and construction is regular. The formal definition is shown below:

Definition 2: *A k-ary n-tree is composed of two types of vertices:* $N = k^n$ *processing nodes and* nk^{n-1}, $k \times k$ *communication switches. Each nodes is an n-tuple* $\{0,1,\ldots,k-1\}^n$, *while each switch is defined as an ordered pair (w,l), where* $w \in \{0,1,\ldots,k-1\}^{n-1}$ *and* $l \in \{0,1,\ldots,n-1\}$.

- *Two Switches* $(w_0, w_1, \ldots, w_{n-2}, l)$ *and* $(w'_0, w'_1, \ldots, w'_{n-2}, l')$ *are connected by an edge if an only if* $l' = l + 1$ *and* $w_i = w'_i$ *for all* $i \neq l$. *The edge is labelled with* w'_l *on the level* l *vertex and with* w_l *on the level* l' *vertex.*

- *There is an edge between the switch* $(w_0, w_1, \ldots, w_{n-2}, n-1)$ *and the processing node* $p_0, p_1, \ldots, p_{n-1}$ *if and only if* $w_i = p_i$ *for all* $i \in \{0, 1, \ldots, n-2\}$. *This edge is labelled with* p_{n-1} *on the level n-1 switch.*

Examples of *k-ary n-trees* are shown below in Figures 11 and 12.

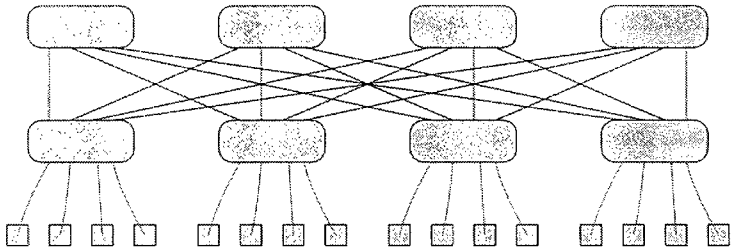

Figure 11: A 16 IP *4-ary 2-tree*

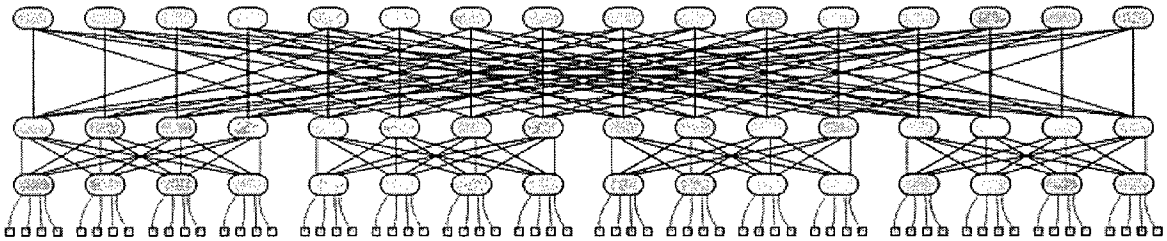

Figure 12: A 64 IP *4-ary 3-tree*

3.1.3.2 *Butterfly Fat-Tree*

The *butterfly fat-tree (BFT)* is another indirect network topology that has been proposed for NoC applications [3][16][17]. The *BFT* is a variation of *fat-tree* where each switch is imbalanced in that is has four child ports which send and accept traffic from further down the tree, and two parent ports that send and accept traffic towards the root of the tree. These ports are indexed as: $parent_0$, $parent_1$, $child_0$, $child_1$, $child_2$, and $child_3$. For referencing, these switches are addressed *S(l,a),* where *l* is the level, and a is the index along that level. The *N* IPs at the lowest level are connected to $N/4$ switches at the first level such that processor *P(0,a)* is connected to $child_{a \bmod 4}$ of switch $S(1,\lfloor a/4 \rfloor)$. The number of levels in a *BFT* is $\log_4 N$, and each level contains $N/2^{l+1}$ switches. These switches are connected according to the switch's address. $Parent_0$ of *S(l,a)* is connected to $child_i$ of $S(l+1,\lfloor a/2^{l+1} \rfloor \cdot a \bmod 2^l$ and $parent_1$ of *S(1,a)* is connected to $child_i$ of $S(l+1,\lfloor a/2^{l+1} \rfloor \cdot 2^l + (a+2^{l-1}) \bmod 2^l$, where $i = \lfloor a \bmod 2^{l+1} / 2^{l-1} \rfloor$. The number of switches per level is reduced by a factor of 2 as the level increases. From this the total number of switches S_T can be calculated as:

$$S_T = N/4 + (1/2)N/4 + (1/2)^2 N/4 + \ldots + (1/2)^L N/4$$

$$= N/4 \frac{(1-(1/2)^L}{(1-1/2)}$$

$$\lim_{L\to\infty} N/4 \frac{(1-(1/2)^L}{(1-1/2)} = N/2$$

Examples of *BFTs* are shown below in Figures 13 and 14.

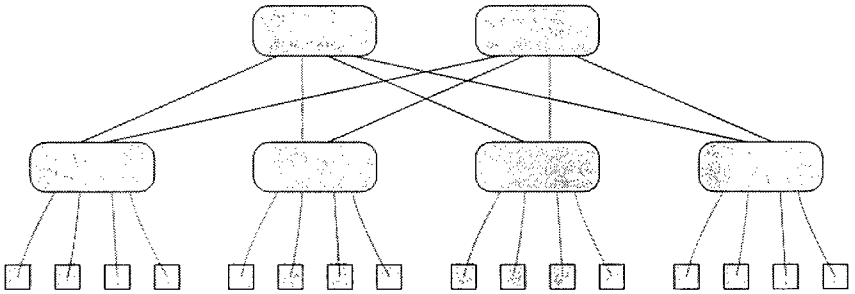

Figure 13: A 2 level, 16 IP *Butterfly Fat Tree*

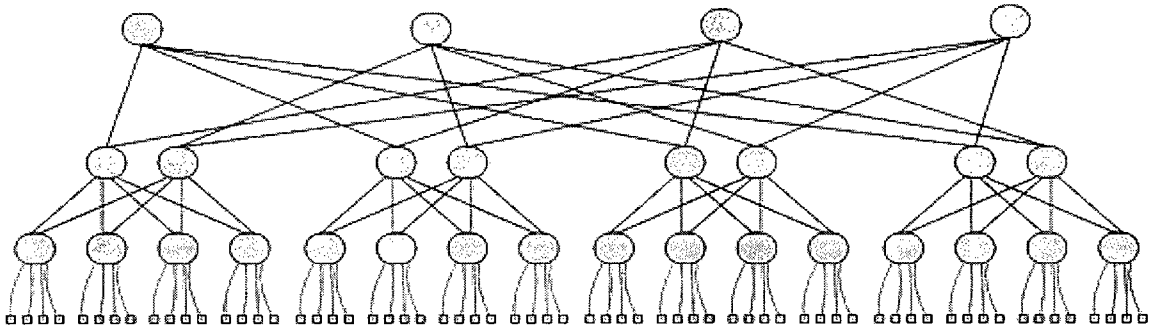

Figure 14: A 3 level, 64 IP *Butterfly Fat Tree*

3.2 Switching

A switching mechanism determines how and when router input channels are connected to output channels that have been selected by the routing algorithm. It determines how network resources are allocated for message transmission. Popular switching mechanisms are *circuit switching*, *packet switching*, and *wormhole switching* [31].

In *circuit switching*, all channels required by a message are requested and reserved prior to the message being transmitted. Thus, a dedicated path is established and the transmission encounters no contention once the path is established. The establishment of the path can be delayed by other messages reserving resources beforehand.

In *packet switching*, also called store-and-forward switching, messages are divided into packets before being sent though the network. Packets advance into resources as soon as they become available. Buffer space at each switch is required for packet switching as packets are held in a buffer until the next channel becomes available.

Wormhole switching also involves breaking a message into packets at the source like packet switching [32][33]. The packets are then further divided into flow control units called *flits*. The first *flit*, or header, contains routing and status information and advances into free resources like *packet switching*. *Flits* trailing the header *flit* follow in a pipeline fashion. If at anytime the header *flit* cannot advance because its desired input or output channel is occupied, the header becomes blocked and the trailing *flits* are forced to wait as well until advancing is again possible. Buffer space at each switch, but each buffer only needs to be large enough to hold one flit. Packets become spread out over many switches, so the buffer demands at each switch are a fraction of what they are in packet switching where each buffer must be large enough to hold an entire packet. Because of this actuality, *wormhole switching* is popular in the SoC domain where there is an aim to minimize switch hardware overhead.

Since only wormhole switched networks have been proposed for large scale on-chip networks, *NoCSim* only supports *wormhole switching.*

3.3 Virtual Channels

The use of virtual channels is generally accepted to be beneficial for NoCs, so *NoCSim* supports the use of virtual channels by allowing the user to select the number of virtual channels per input/output channel. All channels must have the same number of virtual channels.

3.4 Routing Options

How incoming packets are forwarded to outgoing channels depending on the active routing algorithm. Since routing depending greatly on the network topology [31], routing algorithms for each available topology are discussed in turn below.

3.4.1 *K-ary N-cube* Routing

NoCSim supports the most common simply routing algorithm used for *k-ary n-cubes*, *Dimension Ordered Routing* (*DOR*). It is a distributed scheme meaning routing decisions are made locally at each node by reading header tags. It is also a non-adaptive or oblivious algorithm, meaning messages cannot be dynamically rerouted to avoid congestion. Each source-destination pair will have the same path each time.

DOR first indexes the dimensions in the network so that a set order is established. When a header arrives at router, it compares the destination address $D\ (d_{n-1}, d_{n-2}, ..., d_1, d_0)$, with the node's address $(x_{n-1}, x_{n-2}, ..., x_1, x_0)$, where n is number of dimensions and x_i is the coordinate in dimension i. If the coordinates in the first dimension, $i = 0$, do not match the header is routed in the direction of the destination in that dimension. Once the header's destination and current node address match in the first dimension, the routers then compare the coordinates of the next dimension, $i=1$. This process is continued until the destination node is reached. An illustration showing a possible source-destination pair for a 64 IP *mesh* is shown in Figure 15.

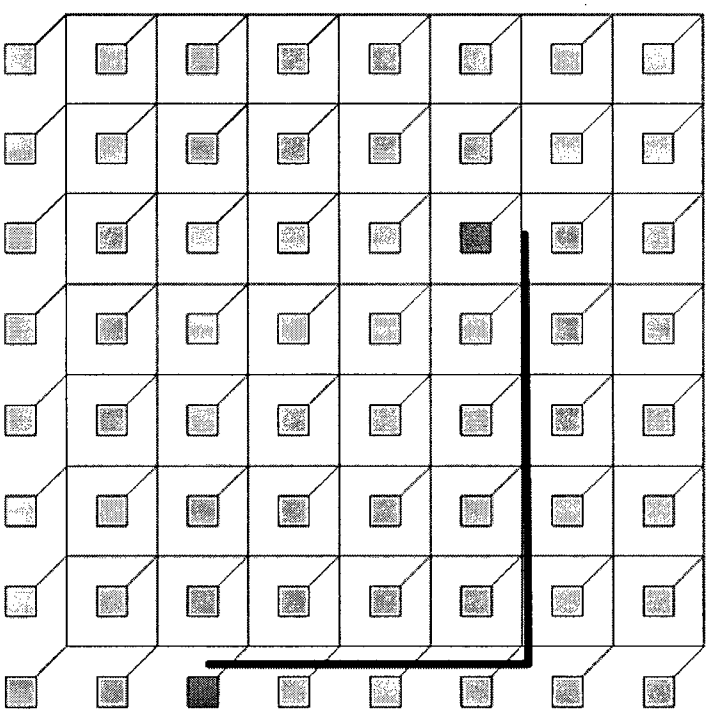

Figure 15: *Dimension Ordered Routing* in a 64 IP *mesh*

Many adaptive routing algorithms have been suggested for *k-ary n-cubes*. These may involve throttling, or avoiding congestion by changing dimensions additional times [35]. While these algorithms have some performance improvement in sub-saturation conditions, they also introduce added routing complexity and overhead due to added logic. For this reason, only non-adaptive algorithms are currently supported by *NoCSim*, however the modular nature of *NoCSim* allows for new routing algorithms, adaptive or non-, with relative ease.

3.4.2 *Octagon* Routing

NoCSim supports a simple, oblivious routing scheme for the *octagon*. It is a simple shortest-path algorithm where nodes determine the appropriate output port by checking the relative address of the destination, see Figure 16. A break down of the possible cases is shown below in Table 1:

Relative Address	Destination Port
0	Route to local IP port
1 or 2	Route clockwise
6 or 7	Route counter-clockwise
3, 4, or 5	Route across

Table 1: Simple shortest path *octagon* routing

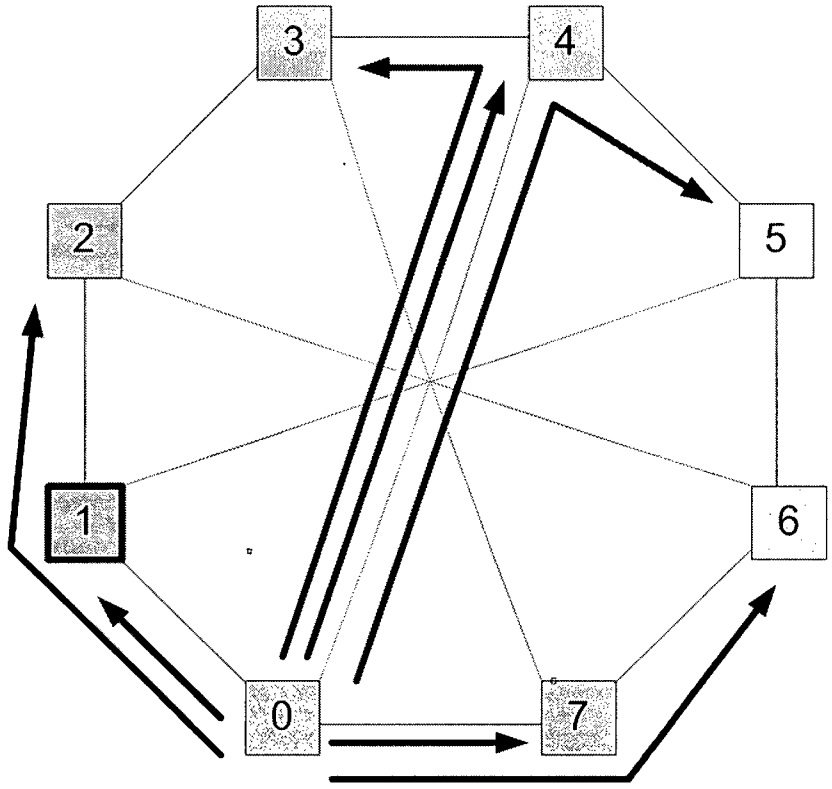

Figure 16: Shortest Path Routing in *octagon*

When dealing with multidimensional *octagons*, this routing scheme can be applied to each dimension in order starting with the highest [19].

3.4.3 *K-ary N-tree* Routing

Routing in trees almost always comes in the form of turnaround routing. In this form of routing, packets ascend the tree until they find their least common ancestor shared by the source and destination nodes. Once reached, the package is 'turned around' and begins its descent to the destination as shown in Figure 17. *NoCSim* supports turnaround routing.

