OPTIMIZATION OF COMMUNICATION TRAFFIC IN HAMMER PROTOCOL USING 3D ELECTRONIC MESH NETWORK ON CHIP

by

Harishankar Suresh

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical and Computer Engineering

Charlotte

2016

Approved by:

Dr. Tao Han

Dr. Asis Nasipuri

Dr. Arun Ravindran

ABSTRACT

HARISHANKAR SURESH. Optimization of communication traffic in hammer protocol using 3D ELECTRONIC MESH NETWORK ON CHIP
(Under the direction of DR. TAO HAN)

In 1980s a processor had only one core, whose performance was improved by increasing the processor frequency. But this caused overheating and thus improvement in performance was halted. To get more performance multi core processor was introduced that gave better performance. For a chip multiprocessor with hundreds of cores, the cache coherence communication affected the memory access time. The goal of this work is to improve scalability in System on Chip(SoC) multi core processors by using the Hammer cache coherence protocol to maintain cache coherency and reduce the communication traffic by using 3D Mesh Network on Chip as the interconnection network. The GEM5 open source simulator is modified as a part of this thesis to simulate the 3D Mesh NoC. The 3D NoC is observed to produce 11% reduction in network traffic than the 2D NoC for the Hammer cache coherence protocol and is predicted to provide better performance for higher number of cores and thus improving the scalability of the system.

This research is organized into three sections. In the first section, the discussion of different types of cache coherence protocols are discussed and the selection of Hammer protocol for scalability is explained. In the second section, the discussion of various Network on Chips are discussed. In the third section, the implementation in GEM5 and the network algorithm is discussed.

ACKNOWLEDGMENTS

I would like to thank my late advisor Dr. Bharatkumar Joshi, for his guidance and motivation, without which, this thesis would not have been possible. His teachings has always been useful and gave encouragement.

It would be my duty to extend my gratitude to the committee members Dr. Tao Han, Dr. Assis Nasipuri and Dr. Arun Ravindran for taking time to be on my committee and assess my work.

I am thankful to EPIC (Energy Production and Infrastructure Center) for its generous support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Many core processor era

Towards the end of the 20^{th} century, the single core processor dominated the market. The Intel Pentium 4 [1] and the AMD Athlon 64 FX-55 [2] were some of the prominent processors used. But they all used very high clock speed nearly 3 GHz to increase processor performance. To increase the performance further, the scalability of the processor has to be increased. Many previous research have focussed on scalability with respect to improving the processor performance with less dense circuit technologies [3]. With the growth of multi core processors in the market, the research was focussed on the problems with implementing many core processors. The shared memory architecture [4] and the distributed memory architecture [5] were studied and developed for experimentation. Various algorithms were developed for mutual exclusion and barier synchronization in shared memory parallel programs [6]. Even though a lot of software optimizations were required to utilize the many core processor to the full extent [7], the system as a whole gave better performance with lesser clock frequency and thus reducing the power and heat dissipation which was a bottleneck for single core processor.

Present day servers and supercomputers need very high performance. To meet the performance requirements, processors that have tens to hundreds of cores are required. The fastest supercomputer in the year 2016 TIANHE-2 (MILKYWAT-2) has 3,120,000 cores distributed in multiple intel Xeon processors [8]. As the number of cores increased, the communication between the processors increased [9]. For scientific workloads, the cache coherence messages were observed to be an overhead for the system as the multi core processors were placed on different chips. So for

every cache miss, the cores had to communicate between the chips using the front side bus which is now replaced by the Quick Path Interconnect or the HyperTransport protocol [10]. This inter chip communication is expensive and thus the chip multi processors were introduced.

In the chip multiprocessor, all the cores are implemented in a single chip and the communication between the cores are implemented on an interconnection network. For example, in the Intel Developer Forum 2006, a 80 core processor prototype was demonstrated. The on chip implementation reduces the latency of memory access as the link delay on a chip is lesser compared to the front side bus [11]. To maintain cache coherency between the cores, the cores follow a cache coherence protocol in which coherence messages are exchanged between the cores through the interconnection network. When the core count increases, the coherence messages on the network increases and the network becomes a bottleneck.

In the chip multiprocessor with hundreds of cores, the communication overhead between the processors reached a saturation and thus reducing the scalability of the system [12]. With present day systems requiring hundreds of cores, the research was shifted to focussing on the interconnection network between the cores for supporting the cache coherence traffic. The bus interconnection network was optimized to support large amount of processors, but they had a bandwidth limit and arbiter delay which slowed down the system [13]. Thus Network on Chip was proposed to provide better bandwidth and there is no need of arbitration to send messages. The research in NoC have been done in the last few years, but they are mainly focussed on the conventional directory based protocol implemented for cache coherence [14]. In our thesis research, we will be focussing on the Hammer cache coherence protocol, which is more scalable than conventional protocols.