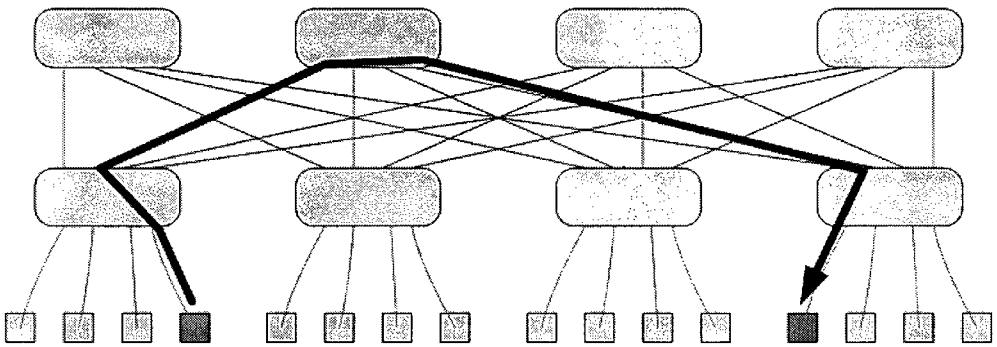

Figure 17: A possible path for a source/destination pair using *turnaround routing* in a 16 IP *fat-tree* topology.

When a header arrives at a switch child port that is not its least common ancestor, that header is routed out through one of the k parent ports. The choice of port is arbitrary as long as the port has an empty channel available; therefore redundancy exists during the ascent of the tree. After the turnaround however, there is only one unique path the header must be routed through to reach its destination, and the redundancy is no more [31].

3.4.4 *Butterfly Fat Tree* Routing

BFTs use the same turnaround routing scheme that is described above in the *k-ary n-tree* section.

3.5 Deadlock, Livelock, and Starvation

Deadlock, livelock and starvation are critical issues that need to be addressed in any network due to their catastrophic consequences [31].

A deadlock is a situation where all packets in the network cannot advance to their destination because the resources requested by them are full. All packets involved in the deadlock are blocked indefinitely. An example of a deadlock in a 4 node *ring* is shown in Figure 18. The buffers of each node are completely full with flits waiting for resources at the next node to become free.

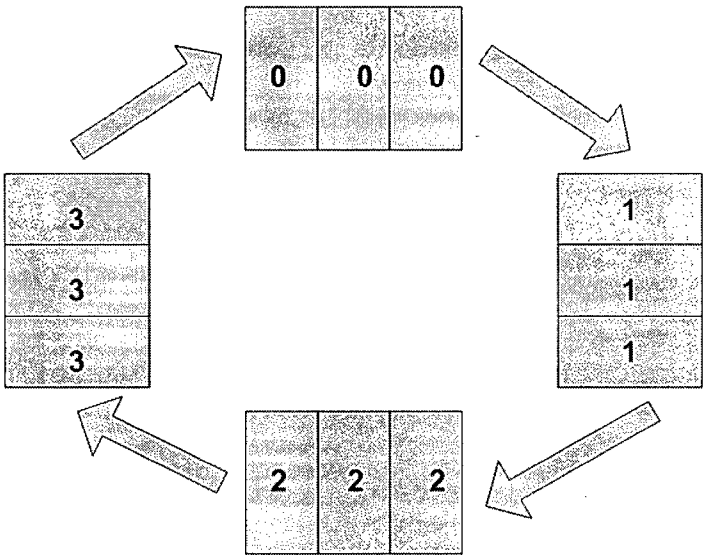

Figure 18: Deadlock in a 4 node *ring*.

Livelock is a similar situation where packets can never reach there destinations, the difference being packages may be moving around the network unblocked in a livelock situation. It can only occur if packets are allowed to follow non-minimal paths however, so since all networks in *NoCSim* are minimal path routed, livelock is not an issue [31]. If adaptive routing schemes are implemented, livelock avoidance strategies must be implemented as well.

Starvation occurs when a packet stops permanently when the resources it is requesting are consistently being granted to other contending packets. It can be avoided by using a fair contention resolution scheme when allocating resources. *NoCSim* allows the user to choose between several fair contention resolution schemes that prevent starvation. Contention resolution is described in detail in Section 3.7.

Methods of handling deadlock include deadlock avoidance, and deadlock detection / recovery. The later technique is only effective when deadlocks are rare because they carry the assumption that the networks can recovery from deadlocks faster than they arise. These techniques also require added hardware and logic at the switches. *DISHA* [13] is a recovery technique that requires each node to have a central "floating" buffer. Since hardware at the switches is to be minimized and deadlocks may occur at rapid rate, only deadlock avoidance schemes were implemented in *NoCSim*.

A necessary and sufficient condition for deadlock-free non-adaptive routing is the absence of cycles in a resource dependency graph [12]. Therefore, topologies that do not have cycles will never experience deadlock. Tree-based topologies like *k-ary n-trees* and *butterfly fat trees* have no cycles when turnaround routing is used. The same is true for *k-ary n-cube* topologies when wraparound links are not used (*mesh*) and *DOR* is used.

For topologies that do contain cycles, further effort is required to break the cycles to avoid deadlock situations. The *k-ary n-cube* with wraparound links (*torus*) and *octagon* topologies do contain cycles even when *DOR* is used.

Dally and Seitz [11] proposed a way to break the cycles in *k-ary n-cube* networks with the use of virtual channels and restrictive routing. To remove the cycles, the virtual channels are first split into two groups, upper and lower. When packets are injected into the network, they restricted to upper virtual channels only. If at any time a packet traverses a predetermined link in each cycle (e.g. the wraparound link) the packet's restriction changes from upper to lower. Each time the packet changes dimension it must be restricted to the upper virtual channels. This restriction changes the cycle nature of the dependency graph into a more spiral-like shape.

To illustrate this process, consider a four node *(N0-N3)* ring example in which the nodes are connected with unidirectional links. This simple topology and its resource dependency graph are shown in Figures 19 and 20.

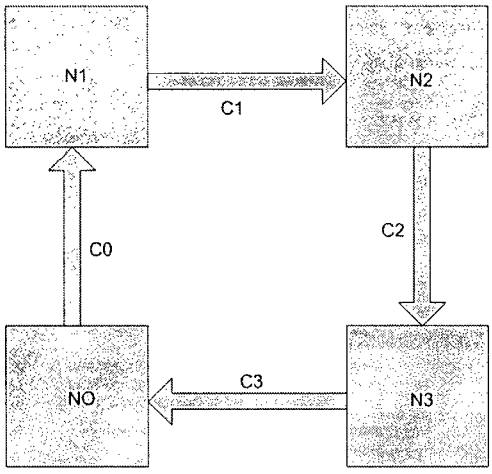

Figure 19: A 4 node unidirectional *ring.*

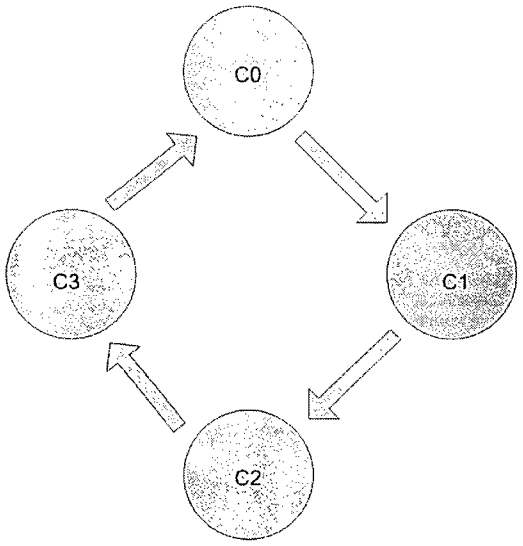

Figure 20: Corresponding dependency graph.

While the channels *(C0-C3)* remain undivided and routing is not restricted, the dependency graph contains a cycle. This cycle can be the source of deadlocks. However, if virtual channels are used to divide each channel into and upper channel *(Ciu i=0..3)* and a lower channel *(Cil i=0..3)* and routing is restricted as described above. Specifically, packets injected are restricted to using upper channels. If packets reach *N0* and are not at their destination however, they are then restricted to using lower channels. The divided *ring* and its dependency graph are shown below in Figures 21 and 22.

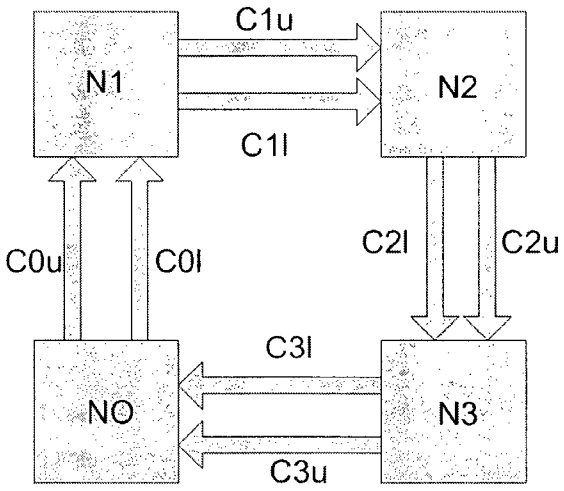

Figure 21: A 4 node *ring* with virtual channels.

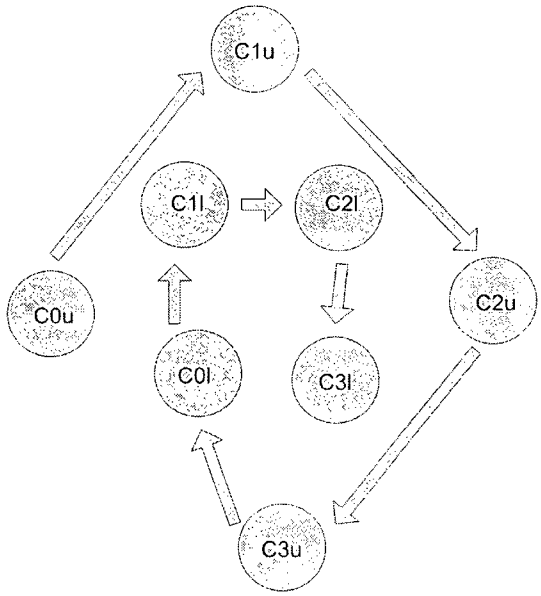

Figure 22: Corresponding dependency graph.

This restrictive routing and virtual channel use can be implemented to remove cycles from any network. It comes with the down side that the resources of the lower virtual channels remain idle when the wraparound link is not used. It is for that reason that mesh networks outperform *torus* networks under certain conditions such as highly localized

traffic loads [14]. Another notable characteristic of networks using this technique is packets must now keep track of which virtual channel they have come from when being routed. Without the restriction, packets are memoryless.

The cycles in the *octagon* could also be removed using this fashion. Since the *octagon* is so connected however, simply restricting routing to prohibit the use of a predetermined link in each level of *octagon* to physically remove the cycle gives the same effect without a very noticeable drop in performance. By doing this, the performance degradation under highly localized traffic is not an issue as virtual channels are not restricted.

3.6 Switch Model

The switch model used by *NoCSim* is shown below in block diagram form, Figure 23. Several parameters are available to customize network switches.

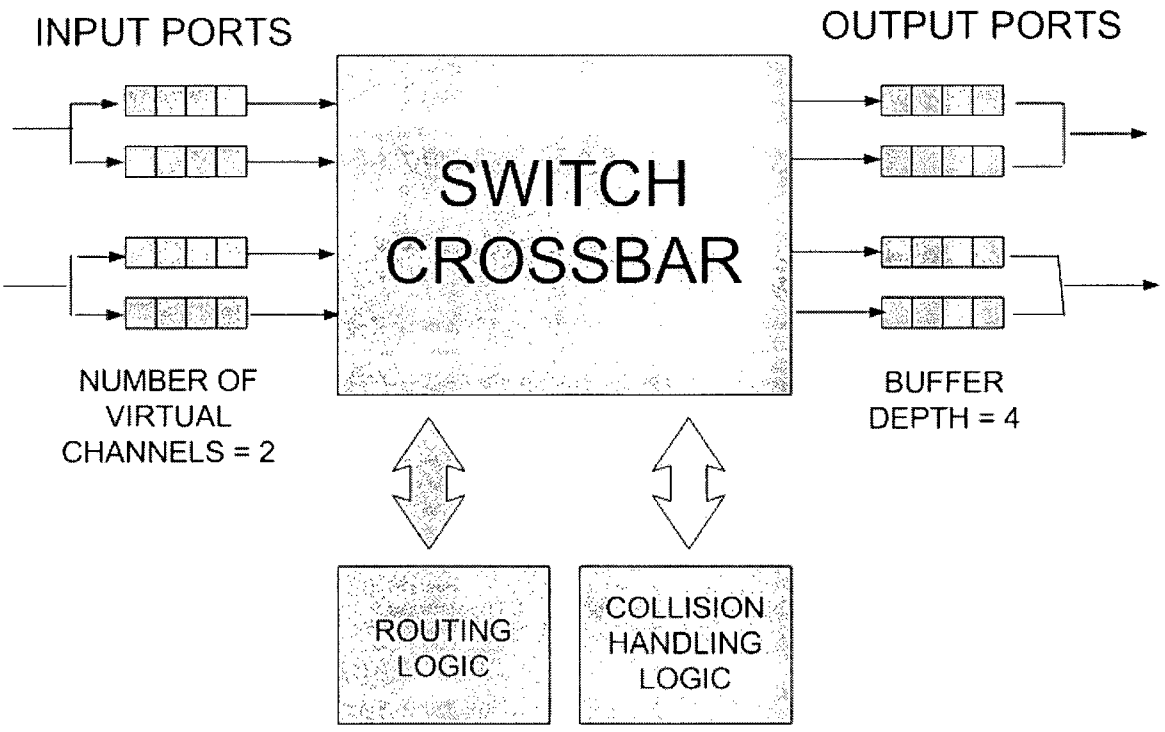

Figure 23: Example Generalized Switch Block Diagram.

The number of input/output ports is predetermined by the topology, although different topologies have different numbers of ports per switch.

For all topologies but torus, the number of virtual channels can be set to any positive number. *Torus* has the additional restriction that it must have at least two virtual channels because of the restrictive routing involved in preventing deadlock in that topology. The number of virtual channels describes how many buffers are available per link at both the input and output end. The number of buffers must be the same at both ends.

The depth of each buffer can be specified as well. This value determines how many flits can fit into each buffer.

The crossbar of each switch is assumed to be fully connected. In other words, every combination of input port to output port is valid.

How flits pass through the crossbar to output ports is determined by the routing and collision handling logic. Data and tail flits simply follow the path of their respective headers, but header flits require routing. As previously mentioned, routing is topology specific and determines the destination output port for each header. Once all headers have been routed, collision handling logic detects and resolves and contention for output ports. Finally, header movements are processed and the appropriate headers move to the determined output buffers. Details of these processes are described below in Section 4.1.

3.7 Collision Handling

To deal with the collisions that arise between contending headers, a collision handling scheme, or select function, is required to select the order of which resources will be granted. *NoCSim* has several options in this respect and they are discussed in next subsection:

3.7.1 *Port Ordered Collision Handling*

In this scheme, the indexes given to ports in the network creation process are used to determine which header's requests are recognized first. In other words, the mechanism will first check all headers in *port 0*, followed by *port 1, 2*, etc. This order does not change throughout the simulation. This scheme is the easiest to implement, however it is prone to starvation, since in saturated traffic a header at a higher numbered port could be forced to wait indefinitely.

3.7.2 *Round Robin Collision Handling*

The *round robin scheme* is similar to the *port order scheme* in that a mechanism checks headers in order of their port index. The difference is that from cycle to cycle, the starting point of said mechanism changes to the next port. When the last port of a switch is reached, the mechanism returns to *port 0*, thus providing wraparound.

3.7.3 *Oldest Goes First Collision Handling*

This scheme involves the collision handler checking the network injection times of contending headers and resolving their requests in the order of oldest goes first. Doing this ensures that starvation will not occur as headers will at some time become the oldest at a particular switch. Generally, this scheme will reduce a network's worst-case latency but to be physically realized would involve headers carrying injection time information.

3.7.4 *Priority Based Collision Handling*

NoCSim has the capabilities of implementing a *priority based collision handling scheme*. In the supported scheme, four different levels of priority of message are possible *(0-3)*. Messages are given this priority upon injection and the ratio of a given priority level to another can be specified as well (e.g. *10% priority 0, 50% priority 1, 20% priority 2, 20% priority 3*). When collisions occur using this scheme, headers with higher priority are given the first available resources, then the next highest, etc. Collisions of headers with the same priority are handled according to *round robin*.

High priority messages blocked by lower level messages temporally transfer their priority to the lower priority message to encourage it to get out of the way. This priority inheritance was proposed by R. Rajkumar in [37] and is shown to further improve the average latency values of high-priority messages.

3.8 Traffic Generation

NoCSim has many options for traffic generation, the default being *Poisson distributed uniform random traffic*. In this class, messages arrive at sources according to *Poisson* distribution are given randomly generated destinations that are all equally as likely. The only restriction being the source and destination cannot be equal. *Uniform traffic* is a long used benchmark for network analysis, but has been proven to be inaccurate when modelling real life networks [31].

Destinations can also be selected by a *bit-complement traffic pattern.* In this pattern, sources will only send messages to the destination IP whose address is the bit complement of the sources address. This pattern is even less accurate than *uniform*, but is used to evaluate networks as it provides insight to the worst-case of traffic since the destinations are usually on the opposite side of the network.

Hotspot traffic is another commonly used traffic pattern that attempts to provide a more accurate model [31]. In hotspot traffic, a number of *hotspot* IPs are selected by the user with the fraction of the load that is direct to only the *hotspots*. This attempts to mimic a network involving some popular IPs such as a memory block with data common, or a busy processor.

Localized traffic is another traffic pattern that tries to provide a more accurate look at network performance [40]. The user is asked to input the fraction of load that is destined for IPs within a local group of IPs. How the local groups are defined depends on the topology. For *k-ary n-cubes*, the local group is determined as the nodes immediately neighbouring the source on each side (see Figure 24). Thus for a 2D, 64 IP *mesh*, a central IP will have four IPs in its local group, and 59 in the non-local group. For *k-ary n-trees* and *butterfly fat trees*, the local group is defined as the group of IPs that share the same level 1 switch (see Figures 25 and 26). So for a *BFT*, the local group consists of three other IPs. *Octagon's* local group is defined by the lowest level of octagon in the hierarchy, meaning the seven other IPs in a sources bottom level *octagon* (see Figure 27).

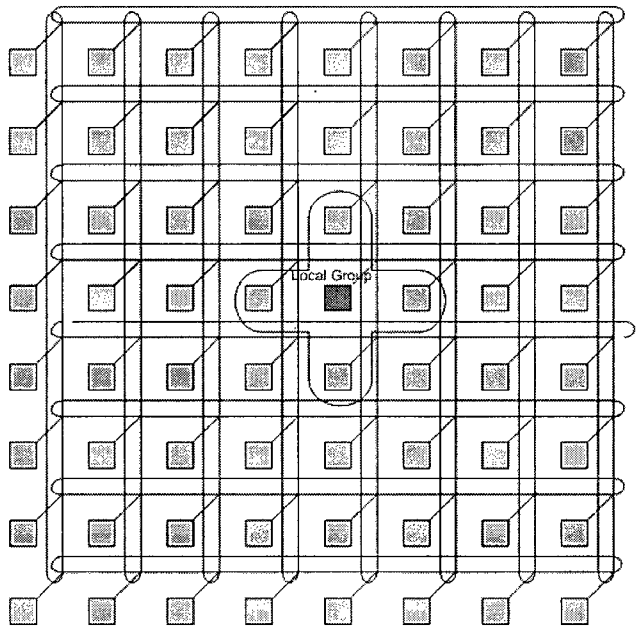

Figure 24: A 64 IP *8-ary 2-cube* with wraparound links. The highlighted IP's local group of 4 is circled.

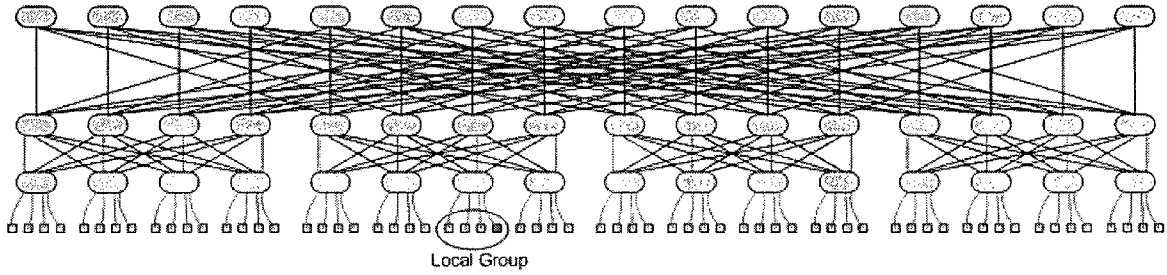

Figure 25: A 64 IP *4-ary 3-Tree*. The highlighted IP's local group of 4 is circled.

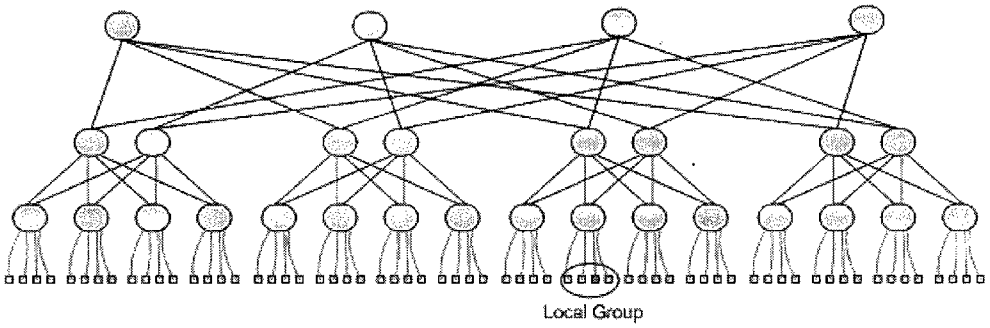

Figure 26: A 64 IP 3 level *BFT*. The highlighted IP's local group of 4 is circled.

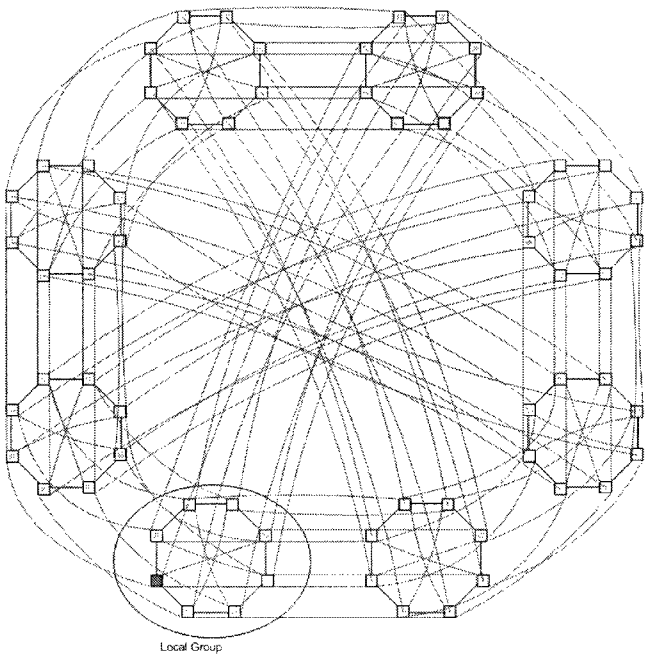

Figure 27: A 2-dimensional 64 IP *octagon*. The highlighted IP's local group of 8 is circled.

It should be noted that these local group definitions vary slightly from topology to topology, but these variations are inevitable as each topology's inherent structure are the biggest determining factor. For that reason, it is fair to compare under these conditions.

In addition to these simulator generated traffic patterns, *NoCSim* has the functionality to input traffic traces from an input file. The format of the file must have the number of messages in the trace file followed by that many source-destination pairs separated by a space. Each pair must end with a return character. This functionality gives the ability to take actual network traffic traces and simulate them with different network topologies and parameters.

The message length in flits can also be determined by the user. *NoCSim* has a limitation that all injected messages must have the same length.

3.9 Source Queue Length

When new messages arrive, they are placed in the message queue at the source IP. It is from this queue that messages are actually injected into the network. This queue is of finite length, and if the network cannot yield a throughput that is equal to the arrival rate, the queue will back-up and eventually overload. *NoCSim* lets the user determine the size of this queue and whether or not the simulation will stop when a queue becomes overloaded. If the later option is not chosen, packets arriving to a source with a full queue will be dropped.

3.10 Simulation Duration

The duration of the simulation in cycles is also to be set by the user. To remove the initial transient effects, a reset statistics time can also be set. It is recommended that durations be long enough such that results are repeatable despite the random nature [23].

3.11 Periodic Statistic Updates

While simulations are running, temporary statistic updates are displayed periodically on-screen. These stats include throughput, average latency, and intra and internode flit movement counts for the previous interval of cycles. The length of these intervals can be set in the main menu, with the default being *2500 cycles*. Figure 28 shows an example output screen.

```
TreeSim     run=1
Topology Type: 0
22 Nodes, 16 IPs
4 Virtual Channels, 1 Buffer Depth
Message length: 16
Queue Size: 100
Load: 1
Duration - reset: 17500
maxMessage= 105
0 temp done: 0
          temp header intra: 0 temp data intra: 0 temp internode: 0 avg_q: 0
2500 temp done: 1287 temp tput: 0.5148 temp lat: 535
 temp h: 0.0008 temp d: 0.0076 temp inter: 0.0132 avg_q: 0.02555
5000 temp done: 1281 temp tput: 0.5124 temp lat: 1694
 temp h: 1.3524 temp d: 20.2044 temp inter: 29.7156 avg_q: 88.4341
7500 temp done: 1277 temp tput: 0.5108 temp lat: 2828
 temp h: 1.3524 temp d: 20.3464 temp inter: 29.8996 avg_q: 98.2632
10000 temp done: 1277 temp tput: 0.5108 temp lat: 3238
 temp h: 1.3276 temp d: 19.8732 temp inter: 29.3604 avg_q: 98.2626
12500 temp done: 1311 temp tput: 0.5244 temp lat: 3172
 temp h: 1.3696 temp d: 20.5268 temp inter: 30.2848 avg_q: 98.1986
15000 temp done: 1356 temp tput: 0.5424 temp lat: 3088
 temp h: 1.3892 temp d: 20.8592 temp inter: 30.926 avg_q: 98.0828
17500 temp done: 1291 temp tput: 0.5164 temp lat: 3083
 temp h: 1.3604 temp d: 20.4164 temp inter: 30.0812 avg_q: 98.1699


**********
Testing Duration: 17500
Total Messages Done: 9089
Throughput: 0.519371
Avg. Message Latency: 2900.49
Avg. Queue Size: 96.8107
Avg. Active Messages / cycle: 0
Average Number of Iterations per cycle: 1
Intranode Header Flit Total: 23778
Intranode Data Flit Total: 356475
Internode Flit Total: 525613
Intranode Header Flit Total/DUR: 1.35874
Intranode Data Flit Total/DUR: 20.37
Internode Flit Total/DUR: 30.035
Elapsed Time (mins:sec) 0:6
Press any key to continue
```

Figure 28: Example *NoCSim* output screen.

3.12 Other Settings

The use of control signals used to probe buffer states can be emulated in *NoCSim*. An extra delay of one cycle can be added before header flits can enter a free buffer to emulate the delay caused by request and acknowledge control signals. This setting is found in the main menu.

For convenience *NoCSim* can string together a series of runs where all network parameters are held constant while one is varied. To setup such a series, the user must

select *'s'* on the main menu. Upon that selection, a menu containing the parameters that the sweep feature supports will be displayed. The supported parameters are:

- Number of Virtual Channels
- Message Length in flits
- Buffer Depth
- Source Queue Length
- Traffic Localization (localization traffic pattern only)
- Traffic Normalized Load
- Fraction of Traffic Destined for Hotspots (hotspot traffic pattern only)

In addition to the input options and parameters, *NoCSim* has some options that relate to the output produced. These include network adjacency graph print outs, latency histograms, average queue length calculations, and debugging features. All available options can be selected in the main menu.

Traffic events can be traced to an output file for debugging. If selected, a message file (*messages.txt*) will be produced containing the times a locations of new arrivals to source queues, the times locations and destinations of messages entering the network from source queues, and similar information about messages being completed.

Also useful for debugging is a port contents print-out option. If selected, a text file (*ports.txt*) will be created containing the contents of every buffer, each cycle. *Flits* are

described by tokens which contain information about the *flit's* type, and message or origin.

Data to produce latency histograms can also be stored and saved to a file (*latencyHistogram.txt*) if chosen in the main menu.

All output files are semi-colon delimited so that parsing is easy and they can be imported into a spreadsheet tool such as *Microsoft Excel* to create graphs.

4.0 Simulator Engine

NoCSim is a *flit*-level cycle based network simulator developed as a *Win32* console project in *C++* using *Microsoft Visual Studio 6.0*. This developing environment was chosen because of its power and debugging and organization strengths.

When the executable is run, default values are set and a menu then displays the simulation parameters and options. Changes to parameters can be made by first entering the menu item character, and then following the appropriate sub-menu that is displayed if new values are to be input. If *'q'* is entered in the main menu, the program will stop. When the run command is entered, *'r'*, the menu exits and network initialization begins.

4.1 The Simulation Cycle

NoCSim is an iterative, cycle-based simulator. The state of the network is stored in data structures described in Appendix B. Every cycle, the network state is updated by an update network function that in turn calls a series of functions with specific tasks. The basic break down of the simulation cycle is outlined below in Figure 29, followed by a closer look at each task.