1.2 Objective

This thesis discusses the methods to support hundreds of cores on a single chip and thus improving the scalability of the system. The first goal is to identify the scalable cache coherence protocol for on chip multiprocessor. The second goal is to discuss different types of interconnection network to reduce the communication overhead so that the scalability of the system is improved. Network on Chip is an emerging technology that enables the communication network to be implemented as a part of the System on Chip. Since this technology has not come up in the market [15], we will be using GEM5 simulator for simulation. The GEM5 has cache coherence protocols and 2D Mesh NoC simulation package and being an open source simulator we will modify the simulator for implementing the 3D Mesh NoC. The NoC traffic generated for the scalable Hammer protocol is observed to be 30% more than the efficient but non scalable directory protocol and by implementing the 3D NoC in GEM5, our goal is to reduce the traffic overhead.

1.3 Thesis Motivation

A lot of work have been done in the multi core architecture field. But with the present System on Chip and Network on Chip technology, the scalability of multi core processors is not researched with the most scalable cache coherence protocol. The cache coherence is the main criteria that affects the memory access latency. The processor speed is higher than the memory speed and this gap in performance is one of the main research interests for improving system performance. This acts as the motivation for this thesis.

In chip multiprocessor, the cache coherence protocol and the Network on Chip are responsible for the memory delay. The use of directory based cache coherence protocol in 3D NoC have been simulated in previous research [16]. But they did not focus on the high on chip area overhead of the directory protocol and thus not providing a complete scalability solution. This research overcomes this problem by

use of the Hammer cache coherence protocol with the 3D Network on Chip. This produces a system in which the memory traffic is reduced by using a 3D NoC and producing almost the same NoC traffic that is produced in the Directory protocol based implementation.

There have been research going on the 3D NoC routers [17] and the implementation in hardware. But 3D Network on Chip is still an emerging technology [15] and is predicted to reach the market in near future. In the implementation level, research is still going on, but a lot of simulators have been developed for predicting its performance. The main focus of this thesis is on scalability of multicore processors in System on Chip. The hammer cache coherence protocol for distributed systems is chosen as it is scalable and the communication traffic is proposed to be reduced. The main disadvantage of this protocol is the high communication overhead and thus leading to higher power consumption. Since this protocol uses a Network on Chip for communication between the cores, the research focusses on network on chip and how to reduce communication overhead. Thus, the hypothesis is:

The scalability of multicore processors will be improved when using a non uniform cache memory, uses the hammer protocol for cache coherence and a 3D Mesh Network on Chip to reduce communication overhead.

1.4 Contribution

The 3D Network on Chip is still an emerging technology, so much of the research in this area is focussed only on the communication network. In this thesis the scalability of System on Chip scalability with respect to the cache coherence protocols is discussed. The following are the major contributions in this research:

- The Hammer Protocol was chosen as the scalable protocol due to its low on chip memory area overhead. The previous research was focussed on using the Directory based coherence protocol, which occupies more on chip area and thus reducing scalability.

- On executing the simulation for our system, the Hammer protocol produced consistently 30% more traffic compared to Directory Protocol in 2D Mesh NoC. Since the Hammer protocol is a broadcast based protocol, the extra traffic is produced and this traffic increases with increase in number of cores.

- 3D Mesh NoC was proposed to reduce the extra traffic caused by the Hammer Protocol. The 3D NoC was chosen as the net diameter of the network is reduced and the amount of traffic per instruction is reduced compared to the 2D NoC.

- GEM5 computer architecture simulator was modified to implement 3D Mesh NoC. The GEM5 simulator was chosen as it is an cycle accurate simulator that had the Hammer and Directory protocol implemented, and also supported 2D Mesh NoC.

- The 3D Mesh NoC was observed to reduce traffic by 11% for 64 core count X86 processor and the amount of traffic reduction increases with increase in number of cores. Since the GEM5 simulator can simulate only upto 64 cores, the amount of traffic reduction is predicted to reduce with higher number of cores and thus improving scalability of the system.

- Thus the many core System on Chip can be made scalable by using the Hammer Cache Coherence Protocol and 3D mesh Network on Chip for high traffic workloads.

1.5 Organization of Thesis

This thesis is organized into five chapters. The chapter 1, gives the pre requisite knowledge for this research. The literature survey and research done is presented in the chapter 2. The implementation and the design of the GEM5 software is discussed in the chapter 3. The results obtained from the simulation and observations is presented in charpter 4. The conclusion and summary of the research is written in the chapter 5.

CHAPTER 2: BACKGROUND

This chapter discusses the basic topics that are necessary to understand the implementation of this thesis. The cache coherence protocols and different types of Network on Chips,its implementation, are discussed focussing on the scalability of multi-core processors. For those familiar with these topics can proceed to the Section 3.

2.1 Multi Core Architecture

In the 1980s single core processors was implemented in which as the frequency of switching increased, the dynamic power consumed increased leading to heat dissipation [18].

$$Power_{dynamic} \propto 1/2 \times CapacitiveLoad \times Voltage^2 \times FrequencySwitched$$

Thus the performance hit the power wall at around the year 2002 as shown in figure 2.1. Despite innovations like pipelining and Dynamic Voltage-Frequency Scaling (DVFS), the performance of a single core processor was not enough. So the multi core processor was introduced into the market, which gave better performance with less frequency of switching. There are two types of multi core architecture based on the memory between the cores:-

- Shared memory multi core processors

  In shared memory multi core processors, the cores have their caches private and share the memory through the interconnect bus as shown in figure 2.2. As the number of cores increased, the bus became the bottleneck for the system and thus the distributed memory system was introduced.

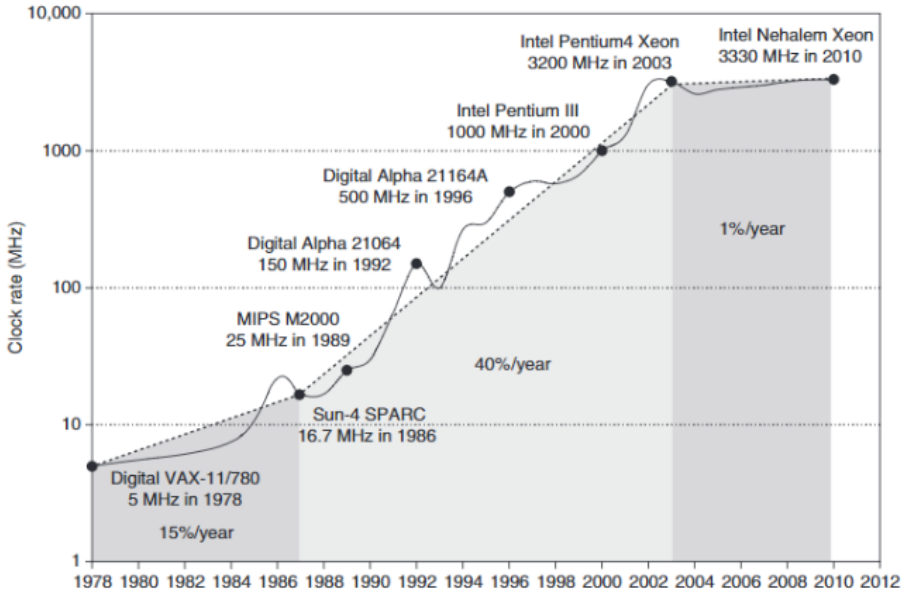

Figure 2.1: **Growth in clock rate of microprocessors [18].**The processor clock frequency is increased from 5 MHz in 1978 to 3000 MHz in 2004. After 2004, the clock speed could not be increased due to the heat generated. Thus the processor speed met the power wall in 2004 and the processor performance did not improve until the multi core processor was brought in the market.

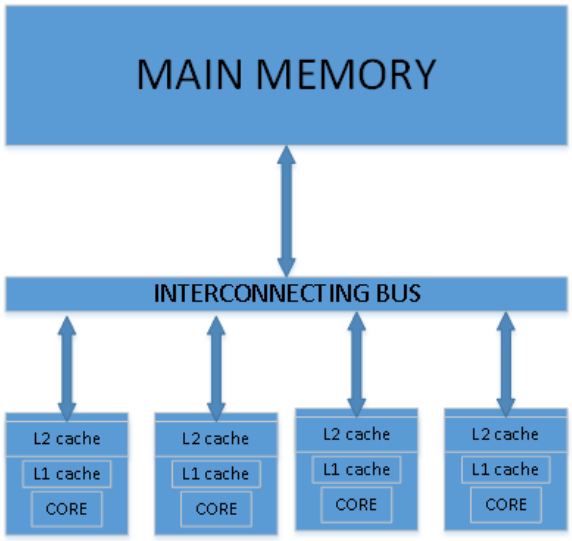

Figure 2.2: **Multi-core chips with shared memory [19].**In shared memory system, all the processors shared a common main memory using a bus architecture. The provides the indirection in the system to avoid race conditions, but has a upper limit for the bandwidth and arbiter delay.

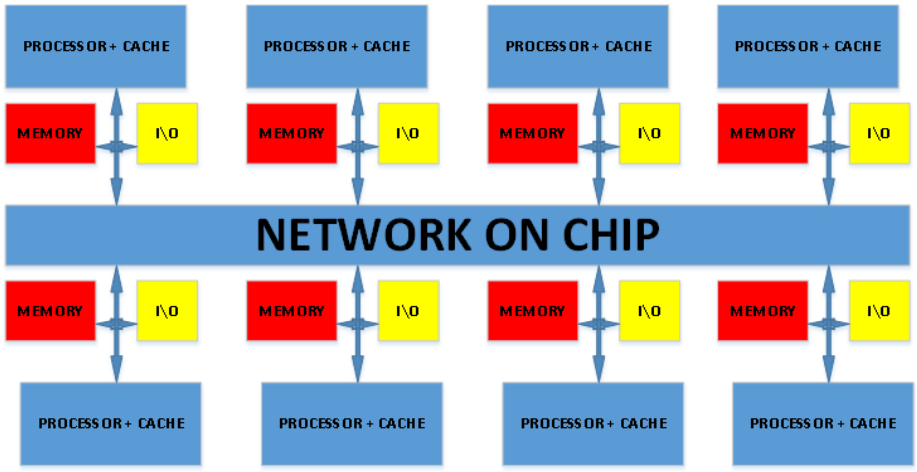

Figure 2.3: **Multi-core chips with distributed memory [4].** In the distributed memory architecture, each core has a physically separated memory, but they are logically shared between the cores. The interconnection network should support the point to point communication between the cores for using this architecture. Even though this architecture uses more hardware, it is more scalable compared to shared memory multiprocessors.