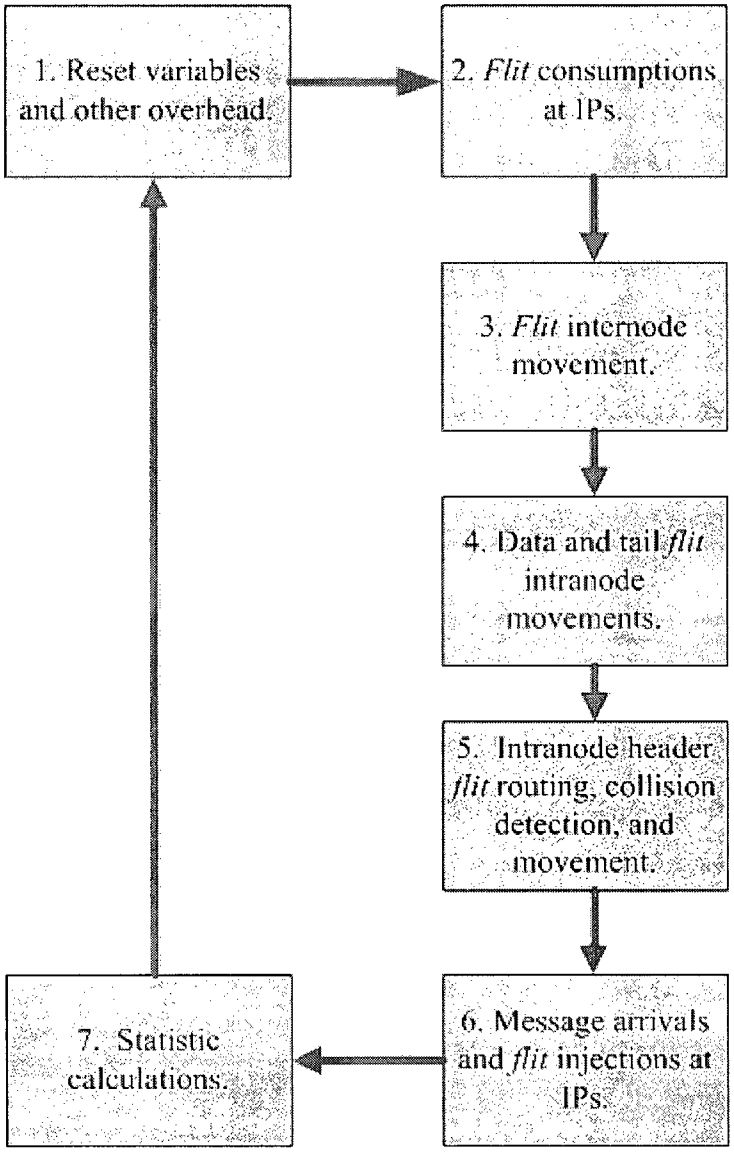

Figure 29: *NoCSim* Cycle Flowchart

To help illustrate the *flit* movement, a simple example is shown along with the explanation. In the diagrams (Figures 30-34), flit types are denoted by the first letter of the token (*H = header, D = data, T = tail*), the message identifier is denoted by the second letter (*message A, B, C, ...).*

1. The state of each message (e.g. header in input buffer) must be reset before the bulk of each cycle. Other small overhead tasks are also done.

2. At each IP, the buffer at each incoming virtual channel from the router is checked for flit tokens to process. If found a *flit* is found, the *flit* token is removed. If a data or header *flit* is consumed, the IPs *.consume* count of the according virtual channel is reduced, and a message specific reserve token is left in the buffer. If a tail *flit* is consumed, the buffer is left blank, and consumption statistics are updated such as latency averages and histograms. The message is also taken off of the active message list.

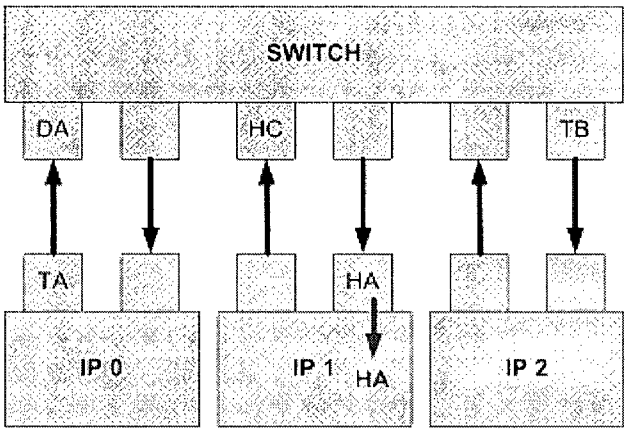

Figure 30: *Flit* marked *HA* is consumed by *IP 1*.

3. Next, *flits* are given the opportunity to advance across internode links, from node output buffers, to node input buffers. In turn, each virtual channel output buffer is checked for flit tokens to advance. Only one *flit* is permitted to cross a link in a given direction during each cycle. The virtual channels are polled in order, but the start of the order rotates in a round robin fashion. This process fairly distributes the available link bandwidth between messages that co-populate the link's buffer resources. When

processed, the *flit* token advances to the input port pointed to by the output port's *.next* variable. *Flit* count statistics are updated to be later used for energy calculations.

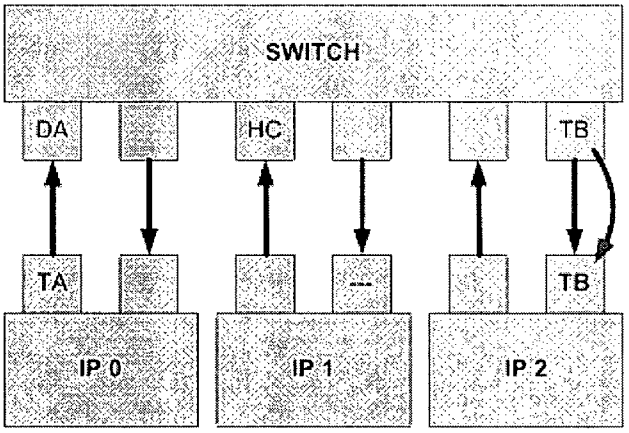

Figure 31: *Flit TB* moves from the switch output port, to IP 2's input port.

4. Data and tail *flits* then advance across nodes internally, from input ports to the appropriate output buffers. Each input virtual channel buffer is checked for non-header *flit* tokens in turn. Like with internode movements, the starting point for buffer polling is rotated in a round robin fashion. Routing is not required for these flits as they simply follow the path of their header flit stored in their message object's path arrays.

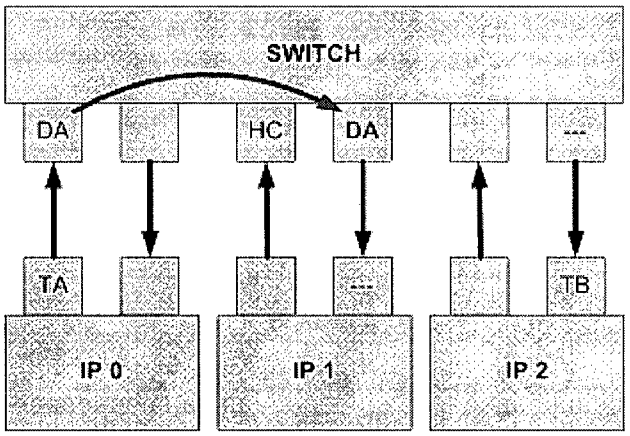

Figure 32: *Flit DA* moves through the switch by following the path established by message *A's* header.

5. Header *flits* are now processed for intranode movement. The active message list is scanned for messages with headers in input ports waiting to be routed. These headers are then routed according to the topology and selected routing algorithm. The requested port's cycle-wise running total of headers requesting it is increased, and the port's collision list is appended. Once all headers have been processed, the totals are scanned for non-zero values. The chosen contention resolution function determines in which output buffers are distributed (see Section 3.7).

If priority based contention resolution is used, blocked messages must be detected at this stage so that priority inheritance can take place.

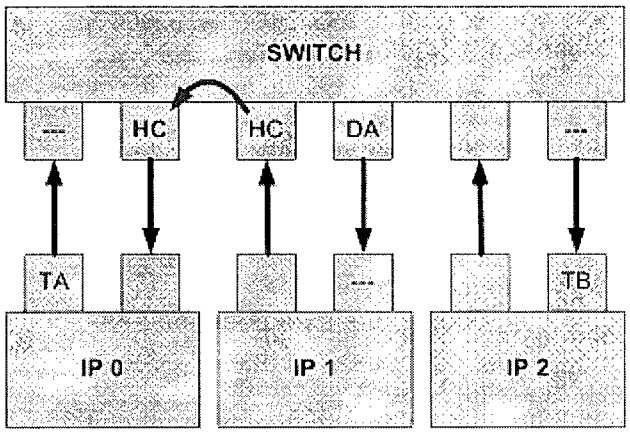

Figure 33: Header *flit HC* is routed to the switch's *IP 0* output port. No collision is detected, so the *flit* is advanced.

6. Each IP is checked to see if its next arrival time matches the current time. If so, a new message is placed in the source queue. Each IP's injection channels are processed for empty buffers and unfinished injections. If a virtual channel is free and the queue is not empty, a new message injection must occur. When that happens, the message index is increased, the active message list is appended, and the IP's inject state count is set to the

message length. The message destination is also determined at this time and all the message state is initialized.

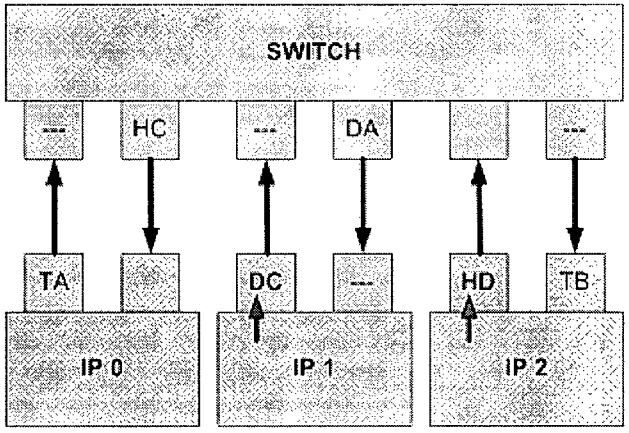

Figure 34: *IP 1* injects *DC* to its output port. A new message *D* is injected from *IP 2*.

7. Cycle-by-cycle calculations are done at the end of each cycle if selected for output. Examples include average queue length calculations, and concurrent message calculations.

This collective process repeats until the cycle count equals the duration parameter set in the main menu. When it does, network statistics are finalized, information is displayed on-screen, and output files are appended. If a variable sweep was selected, re-initialization of all state variables and data structures, statistics are reset, and simulation begins again at cycle zero.

4.2 Time Complexity Analysis

The time required for *NoCSim* to complete a given simulation depends on the selected parameters. Firstly, the impact of each of the modifiable parameters will be discussed.

Simulation Duration (cycles): Each cycle the simulator must perform a number of different processes that includes polling each buffer resource for pending movements. The processes are mostly consistent in their time although process times can grow with congestion with saturated traffic loads. As a result, run times approximately vary linearly with the number of simulation cycles.

Topology: The size and type of topology determines the total number of ports in the network that require polling each cycle. For example, a 16 node *mesh* has 16 5-port switches, (one in each direction and one towards the IP at each switch). The IPs must also be polled for possible injections as well.

Buffer Depth: Each buffer can hold this many *flits*. Each cycle, the worst case is that each space must be polled in every buffer for *flits*. When *flits* are found and moved, the remaining *flit* places are not polled.

Virtual Channels: Each port contains a buffer capable of holding a number of *flits* equal to the buffer depth. Thus increasing the number of virtual channels increases the total number of buffer spaces proportionally. Once a *flit* has been selected from virtual

channel to traverse a link, no other virtual channels in that link and direction will be polled. That is because only one physical link exists between nodes in both directions.

Source Queue Length: This stores the number of messages that can wait at an IP before overloading. Each time a message is taken from an IPs source queue and injected into the network the queue must be updated by shifting all contents of the queue one space closer to the head. Thus in the worst case, the full queue length must be processed at each queue. The queue length is typically small in size relative to the total number of buffers that are polled each cycle however, so queue length only has a large impact on the run time if both load and source queue length are very high.

Traffic Type: Traffic type only slightly alters the time take when destinations are determined for injected messages. It has no significant impact on run time.

Load: The load impacts the number of waiting messages in source queues as well as the likeliness that congestion control logic will be required.

Message Length (flits): The number of *flits* per message impacts the congestion control required, but not the number of buffer resources. Therefore, it is insignificant in comparison and does not alter the run time substantially.

To show the impact of each of these parameters, each parameter will be varied independently and compared to a base case. The base case network simulation has the following parameters:

- Topology: 4x4 *mesh*
- Buffer Depth: 2 *flits*
- Virtual Channels: 4
- Simulation Duration: 10000 cycles
- Load: *Uniform* 0.3 (note at this load, the network is not saturated)
- Message Length: 16 *flits*
- Source Queue Length: 10 messages

Parameter	**Small Value**	**Time (s)**	**Base Value**	**Time (s)**	**Big Value**	**Time (s)**
Topology	4x4 *mesh*	5	8x8 *mesh*	22	16x16 *mesh*	244
Buffer Depth	1 *flit*	21	2 *flits*	22	4 *flits*	31
Virtual Channels	2	18	4	22	8	43
Simulation Duration	10000 cycles	12	20000 cycles	22	40000 cycles	46
Uniform Load	0.01	16	0.3	22	1.0	51
Message Length	8 *flits*	22	16 *flits*	22	32 *flits*	24
Source Queue Length	5 messages	22	10 messages	22	20 messages	24

Table 2: Time Analysis Parameters

From the above results shown in Table 2, it is clear that topology has the greatest impact on run time.

The overall complexity can be best approximated as *O(number of buffer resources).* Breaking the number of buffer resources into its components gives more insight to the formula. After expanding, the complexity becomes:

*O(number of switches * number of ports * number of virtual channels * buffer depth)*

4.3 Limitations

NoCSim does not support adaptive routing algorithms. Adding this feature would require giving the simulator knowledge of the deadlock state of the network.

NoCSim only supports the topologies named in Section 3. Adding new topologies would require adding a create network module to the program as well as adding any topology specific routing modules required.

5.0 Simulation Results

NoCSim has a wide variety of output options for producing useful data. When selecting an appropriate on-chip interconnection network, different parameters must be varied to understand its impact on performance. When a simulation is run, several files containing network statistics are produced.

As an example, some useful results are discussed next with a brief explanation of how results are obtained, and what impact on performance is observed.

5.1 Throughput

The throughput of a communication infrastructure is highly dependent on the traffic pattern. Measuring throughput under uniform random traffic conditions is a generally accepted metric [21] for evaluating parallel systems. Throughput can be thought of as the rate at which a network can consume messages, and is closely related to the peak data rate sustainable to the system. Intuitively, this consumption rate cannot exceed the rate at which messages are injected into the network. Under sub-saturation loads, networks consume messages at the same rate as injected. Once saturated, a deterministic network will consume messages at its maximum rate even if that rate is less than the injection rate. Similar to sand flowing through an hour glass, the rate of sand falling through is independent of the amount of sand waiting to fall.

Throughput output from *NoCSim* is normalized with the following formula [31]:

$$TP = \frac{(TotalMessagesCompleted)(MessageLength)}{(NumberOfIPBlocks)(SimulationDuration)}$$

Where message length is given in flits and simulation duration is given in cycles. The resulting unit for throughput is *flits/cycle/IP*.

Thus, throughput is measured as the fraction of the maximum load that the network is capable of physically handling. A normalized throughput of 1.0 corresponds to all end nodes consuming one flit every cycle.

Throughput by default is always calculated by *NoCSim*.

5.2 Transport Latency

Transport latency is defined as the time (in cycles) that elapses from the arrival of a message in a source IP queue, and the consumption of the tail *flit* at the message's destination IP. At sub-saturation levels of traffic, source queues are usually empty or short, so latency is made up mostly of the delays occurred when a message passes from node to node through the network. Above saturation, source queues become full and long. Messages wait for an amount of time that is dependent on the physical length of the source queue before even entering the network. The source queue wait time is the by far the most significant of the transport latency during times of saturation. As a result, message latencies approach infinity as traffic loads approach saturation.

5.3 Energy Consumption

Energy consumption is an increasingly critical issue in on-chip network design [3]. *NoCSim* produces output that allows a user to calculate the average energy consumed per cycle.

Flit movements are tallied during simulations to give the following output statistics:

- Average internode *flit* movements / cycle
- Average intranode header *flit* movements / cycle
- Average intranode data and tail *flit* movements / cycle

These statistics can be used to calculate the total average energy consumed per cycle by multiplying them by the appropriate network specific energy values. Required values are:

- Average energy consumed by a *flit* traversing an internode link
- Average energy consumed by a header *flit* going through decoding, routing and collision handling logic.
- Average energy consumed by a *flit* moving across a node from input buffer to output buffer.

These values depend on many variables such as topology type, switch size, link length and bandwidth, virtual channel use, and *flit* size. Keeping the calculations of these values external to the simulation tool provides flexibility for energy calculations.