- Distributed Memory multi core processors

  In the distributed memory system, the memory is distributed between the cores, but they are logically shared between the cores as shown in figure 2.3. The interconnect is used to access the memory of other cores.

2.2 System on Chip

According to Moores law [20] the transistor density increased as shown in figure 2.4, which led to the development of System on Chip (SoC). System on Chip provides a very efficient solution, which incorporates all the processor cores and reusable components like L1 cache, L2 cache, the Network Interface and the network on chip/bus on one single chip as shown in figure 2.5. With the invention of 3D stacked ICs, the implementation of multiple layers of SoCs are possible.The SoC helps to reduce the I/O count, system noise, power, EMI, and cost, and to increase performance [21].

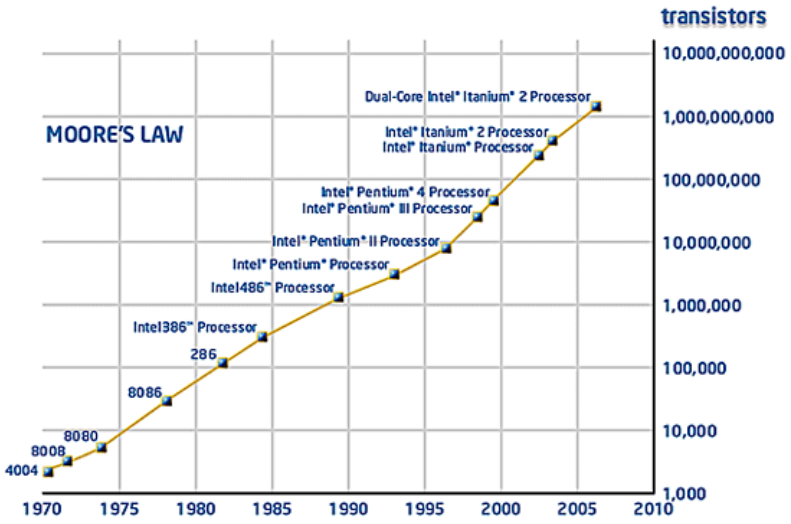

Figure 2.4: **Transistor integration density per die [20].** According to Moores law, the transistor density per square inch doubles every year. This diagram shows the transistor density growth in Intel processors from the year 1970 to 2010, and this increase in trend is predicted to grow in the following years.

2.3 Tiled Chip multiprocessor

As the number of cores increases, the system should be able to support the scalability, which includes the memory organization, cache coherence protocol and the communication network. The tiled Non Uniform Cache Hierarchy (NUCA) provides the cores to virtually share distributed memory. Here each tile consists of a CPU core, Private L1 cache,logically shared L2 cache, router to communicate with the network and directory for storing cache coherence information as shown in figure 2.6. The system must be able to support the increase in traffic, inter core communication, heat dissipation and synchronization. The system should also implement routing algorithms to avoid deadlock, livelock, starvation and aliasing [22]. In this thesis we will be focussing on optimizing the communication overhead for kilo core processors.

2.4 Symmetric Multiprocessors vs Non Uniform Memory Access

Symmetric Multiprocessing (SMP) is the implementation of shared memory with uniform access time and bus interconnect. Cache Coherent Non Uniform Memory Access (cc-NUMA) is the implementation of distributed memory with non uniform

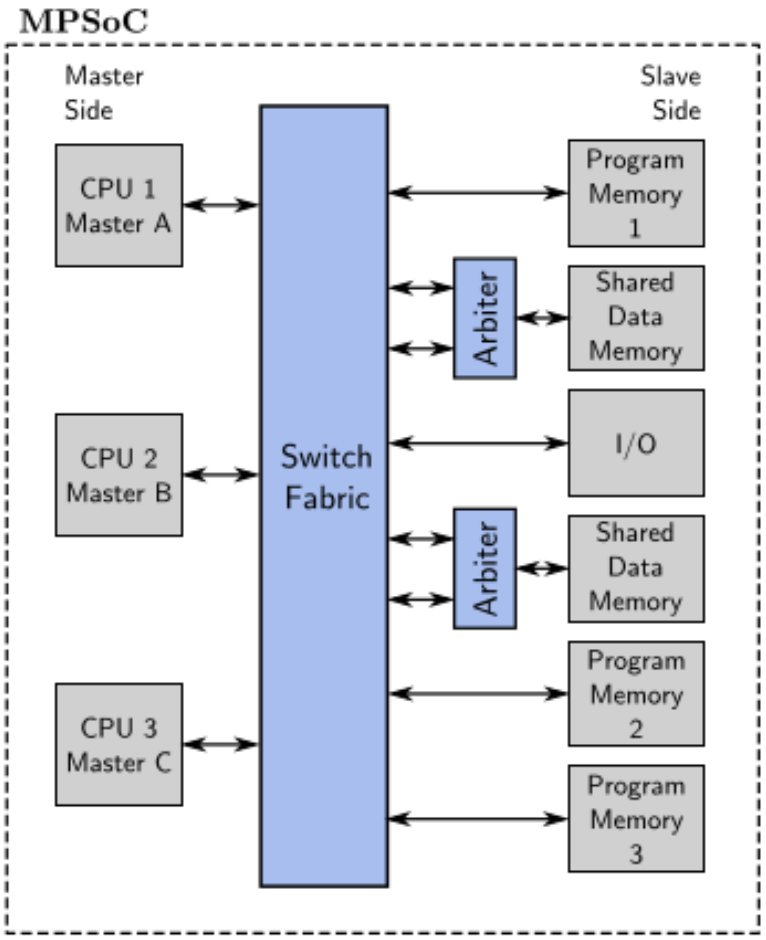

Figure 2.5: **System on Chip organization [21].** The System on Chip uses the high transistor density to incorporate the full system on a single chip. The system consists of the CPU cores, cache memory, the interconnection network/switch fabric and the main memory of the system.

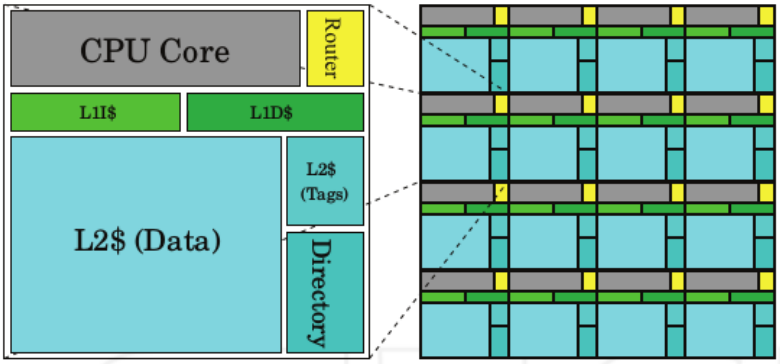

Figure 2.6: **Tiled chip multiprocessor organization [22].** In the Tiled chip multiprocessor, the CPU core, private L1 cache, logically shared L2 cache, the directory and the router are on a tile, and many tiles put together forms the chip. The L2 cache is accessible to all the cores through the network whereas the L1 cache is private and accessed only by one core.

access time and scalable interconnect [4]. The table 2.1 compares the two architectures based on scalability and programmability.

As we are focussing on the scalability in this thesis work, Non Uniform Memory Access architecture is preferred. It provides more scalability than symmetric multiprocessors provided we implement a scalable point to point interconnection network to communicate between the nodes.

2.5 CPU and memory Performance Gap

In the single core processor, as the frequency increased, the processor speed was improved. Micro-architecture level changes became prominent in the beginning of the 21^{th} century. Pollacks's rule[1] [24] have led to the successful introduction of multi-core processors in the market. It is also said that the micro-architecture improvement lead to two times improvement in performance while maintaining the same power consumption. This had led to the processor performance improvement with lower frequency and power consumption.

But the memory access speed only had little improvement for the single core processor as shown in figure 2.7. The processor line shows the increase in memory

[1]Pollacks's rule states that the micro-architecture causes performance improvement at the rate of square root of increase in complexity.

Table 2.1: Comparison between Symmetric Multiprocessor and Non Uniform Memory Access [4]

Field Name	Symmetric Multi Processors	Non Uniform Memory Access
Key Value	Less scalable	More scalable
Programmability	Easily programmable	Requires substantial application in order to handle capacity and conflict misses well
Scheduling policy	Easy to schedule processes	Requires operating-system optimization and scheduling algorithm is not trivial
Real Communication workload performance	SMPs are also more efficient in handling communication misses , which are common in commercial applications	They are less optimal for access patterns caused by "real" communication, such as producer-consumer and migratory data [23]

requests per second on an average [2], while the memory line shows the increase in DRAM accesses per second [3]. Initially to access a memory it was expected to approach 100 clock cycles. But there have been steady improvement from the year 1990. The faster SRAM cache memory was used to provide better hit time and thus faster memory, but it was not enough to match the processor speed which had an exponential performance rate. Various workloads have been run to show that the memory accesses became the bottleneck of the single core system [25].

With the growth of multi core processors, the memory was distributed among the processors and thus a faster hit time was obtained, but this architecture had a new memory overhead which is maintaining the cache coherence and memory consistency between the cores [26]. The main reason for slow memory in multi core processors is the overhead to maintain cache coherency in the system. Various software and operating system support [27] was researched for improving the data locality and the cache misses, but they could not provide enough improvement to meet the processor speed. Various protocols have been developed for maintaining cache coherency and the research now focusses on the communication network between the cores for efficient communication [28].