This calculation only takes into account the energy consumed by *flit* movements. Static energy consumption (e.g. leakage current energy) must be added after the fact to provide more accurate approximation.

5.4 Validation

In order to validate the results generated by *NoCSim*, a comparison with *Flexsim1.2* [27] generated results is shown below. Since no tool exists that simulates *octagon*, or *fat-tree* topologies with *wormhole switching*, this example comparison only looks at *k-ary n-cube* topology.

The parameters for both simulations are as follows:

Topology	8 x 8 *mesh*
Buffer Depth	2 *flits*
Virtual Channels	4
Simulation Duration	20000 cycles
Load	*Uniform* 1.0 (note at this load, the network is saturated)
Message Length	16 *flits*
Source Queue Length	10 messages
Control Flits	Used

Table 3: Validation Parameters

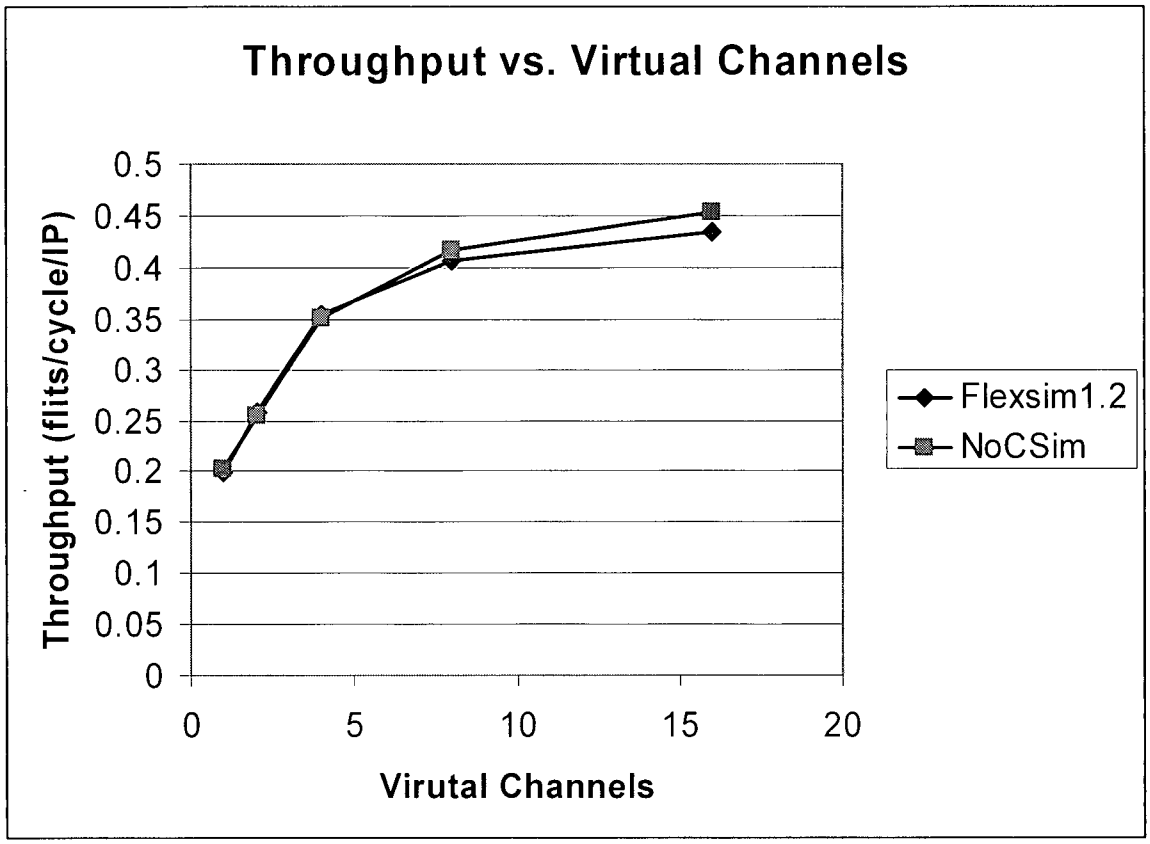

Figure 35: 64 IP *mesh FlexSim1.2* and *NoCSim*

As seen in Figure 35, the results obtained from *FlexSim1.2* and *NoCSim* are very similar. The slight discrepancy can be accounted for by minor implementation differences in collision handling logic and randomness. *Mesh* and *torus* topology results have been matched between *NoCSim* and *FlexSim1.2* under many permutations of network parameters.

5.5 Sample Results

This section serves to illustrate some of the functionality of *NoCSim*, and to show how certain simulation output can give engineers insight to make better decisions regarding on-chip interconnect.

For the examples shown in this document, the following default settings were assumed:

Buffer Depth	1 *flit*
Simulation Duration	20000 cycles
Message Length	16 *flits*
Source Queue Length	10 messages
Reset Statistics Time	2000 cycles
BFT, Fat-Tree Routing	*Turnaround routing*
Mesh	*Deterministic dimension ordered routing*
Torus	*Deterministic dimension ordered routing with virtual split.*

Table 4: Default Parameter Values

5.5.1 Throughput vs. Load

Physical Network Parameters:	
IPs	64
Virtual Channels	4
Traffic Parameters:	
Type	*Poisson distributed uniform random traffic*
Load	Varied.

Table 5: Parameters for Figures 36-37

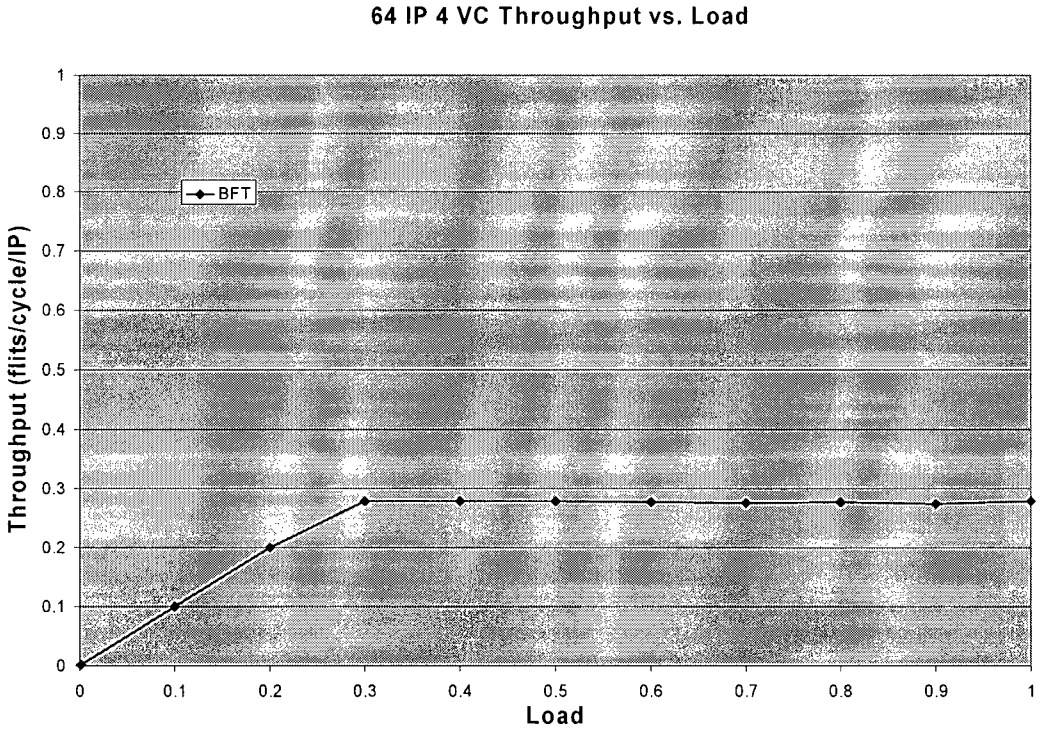

Figure 36: 64 IP *BFT* throughput vs. load

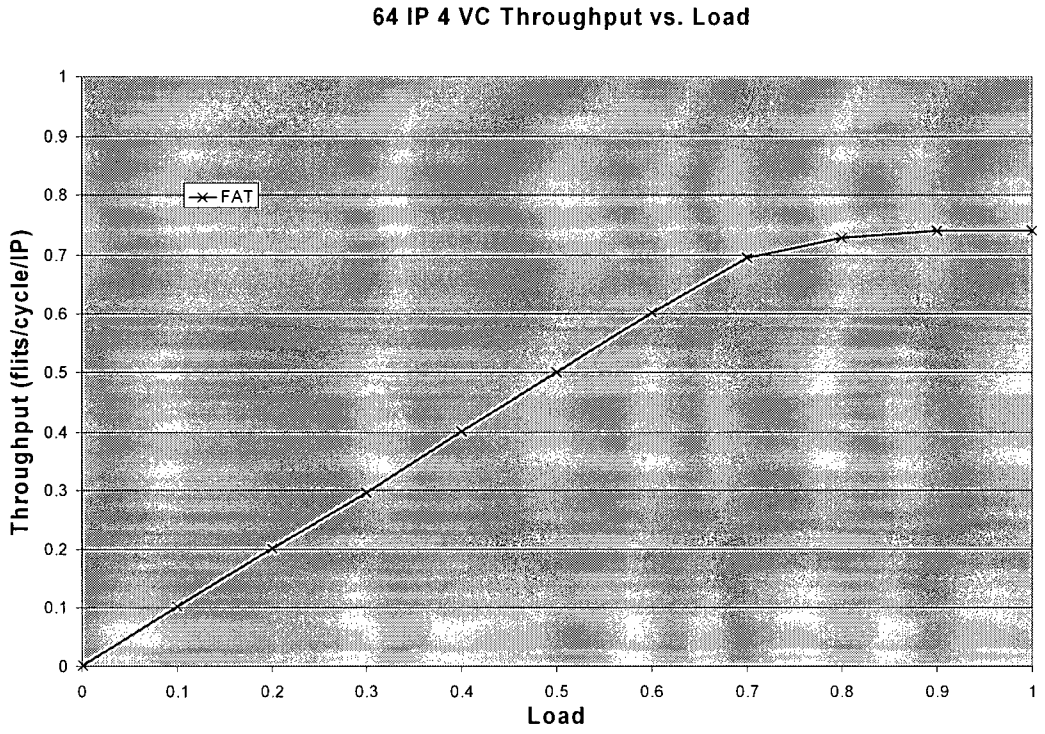

Figure 37: 64 IP *Fat-Tree* throughput vs. Load

Figures 36 and 37 show injection load on the x-axis and throughput on the y-axis. They illustrate the saturation that occurs as load increases past the network's peak rate. Both topologies have equivalent throughput values under saturation. This is because under saturation, there is no accumulating delay caused by messages backing up and messages are consumed at roughly the same rate as they are injected. When deterministic routing algorithms are used no decline is seen when load is increased beyond saturation. Adaptive algorithms experience peaks in throughput as they throttle, or misroute packets to avoid congested areas [35]. These techniques cause performance to fall below deterministic saturation levels.

In this example, the *fat-tree* saturates at a much higher throughput than the *butterfly fat-tree*. This is mostly due to the difference in connectivity between the two topologies.

NoCSim's sweep variable setting makes producing graphs like Graph 2 and 3 simple. The GUI makes it easy to switch between topologies and compare output.

5.5.2 Throughput vs. Virtual Channels

Physical Network Parameters:	
IPs	64
Virtual Channels	Varied.
Traffic Parameters:	
Type	*Poisson distributed uniform random traffic*
Load	1.0

Table 6: Parameters for Figures 38-39

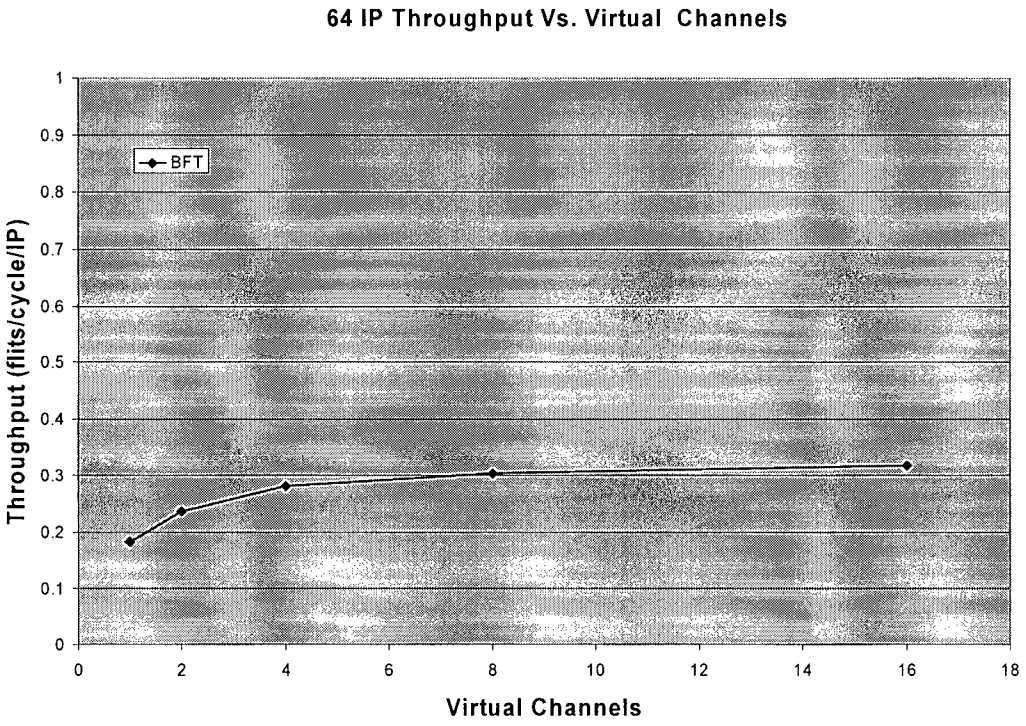

Figure 38: 64 IP *BFT* Throughput vs. Virtual Channels

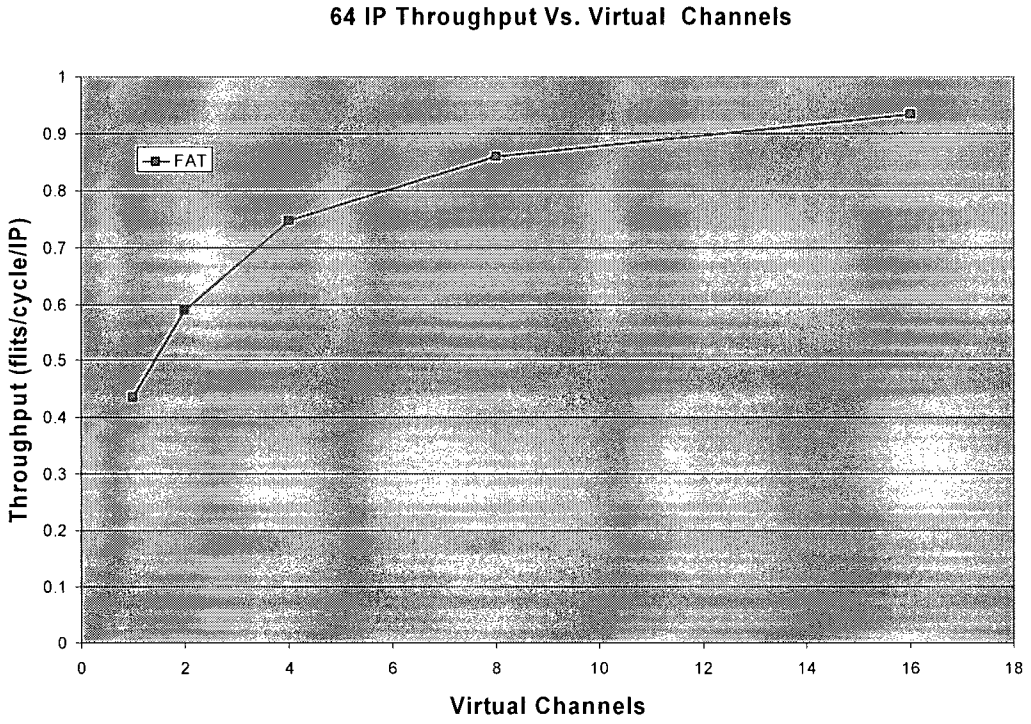

Figure 39: 64 IP *Fat-tree* Throughput vs. Virtual Channels

Figures 38 and 39 show how throughput generally saturates to a peak value with the increase in number of virtual channels. This result is common across all topologies, but saturation occurs at different paces. This graph is useful because even though 4 virtual channels has been generally accepted [9], other network parameters such as topology type and size greatly impact the throughput vs. virtual channel curve and thus could change the optimal number of virtual channels. *NoCSim* makes it possible to create these graphs and determine the optimal number for a given network.