2.6 Cache Coherence protocol

With the increase in number of cores, the private cache for each core have to be coherent with each other. Various coherence protocols for shared and distributed memory are used for maintaining coherence. Bus-Snooping cache coherence protocol is a broadcast based protocol using a bus architecture used in symmetric multi processors. Each processor snoops the bus for shared data. The protocol uses write update or write invalidate policy. But as the number of cores increased, the bus became the bottleneck of the system [29]

So different protocols for tiled chip multi processor was introduced:-

[2]The inverse of the latency between memory references
[3]The inverse of the DRAM access latency

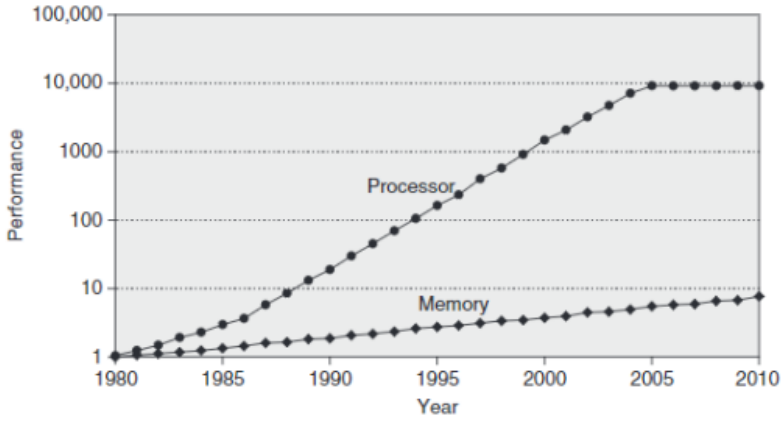

Figure 2.7: **CPU and memory performance gap [18].** Till 1985, the processor speed and the memory speed has little difference. But as the research in processor speed grew, the performance gap increased and the memory became the bottleneck of the system. This memory and CPU performance gap is a major motivation for research in industries.

2.6.1 Hammer CMP protocol

This is the protocol used by AMD in their Opteron systems [22]. This protocol like snooping based protocol does not store any information about the blocks in private cache and is based on broadcasting misses to all the tiles. The Hammer protocol is targeted for unordered point to point interconnection networks. The ordering point in this network is the home tile.

The protocol does not store any block information which implies that the area overhead is constant with respect to the number of cores.Thus it is more scalable compared to the directory protocol. But the protocol has one level of indirection through the home tile and thus requires three hops for the requestor to get the data block as shown in figure 2.8. Also, due to the broadcasting of invalidation messages there is a lost of traffic in the network thus increasing energy consumption.

2.6.2 Directory CMP protocol

The directory protocol is similar to the intra-chip protocol used in Piranha [22]. This protocol has a directory that keeps track of each cache block in the system. This allows the protocol to send invalidation messages only to the tiles that share

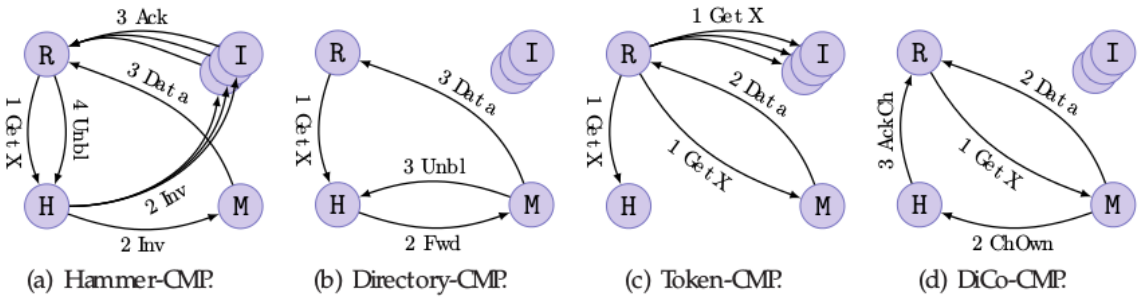

Figure 2.8: **A Cache-to-cache transfer miss [22].**

the block, thus reducing unnecessary communication overhead. In this protocol also, ordering is maintained by indirection to the home tile.

The Directory CMP also requires three hops as there is one level of indirection through the home tile as in figure 2.8. More directory memory is compensated by low amount of network traffic. As the number of cores in the system increases, the directory in each core grows the area increases in the order of $O(N)$, where N is the number of cores and thus this protocol does not support scalability.

2.6.3 Token CMP protocol

Token coherence maintain cache coherence by assigning a fixed number of tokens to each memory block [30]. A processing core can read a block only if it has at least one token and can write only of it has all the tokens with it. On every cache miss the requesting core broadcasts cache miss to every other tiles as in figure 2.8. In case of a write miss, they have to answer with all the tokens that they have. The data block is sent along with the owner token.

Since this is also a broadcast based protocol, this produces traffic in the network. Its area overhead grows in the order of $O(log(N))$, where N is the number of cores. Thus this reduces indirection, but does not support scalability.

2.6.4 Direct Coherence CMP protocol

In direct coherence protocol, the ownership is changed according to the owner of the data [31]. The ordering point of the request of a particular memory block is the

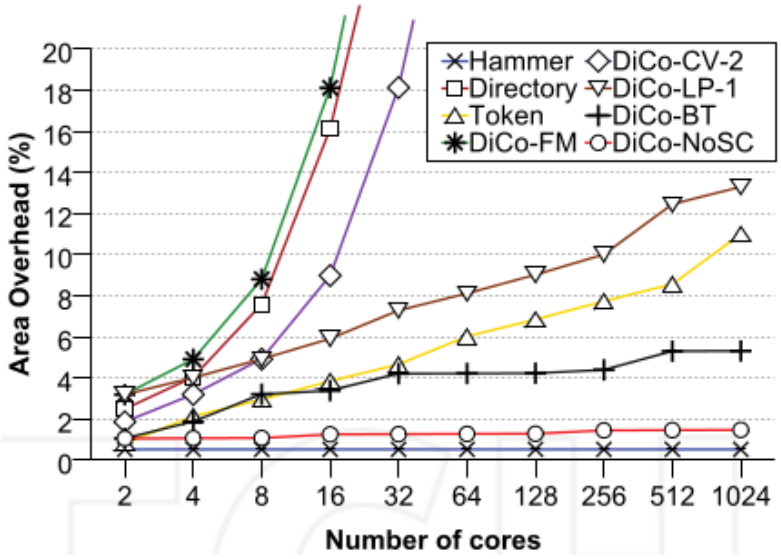

Figure 2.9: **Area overhead for cache coherence protocols [22].** The percentage of on chip area required for each cache coherence protocol is plotted in this graph. With the increase in number of cores, it is observed that the Hammer protocol uses the least area whereas the directory protocol has exponential increase. Thus the directory protocol does not support scalability of chip multiprocessor.

current owner tile. In this way the request block is obtained by the requester in two hops as in figure 2.8. Then the ownership change is sent to the home tile, which sends an acknowledgement to the requestor.

The traffic in this protocol on the network is very less, but this protocol has to maintain a directories for each cache block and thus result in area requirement of the order of $O(N)$, where N is the number of cores in the system.

2.6.5 Comparison between the protocols

Thus on observing the protocols, the area requirement grows as shown in as shown in figure 2.9. It is observed that hammer protocol uses the least memory/area overhead compared to other protocols. Its main disadvantage is the high message traffic on the network and thus causing higher energy consumption. In the next section we can see how to deal with this problem by considering different types of Network on Chips.

Table 2.2: Comparison between BUS and Network on Chip [32]

BUS			Network on Chip
Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.	-	+	Only point-to-point one-way wires are used, for all network sizes, thus local performance is not degraded when scaling.
Bus timing is difficult in a deep submicron process	-	+	Network wires can be pipelined because links are point-to-point.
Bus arbitration can become a bottleneck. The arbitration delay grows with the number of masters.	-	+	Routing decisions are distributed, if the network protocol is made non-central.
The bus arbiter is instance-specific	-	+	The same router may be re-instantiated, for all network sizes.
Bus testability is problematic and slow.	-	+	Locally placed dedicated BIST is fast and offers good test coverage.
Bandwidth is limited and shared by all units attached.	-	+	Aggregated bandwidth scales with the network size.
Bus latency is wire-speed once arbiter has granted control.	+	-	Internal network contention may cause a latency.
Any bus is almost directly compatible with most available IPs, including software running on CPUs.	+	-	Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems.
The concepts are siaple and well understood	+	-	System designers need re-education for new concepts.

2.7 Network on Chip

Chip Multiprocessor (CMP) architectures integrate tens of processor cores onto one chip. As the number of cores increases, a shared interconnect between the cores becomes impractical and thus the cores has to be connected through unordered point to point networks as shown in table 2.2. When a Chip Multiprocessor is implemented as a System on Chip, the Network on Chip provides the best scalable interface for communication between the processors.

2.7.1 Open Systems Interconnection Model of NoC

The basic protocol used for a Network on Chip is shown in figure 2.10 [32].

- The *Application Layer* consists of the resources of the system like microprocessors which does several tasks simultaneously.
- The *Presentation Layer* does the conversion between the resources between the CPU layer and the network layer like endian conversion or data type conversion.
- The *Session Layer* establishes connection between the resources using the transport layer.
- The *Transport Layer* ensures secure transmission of all the packets. It also deals with the segmentation of messages into packets and their reassembling. It also handles the flow control which highly affects the performance of the NoC.
- The *Network Layer* handle the routing mechanism of the network using the routing algorithm. The most widely used techniques for routing are circuit and packet switching. In the circuit switching algorithm the network control overhead incurs only once and this algorithm is best in case of persistent communication. Packet switching algorithm has distributed network control overhead and is a more energy efficient for irregular communication.