The decision of number of virtual channels is a trade-off between performance (throughput) and overhead (buffer space). Each added virtual channel brings an additional buffer to each input and output channel. Since buffer space is the largest

contributor to overall interconnect area, efforts should be taken to not waste chip area by selecting a number that is too large. Saturation occurs because eventually, extra virtual channels do not make an impact because they are never populated by flits. The behaviour is further explained by modelling virtual channel population as a *Markov-Chain* process [9]. Throughput increases greatly when two or four virtual channels are used making the trade-off well worth the extra area.

5.5.3 Localization vs. Throughput

Physical Network Parameters:	
IPs	64
Virtual Channels	4
Traffic Parameters:	
Type	*Poisson distributed localized random traffic*
Localization	Varied.
Load	1.0

Table 7: Parameters for Figures 40-41

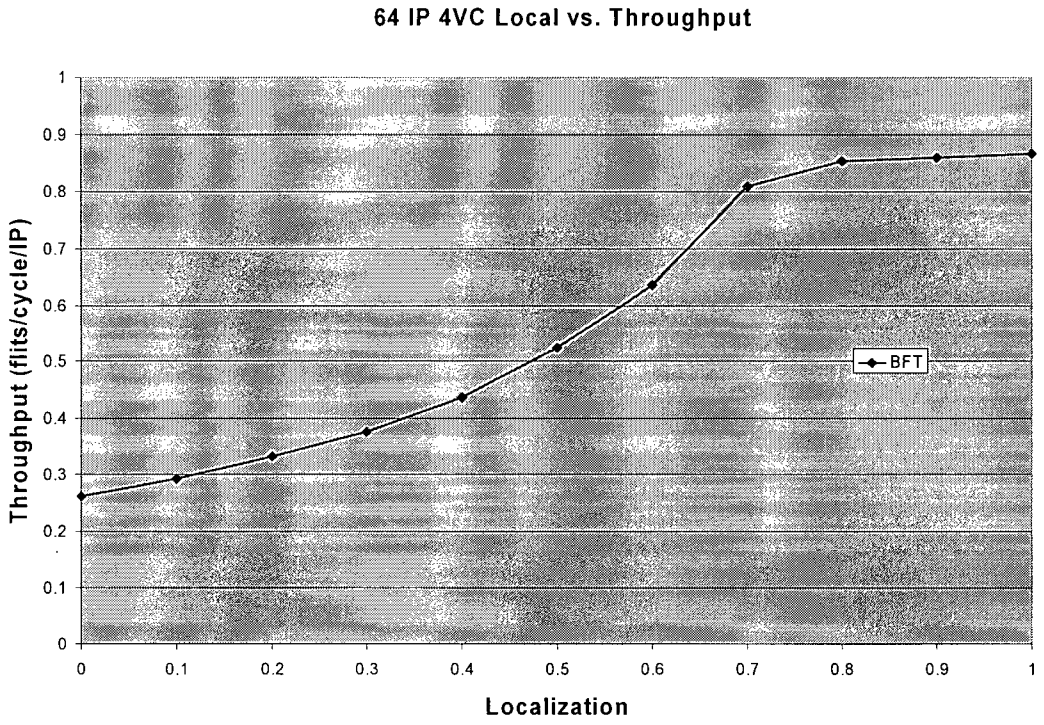

Figure 40: 64 IP *BFT* Throughput vs. Localization

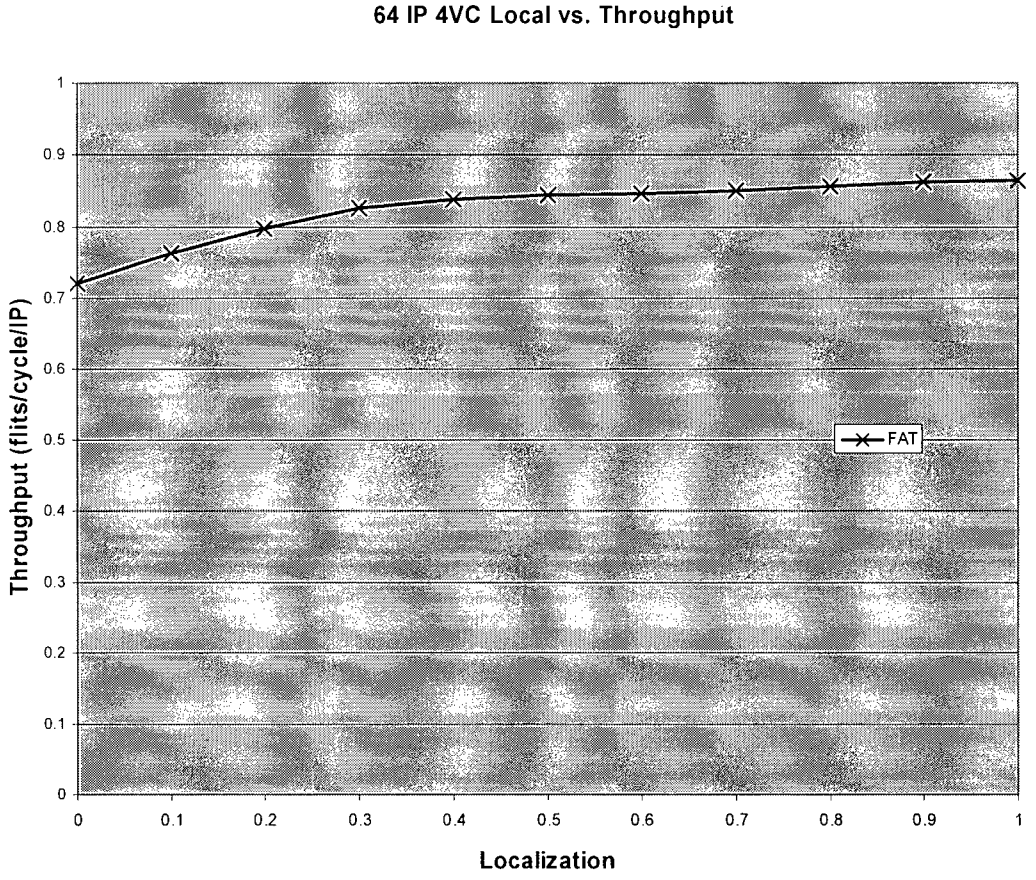

Figure 41: 64 IP *Fat-tree* Throughput vs. Localization

Uniform traffic provides a benchmark for analyzing different networks [31]. Real life on-chip network traffic patterns do not mimic this uniformity however. Localization is a proposed model for traffic that adds some amount of realism to the simulation. *NoCSim* allows the user to select the traffic pattern from the main menu. If localization is chosen, the fraction of traffic that is destined for IPs in the same local group (see Section 3.8 for local group definitions) may also be specified.

This localization graph gives users a chance to study the effects of the change of localization on throughput. From the example graph it appears that these different topologies both experience a rise in throughput with localization, but this relationship is not linear and varies greatly. Topologies with high connectivity seem to be relatively independent of traffic localization, while lesser connected networks show a larger dependence. In this example, the *BFT* shows the greatest dependence on localization, while the fat-tree shows very little. Whatever the topologies, analyzing their behaviour under localized traffic conditions is an important metric, and being able to implement such a situation with little effort is a great feature.

The localization of a traffic pattern is greatly dependent on the application, but having a graph like this gives designers an opportunity to get a throughput value for an approximated fraction of localization.

5.5.4 Latency vs. Load

Physical Network Parameters:	
IPs	64
Virtual Channels	4
Traffic Parameters:	
Type	*Poisson distributed uniform random traffic*
Load	Varied.

Table 8: Parameters for Figures 42-43

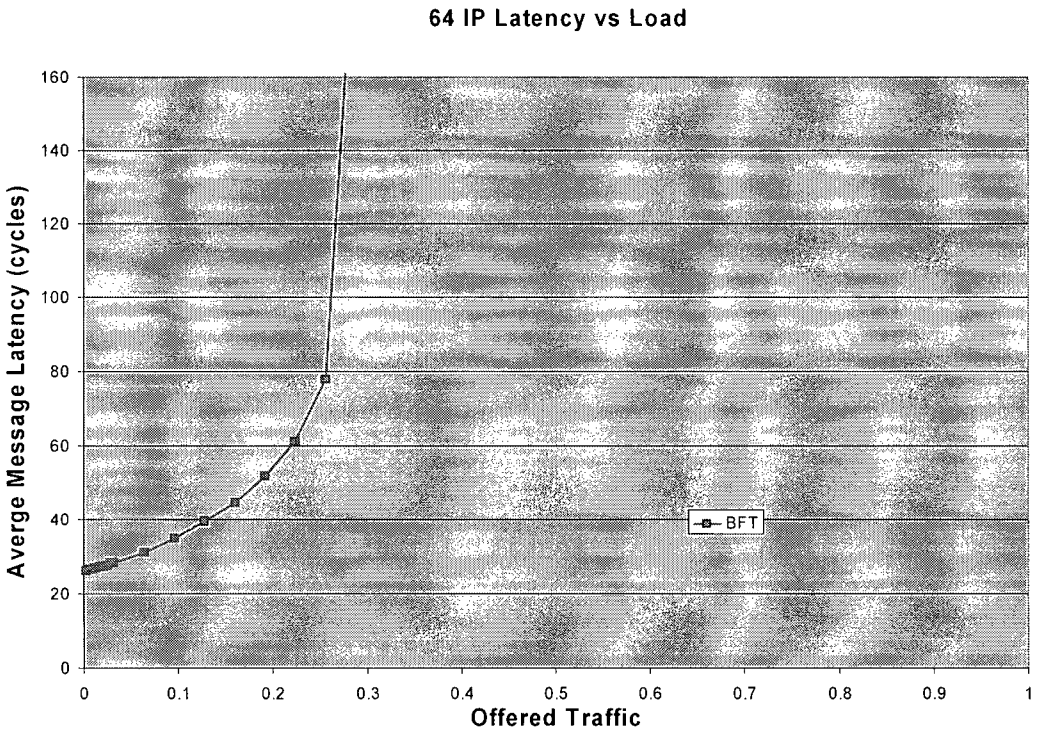

Figure 42: 64 IP *BFT* Latency vs. Load

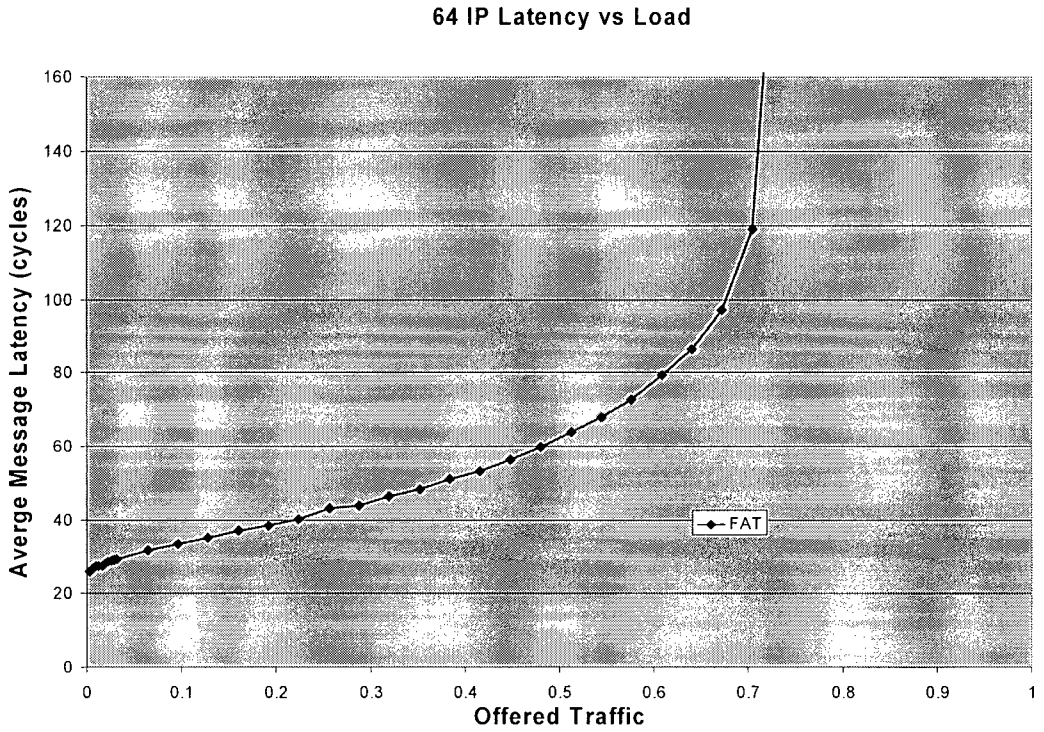

Figure 43: 64 IP *Fat-tree* Latency vs. Load

NoCSim can produce data for latency vs. load graphs to give designers a sense of expected latencies under different loads. From Figures 42 and 43, it is clear that all topologies experience infinite latency beyond their saturation points. This is because source queues become backed up and overflow which effectively forces new messages to wait indefinitely causing latency to approach infinity. The load at which each network saturates is of great concern because of that fact. Communication-intensive applications with expected loads of greater than 0.5 should only consider highly connected topologies.

These graphs were produced with a variable sweep of the load parameter. Average latency is a default output of *NoCSim*, and provides a key piece of the overall network performance picture.

5.5.5 Energy vs. Load

Physical Network Parameters:	
IPs	256
Virtual Channels	Varied.
Average energy per flit link traversal (BFT)	113 pJ
Average energy per flit link traversal (Fat-tree)	113 pJ
Average energy per header flit routed (BFT)	47.79 pJ
Average energy per header flit routed (Fat-tree)	63.71 pJ
Average energy per data or tail flit routed (BFT)	40.32 pJ
Average energy per data or tail flit routed (Fat-tree)	53.79 pJ
Traffic Parameters:	
Type	*Poisson distributed uniform random traffic.*
Load	Varied.

Table 9: Parameters for Figures 44-45

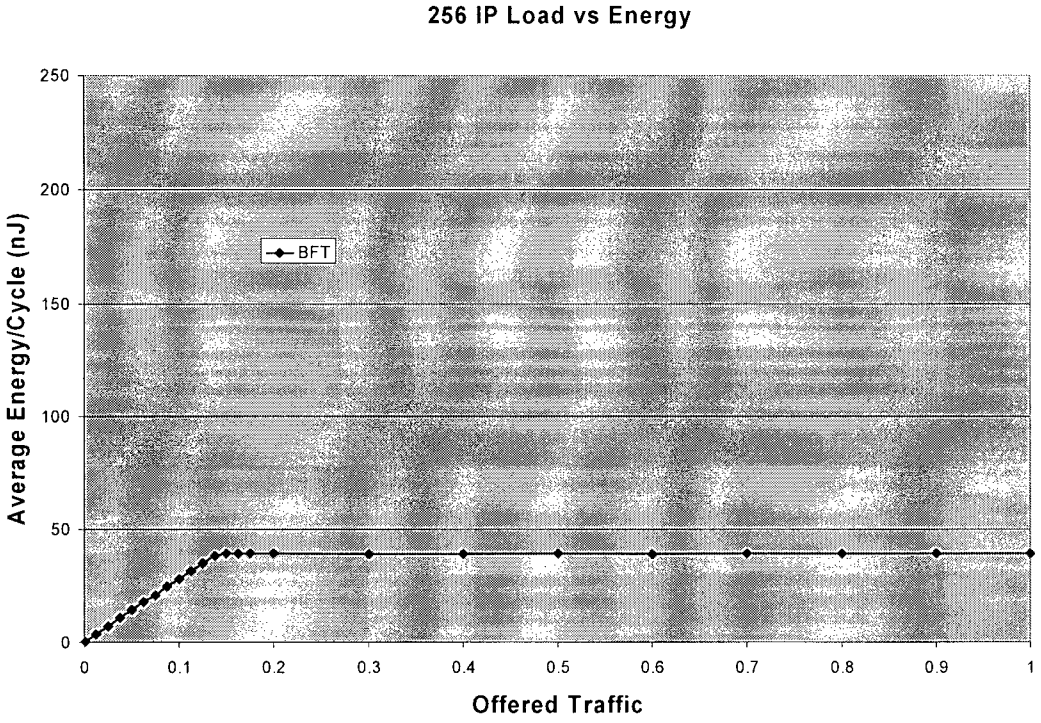

Figure 44: 256 IP *BFT* Energy vs. Load

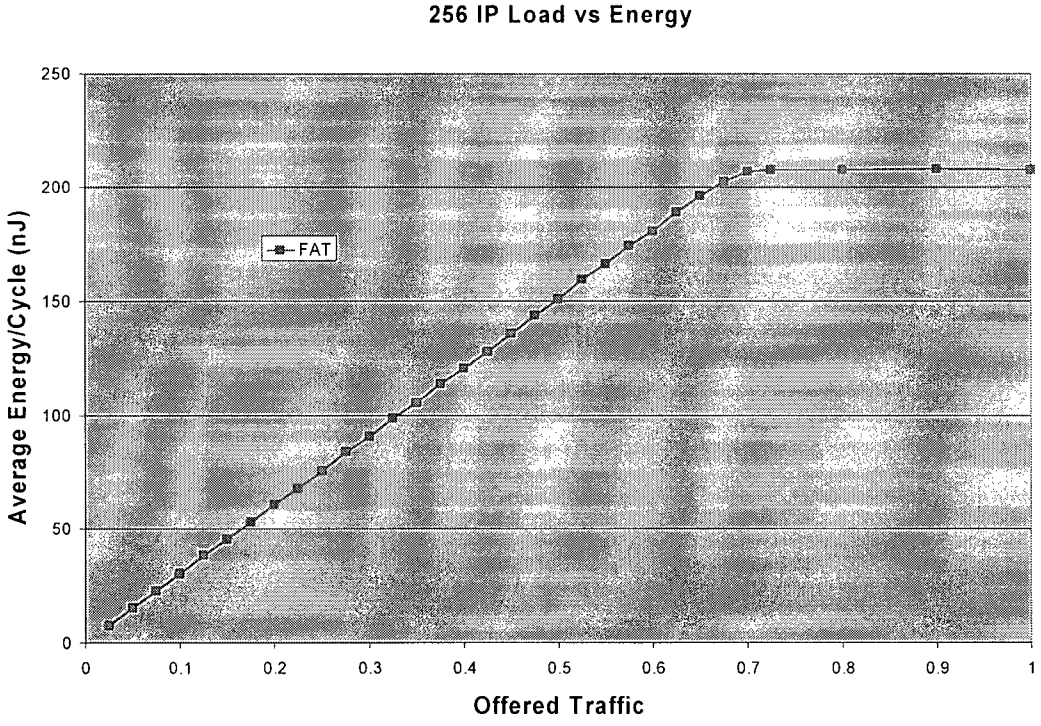

Figure 45: 256 IP *Fat-tree* Energy vs. Load

As the relative importance of energy consumption grows in the SoC domain, so does the need to critically analyze the power requirements of chip interconnect. NoCSim gives users the ability to create energy data from simulation outputs. As mentioned in Section 5.3, the simulator produces flit traversal counts for links and switches. Those counts can be multiplied by specific per/flit energy calculations that are specific to the simulated network to get average power results. Energy vs. load graphs show the effect of load saturation on the energy consumed in the network.

5.5.6 Latency Histograms

Physical Network Parameters:	
Topology	BFT
IPs	64
Virtual Channels	4
Source Queue Length	1
Traffic Parameters:	
Type	*Poisson distributed uniform random traffic.*
Load	1.0

Table 10: Parameters for Figures 46-47

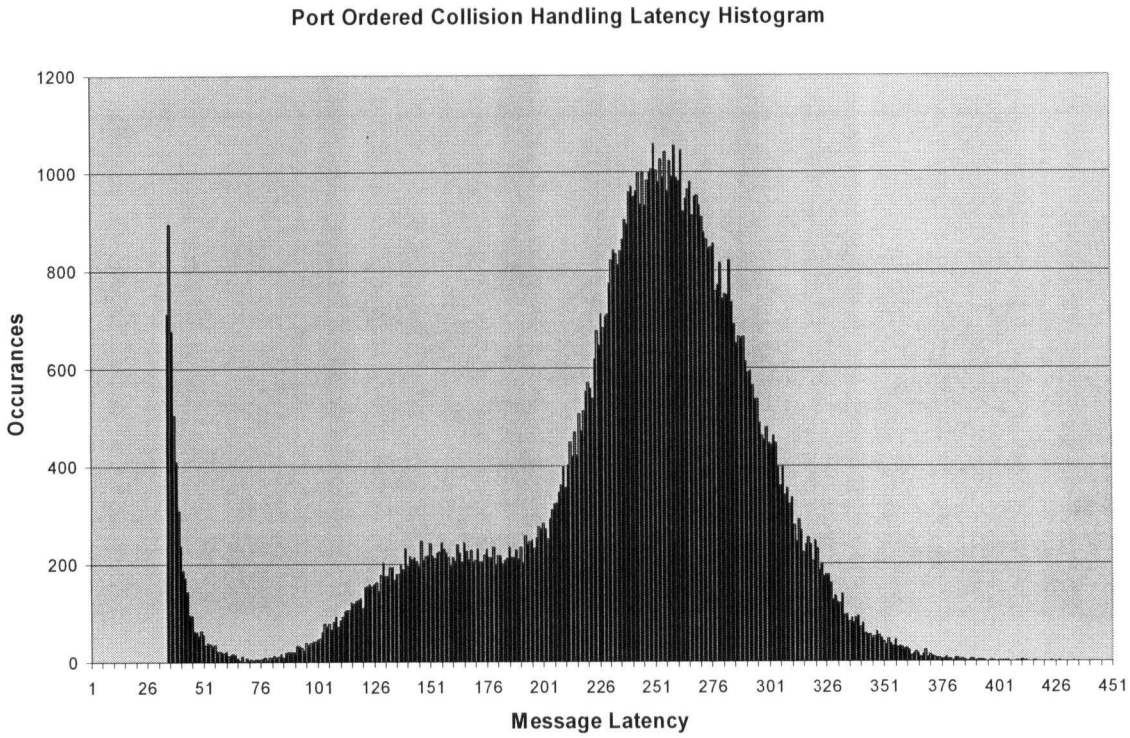

Figure 46: *Port Ordered Collision Handling* Latency Histogram

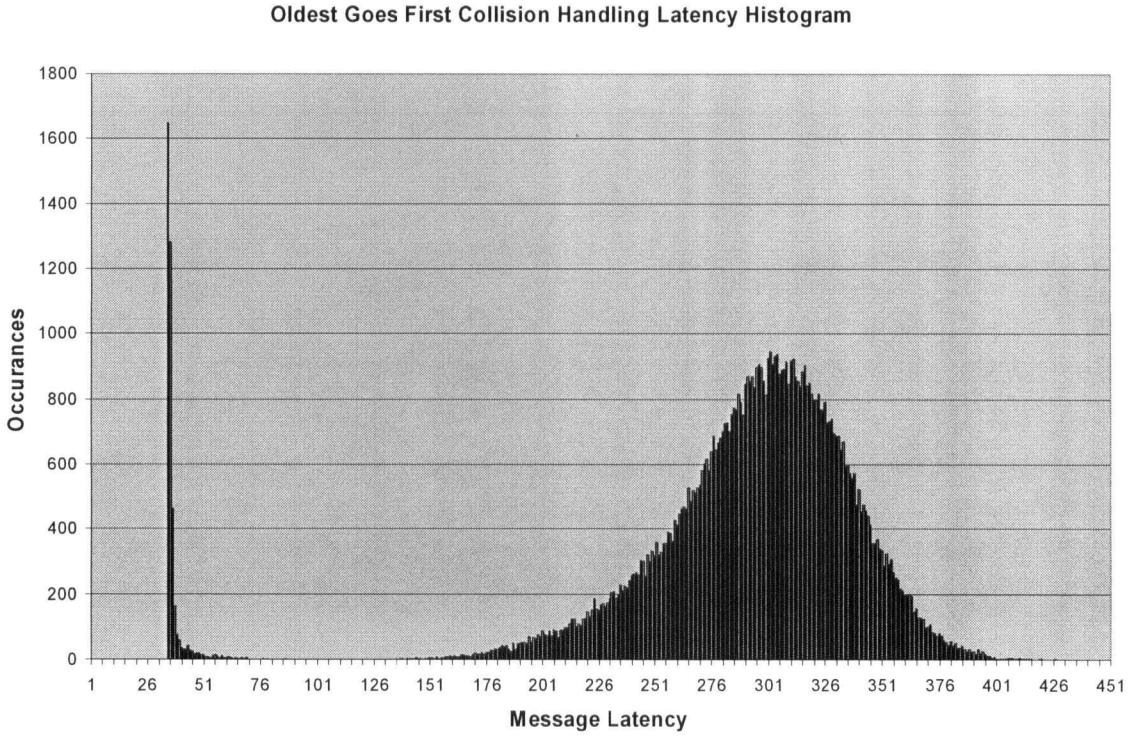

Figure 47: *Oldest Goes First Collision Handling* Latency Histogram

NoCSim also has the functionality to produce latency histogram data if desired by the user. Latency histograms provide a clear way to view the distribution of message latencies and to predict *quality of service (QoS).* Many applications exist where a specific *QoS* constraint must be met. *QoS* constraints may be specified in the form ``*X %* of all messages must have latency lower than *Y*``. To meet these requirements for a network, different strategies could be implemented depending on the degree of *QoS* required, and other network constraints like area, or energy. The distribution of message latencies is dependent on the collision handling scheme. Fair schemes will have lumped results, while greedy schemes will be more spread out with far worse 'bad' cases.

A latency histogram can serve to provide data for calculating if *QoS* constraints are met, and also to give an idea of how close or far the network is away from meeting its target *QoS*. For instance, if network latency was generally low, but a few base cases were pushing the average up, efforts could be taken to remove those cases. Another case would be where the most messages were arriving with latencies that were beyond the *QoS* cut-off. A histogram would clearly show that the network was not capable of meeting *QoS* goals and a different topology with a higher bandwidth might be required to meet the constraints.

In the above graphs, the performance difference between port-ordered collision handling and oldest goes first collision handling is clearly shown. The port-ordered graph is more distributed, while the oldest goes first graph has a clear spike at the shortest path latency, and a clear hump closer to the average latency. The average latency calculation for these schemes may be quite similar, but when a histogram is created, the vast difference in latency distribution is apparent.

6.0 Conclusions and Future Work

6.1 Conclusions

In this thesis we have described the need for a single network on chip simulation platform that is capable of accurately modelling a network on chip interconnect. This need stems from the paradigm shift that will happen in SoC design methodology where large numbers of heterogeneous IP blocks will by integrated together using a standard template. Each IP block is capable of sending and receiving data packets through the interconnect. The non-scalability of buses as on-chip interconnects forces the research community to select a more scalable alternative for communication.

Many different types of network interconnects have been proposed for the NoC domain such as *k-ary n-cubes, butterfly fat-trees, k-ary n-trees,* and *octagons*. The design of these interconnects was shown to have many new challenges such as physical switch design and flow control. After surveying the available network analysis techniques, it was determined that a network simulation tool was the best option for providing designers with the required system performance.

Several network simulation tools exist, but fail to contain all functionality desired for NoC simulation. To fill this void we have developed *NoCSim*, an iterative *flit*-level

network on chip simulator capable of simulating networks under a wide variety of parameters and topologies.

NoCSim's feature set includes the ability to set physical network parameters. This includes topology type, and number of IPs as well as switch parameters like buffer depth, and virtual channels. Routing and collision handling options are also available.

Several network traffic patterns can be chosen such as *uniform random traffic, hotspot,* and *localized.* Functionality is also in place to input trace driven traffic from external sources.

Simulations can be run as a series or as a stand alone with settable duration, reset time, and a number of output options including latency histograms, energy calculations, throughput, and latency.

Overall, *NoCSim* gives designers the ability to make informed decisions about on-chip network interconnection. It does not provide a clear answer to "which is the best topology in every situation" because that answer does not exist. Since there are so many variables that depend on each other so tightly, gathering a large amount of accurate simulation data is the best and only way to provide insight to which network suits which application best.

In conclusion, the large-scale SoC era will soon be a reality. Efforts need to be made now to beginning solving the design issues that face SoC engineers in this new NoC paradigm. *NoCSim* is capable of providing useful, accurate system performance estimates that can be used as clear evidence for backing interconnect implementation decisions.

6.2 Future Work

To better evaluate networks for NoC purposes, additional considerations need to be made. For one, fault tolerance needs to be accounted for when analysing topologies. Certain topologies are more resistant to faults due to inherent redundancy in their connectivity graphs. Routing also plays a large role in the impact of fault tolerance. For instance, an adaptive routing algorithm implemented on a 2D mesh makes it much more tolerant than the same network using deterministic *dimension ordered routing*. This is because if one link joining two nodes fails, the adaptive algorithm has the means to route packets along a different route to reach their target. Under *DOR* however, source-destination pairs must always follow the same path, so paths that go through the failed link will never reach their destination. Faults could be easily simulated in *NoCSim* with some extra logic if desired.

The problem of coupling realistic traffic patterns to applications is a difficult one to solve. While *uniform traffic* provides a benchmark, it fails to show what the network behaviour will be under real application traffic. Categorizing application traffic would

give more insight, but this is difficult as traces from SoC traffic is application specific and hard to come by. Benchmark traffic traces from a wide variety of applications is needed to create clear results. *NoCSim* currently has the ability to run traffic traces, but the functionality to vary message length from packet to packet needs to be added.

Multicasting functionality is another commonly studied problem that should be added to the simulator to provide a more complete picture. Header flits could include multiple destination addresses, or additional header flits could be appended to a packet to provide the functionality. These options and more should be implemented to first determine if multicasting is beneficial, and how to go about using it.

Appendix A: The *NoCSim* Source Code in C++

Source code from the No*CSim* application can be found at the following link. Please see *readme.txt* for details.

`/CMC/tmp/library/thesis/michaelj`

Appendix B: *NoCSim* Data Structures

Initialization of the several data structures involved in the simulation process is done in the initialization routing. Together these data structures make up the state of the network as they are produced according to the chosen network parameters and store all flit level information. Some of the major data structures are described below.

B.1 *Ports* Data Structure

The ports data structure is an array of port objects. The size of this array is equal to the size of the port object multiplied by the total number of ports in the network. The total number of ports value is given by the number of nodes multiplied by the number of ports per node. Each port contains an input and output portion since all links are assumed to be bi-directional. The port object is made up of several variables:

Type	Variable name	Description
int*	o_buff	output buffer contents
int*	o_buff	input buffer contents
int	next	port at other end of the link
int	last_virt_sent	stores the last virtual channel serviced from output channel
int	last_virt_sent_intra	stores the last virtual channel serviced from input channel
bool	sent_this_cycle	if a *flit* has advance to the next port from this output buffer
bool	sent_this-cycle_intra	if a *flit* has advance to the next port from this input buffer

Table 11: *Ports* Data Structure

The o_buff and i_buff pointers point to the arrays which contain the contents of each flit space in the port. Each array can hold (*buffer depth * number of virtual channels*) integers. Integer tokens are stored in these spaces to represent flits of different types, place holders, or empty spaces. The token representation system is as follows:

Token Value	Representation
< 0	Space is reserved for message with index equal to -1*token.
0	Space is empty
(1, maxMessage)	Space contains a data flit belonging to message with index equal to token
(maxMessage, 2*maxMessage)	Space contains a tail flit belonging to message with index equal to token mod maxMessage
> 2*maxMessage	Space contains a header flit belonging to message with index equal to token mod maxMessage

Table 12: Token Representations

Each array must be indexed properly when being referenced. The indexing is ordered such that all of a virtual channel's buffer spaces are given before moving to the next virtual channel.

The variable `next` holds the ports array index of the port at the other end of this port's link. These values are set in the network creation function and effectively join the topology together.

The `last_virt_sent` and `last_virt_sent_intra` integers are used to make sure virtual channels are alternated in turn. I.e. to ensure virtual channel x is not always serviced from a given port. `Sent_this_cycle` and `sent_this_cycle_intra`

ensure multiple virtual channels of input or output ports are not sent during the same cycle.

B.2 *Msgs* Data Structure

The `msgs` array holds a number of `msg` objects. The number of said objects is determined by the `maxMessage` variable and has a value greater than the total number of active messages that could fit in the network at one time (*number of nodes * number of virtual channels*).

As messages are injected into the network from source queues, a `msg` object is initialized at the current `msgIndex` location of the `msgs` array. The `msgIndex` holds the next available slot in the `msgs` array that does not contain an active message. Active messages are defined as messages that have been injected from a source queue, but have not had their tail flit consumed at their destination IP. If the `msgIndex` is increased beyond the `maxMessage` limit, it is restarted at one, reusing the memory allocated for old messages which have since been completely consumed. This measure ensures that memory requirements remain low throughput simulations of any duration.

Each `msg` object holds the state of a particular message in the network.

Type	Variable name	Description
int*	source	address array pointer
int*	dest	destination array pointer
int*	path	store ports along path
int*	vpath	
int	path_length	
int	end_time	
int	upper	for torus deadlock free routing
int	dim	holds dimension of travel for message
int	num	message number
int	next_collide	used for collision detection linked list
int	prev_collide	used for collision detection linked list
int	next	used for active message linked list
int	prev	used for active message linked list
int	virt	current virtual channel
int	priority;	used in priority scheme collision handling
int	temp_priority;	inherited priority
int	req_port	the next port the header would like to enter
bool	header_moved;	if the header has moved this cycle
bool	header_done	if the header has been consumed at the destination
bool	header_in;	true if the msg header is in an input buffer
bool	is_blocked;	used for priority inheritance

Table 13: *msgs* Data Structure

The `source` and `dest` pointers store the starting locations of address arrays. The size of these addresses depends on the topology type and size. This form of address makes routing simpler for topologies with multiple levels or dimensions.

The path of each message header is stored in port and virtual channel arrays which are the size of the worst case path through the network. These arrays are used to allow data flits follow the path set by the header flit through the network.

The start and end time of each message is stored in the `msg` object also. This has obvious latency calculation uses, but also provides a way to determine the state of the message. When messages are injected, the `end_time` is set to -2, meaning active. When consumed, `end_time` holds the cycle at which the consumption was completed.

As well as being in the `msgs` array, to improve speed messages are linked together in two separate linked lists. One being the active message list, used to determine the locations of headers when routing and in collision detection. The other being a collision linked list, used for collision handling. This second list contains all messages in contention for the same resource. The short list enables different collision handling routines compare message information quickly. More information regarding collision handling is found in Section 3.7.

Message header state information is also stored in each `msg` object. This includes if the header had moved this cycle, if the header has been consumes, if the header is in an input buffer, and if the header is blocked.

B.3 *IPs* Data Structure

To keep track of IP message injection and consumption, an array of `ip` objects is used. Not surprisingly, the array contain *n* IPs, where n is the number of IPs in the network. The variables of an `ip` object are shown below:

Type	Variable
`int*`	`generate`
`int*`	`generate_msgs`
`int*`	`consume`
`int*`	`consume_msgs`
`int`	`current_msg`
`int*`	`queue`
`int`	`queue_pop`
`int`	`next_arrival`

Table 14: *IPs* Data Structure

The `*generate` pointer refers to an array of values with length n, where n is equal to the number of virtual channels. The slots in the array represent the different injection lanes at the particular IP since each IP is assumed to be able to inject up to n messages at a time. The values in those slots represent the number of flits left in the generation of a particular message. E.g. an *n=3* IP with `generate[0]=5, generate[1]=3,` and `generate[2]=0` would have five flits left to generate in lane 0, 3 left in lane 1, and not be injecting a message in its third lane.

The `*generate_msgs` pointer refers to a similar array which holds the `msgs` array indexes of the messages being generated in each lane.

Likewise, the `*consume`, and `*consume_msgs` pointers refer to arrays that similarly keep track of IP flit and message consumption. The value of `consume[i]` refers to the number of flits left to completely consume message `consume_msg[i]`. IPs are also assumed to be able to consume n messages at a time.

IPs are also responsible for keep track of the source queue information. The `*queue` pointer refers to the source queue array which holds a number of integers equal to the source queue length. When new messages arrive at the queue, their arrival time is stored in the queue so accurate latencies can be calculated. Messages are not given indexes or destinations until they are injected into the network.

The `next_arrival` integer stores the cycle time for the next arriving message at this node. `Next_arrival` is determined by adding an exponentially distributed variable to the current cycle. Doing so gives *Poisson* distributed arrival times [22].

B.4 *Headers* Data Structure

The headers array holds x header objects, where x is equal to the total number of ports in the network. Each header object holds information required when detecting collisions.

Type	Variable
`int`	`numRequested`
`int`	`firstCollision`
`int`	`currentCollision`

Table 15: *Headers* Data Structure

The `numRequested` integer holds the number of headers that have requested the particular port specified by the index of the headers array. `firstCollision`*n* gives the index of the first message that requested the port, and serves as a pointer to the beginning of the header collision list for this port. `currentCollision` serves as a place holder for passing through said list.

B.5 Flit Transfer Arrays

Each cycle, flits are set for movement if there is buffer space available at their requested channel. This information is stored in a series of integer array of size x, where x is the

total number of ports in the network. The array names are listed below followed by a brief overview of their functions:

Array
to_internode_move_ports
to_internode_move_oldports
to_internode_move_virts
to_internode_move_oldvirts
to_internode_move_flits
to_intranode_move_ports
to_intranode_move_oldports
to_intranode_move_virts
to_intranode_move_oldvirts
to_intranode_move_flits

Table 16: Flit Transfer Arrays

The `inter` and `intra_move_ports` arrays store the port indexes of ports that will be receiving a new flit at the end of the cycle. The `oldports` versions of those arrays store the indexes of the ports that the flits will be removed from. Similarly, the `virts` arrays store the associative virtual channel indexes to accompany the port information. The flits arrays store the actual flit tokens that are to be transferred.

Each of these arrays are coherent in index, meaning `to_internode_move_port[x]` and `to_internode_move_oldports[x]` describe information about the same flit

transfer action. The intra, and inter node arrays are independent however. All of these arrays are reinitialized to zero at the start of each cycle.

References:

[1] P. Guerrier, A. Greiner, "A Generic Architecture for On-Chip Packet-Switched Interconnections", *Proceedings of Design, Automation and Test in Europe Conference and Exhibition 2000*, pp. 250-256.

[2] W.J. Dally, B. Towles, "Route Packets, not Wires: On-Chip Interconnection Networks", *Proceeding of DAC 2001*, pp.684-689.

[3] P.P. Pande, C. Grecu, A. Ivanov, R. Saleh, "Design of a Switch for Network on Chip Applications", *Proceedings of ISCAS*, pp. 217-220.

[4] P.P. Pande, C. Grecu, M. Jones, A. Ivanov, R. Saleh, "Architecture Evaluation for Communication-Centric SoC Design", *submitted to ISCAS 2004.*

[5] AMBA Bus specification, http://www.arm.com.

[6] CoreConnect Specification, http://www-3.ibm.com/chips/products/coreconnect/.

[7] Wishbone Service Center, http://www.silicore.net/wishbone.htm.

[8] Design and Reuse website, http://www.us.design-reuse.com/sip.

[9] W.J. Dally, "Virtual-Channel Flow Control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, March 1992, pp. 194-205.

[10] Open Core Protocol, www.ocpip.org

[11] W.J. Dally, C.L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks", *IEEE Transactions on Computers*, vol. C-36, no. 5, May 1987, pp. 547-553.

[12] J. Duato, "A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks", *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, October 1995, pp. 1055-1067

[13] K.V. Anjan, T.M. Pinkston, "DISHA: A Deadlock Recovery Scheme for Fully Adaptive Routing", *Proceedings of the 9th International Parallel Processing Symposium*, 1995, pp. 537-543.

[14] W.J. Dally, C.L. Seitz, "The Torus Routing Chip", *Journal of Distributed Computing*, vol.1, no. 3, 1986.

[15] C. Leiserson et al. "The network architecture of the connection machine CM-5", *Symposium on Parallel and Distributed Algorithms*, June 1992, pp. 272-285.

[16] F. Petrini, W. Chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High-Performance Clustering Technology", *IEEE Micro*, 2002, vol. 22, no. 1, pp 46- 57.

[17] R.I. Greenberg, L. Guan, "An Improved Analytical Model for Wormhole Routed Networks with Application to Butterfly Fat-Trees", *Proceedings of the 1997 International Conference on Parallel Processing*, pp. 44-48.

[18] F. Petrini, M. Vanneschi, "k-ary n-trees: High Performance Networks for Massively Parallel Architectures", *Proceedings of the 11th International Parallel Processing Symposium, IPPS'97*, Geneva, Switzerland, April 1997, pp. 87-93.

[19] F. Karim, A. Nguyen, S. Dey, "An Interconnection Architecture for Networking Systems on Chips", *IEEE Transactions of Computing*, September 2002, vol. 22, no.5, pp. 36-45.

[20] C. A. Zeferino, A. A. Susin, "SoCIN: A Parametric and Scalable Network-on-Chip", *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI'03)*.

[21] K. J. Liszka, J. K. Antonio, H. J. Siegel, "Problems with comparing interconnection networks: Is an alligator better than an armadillo?", *IEEE Concurrency*, Vol. 5, No. 4, Oct.-Dec. 1997, pp. 18-28.

[22] A. Leon-Garcia. "Probability and Random Processes for Electrical Engineering Second Edition", *Addison-Wesley publishing Company, Inc.,* May 1994.

[23] R. Jain. "The Art of Computer Systems Performance Analysis, Techniques for Experimental Design, Measurement, Simulation, and Modeling", *John Wiley & Sons, Inc.* 1991.

[24] Yi-Ran Sun, "Simulation and Performance Evaluation for Networks on Chip, Master of Science Thesis", 2001.

[25] NS-2 Home Page: http://www.isi.edu/nsnam/ns/

[26] "IRFlexSim0.5 User Guide", *SMART Interconnects Group*, USC, 2002

[27] "FlexSim1.2 User Guide", *SMART Interconnects Group*, USC, 2002

[28] S. Kumar et al, "A Network on Chip Architecture and Design Methodology", *Proceedings of the IEEE Computer Society Annual Symposium on VLSI* (ISVLSI'02), 2002.

[29] L. Benini, G. De Micheli, "Networks on Chips: A New SoC Paradigm", *IEEE Computer*, vol. 35, no.1, Jan. 2002, pp. 70-80.

[30] W.J. Dally, B. Towles, "Route Packets, not Wires: On-Chip Interconnection Networks", *Proceedings of DAC 2001*, pp. 684-689.

[31] J. Duato, S. Yalamanchili, L. Ni., "Interconnection Networks: An Engineering Approach", *Morgan Kauffman*, 2002.

[32] S. Felperin, P. Raghavan, E. Upfal, "A Theory of Wormhole Routing in Parallel Computers", *Proceedings 33rd IEEE Symposium on Foundations of Computer Science*, 1992, pp. 563-572.

[33] R. I. Greenberg, H. Oh, "Universal Wormhole Routing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no . 8, March 1997.

[34] W. J. Dally, H. Aoki, "Deadlock-Free Adaptive Routing in Multicomputer Networks using Virtual Channels", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 4, April 1993, pp. 446-475.

[35] R. V. Boppana, S. Chalasani, "A Comparison of Adaptive Wormhole Routing Algorithms". *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, 351-360.

[36] OPNET Modeller Website: http://www.opnet.com/products/modeler/home.html.

[37] R. Rajkumar, "Synchronization in Real-Time Systems: A Priority Inheritance Approach", Boston, *Kluwer Academic Publishers*, 1991.

[38] A. Erramilli et. Al., "Self-Similar Traffic and Network Dynamics", *Proceedings of the IEEE*, vol. 90, no. 5, May 2002, pp. 800-819.

[39] P.P. Pande, C. Grecu, M. Jones, A. Ivanov, R. Saleh, "Evaluation of MP-SoC Interconnect Architectures: a Case Study", *IWSOC 2004.*

[40] P.P. Pande, C. Grecu, M. Jones, A. Ivanov, R. Saleh, "Performance Evaluation Design Trade-offs for Network on Chip Interconnect Architectures", *accepted by IEEE Transactions on Computers.*

[41] SNNS Neural Network Simulator Website, *http://www-ra.informatik.uni-tuebingen.de/SNNS/.*

[42], *cnet* network simulator (v2.0.9), *http://www.csse.uwa.edu.au/cnet/,* The University of Western Australia.

[43] Scalable Network Technologies (SNT) Website, *http://www.scalable-networks.com/.*

[44] Cornell University, REAL 5.0 Network Simulator Website, *http://www.cs.cornell.edu/skeshav/real/overview.html.*

[45] Maryland Routing Simulator (MaRS) Version 2.0 Website, *http://www.cs.umd.edu/projects/netcalliper/software.html*

[46] R.V. Boppana, S. Chalasani, and J. Siegel, Wormhole Network Simulator, *http://www.cs.utsa.edu/faculty/boppana/papers/.*

[47] Simured Multicomputer Network Simulator Website, *http://tapec.uv.es/simured/index_en.html.*

[48] ITSW 20001 Documents.

[49] D. Wingard, "MicroNetwork-Based Integration for SoCs", *Proc. DAC 2001*, pp. 673-677, Las Vegas, Nevada, USA. June 18-22, 2001.

[50] MIPS SoC-it Website, *www.mips.com.*