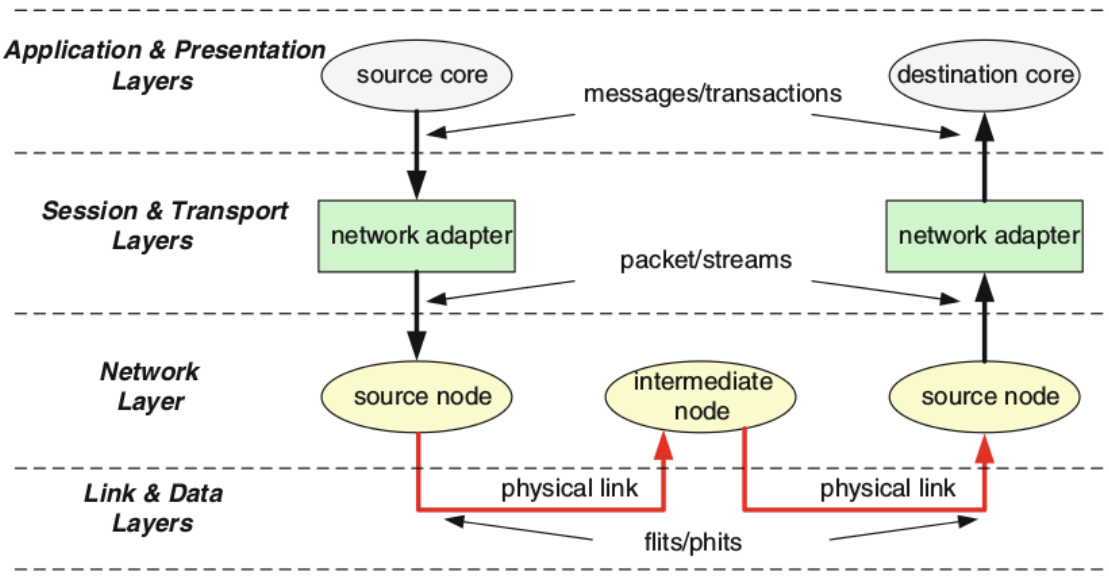

Figure 2.10: **ISO/OSI reference model for NoCs [33].**

- The *Data Link Layer* ensures reliable transfer of data by correcting errors in the physical layer. This may include retransmission of data in case of a failure.

- The *Physical Layer* refers to the driver information, the electrical circuits and the clocking mechanisms. Fault tolerant measures are introduced in this layer in case of a link failure.

2.7.2 Electronic Network on Chip

The complexity of designing Electronic NoCs with increasing number of cores is dealt with the tiles CMP architectures. Here we discuss the design of area-efficient and energy-efficient Electronics on chip networks for tiles CMP architecture [33].

2.7.2.1 Mesh Topology

The mesh topology is a symmetric topology that has equal number of links for every router. In this topology, every node has a router assigned to it as shown in figure 2.11. Every Interconnecting Processor core is connected to one router. Every router except those at the edges are connected to four other routers and one node. All the links are bidirectional. The 2D Mesh topology has all the links of the same

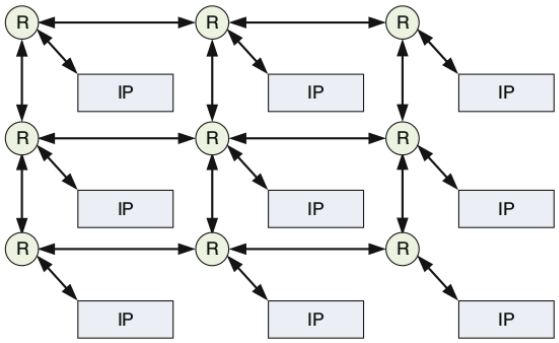

Figure 2.11: **2D Mesh Topology [35].** In a Mesh topology NoC, every core/IP is connected to one router. Each router has four neighbouring routers to form a Mesh structure NoC.

length which imposes a regularity in the topology. The area of the mesh topology grows linearly with the increasing number of cores [32]. Apart from these features, this topology has some drawbacks. The amount of routers in this topology cerates congestion in the middle of the network. So careful mapping of the workload has to be done.

This topology uses the O1Turn routing algorithm [34]. O1Turn algorithm follows the XY or YX routing algorithm, where the messages traverse the X or Y coordinate first and then traverses the other coordinate. Even though this causes a lot of congestion in the middle of the network, this algorithm is optimum for finding one of the shortest path between the nodes.

2.7.2.2 Torus Topology

The Torus Topology is generated by extending the Mesh Topology by adding end-around links at the edges as shown in figure 2.12. This provides uniform channel length at the expense of longer average channel length. Due to this the diameter of the network reduces as shown in Table 2.3. Also the congestion in the middle of the network is reduced as the messages can be routed around the edges. But the drawback in this topology is the high amount of area overhead as the number of cores increases.

Table 2.3: Comparison between NoC Topologies

Network	Nodes	Degree	Diameter	Bisection Bandwidth	Edge Length
2D Mesh	K^2	4	$2(K-1)$	K	Const
2D Torus	K^2	4	K	$2K$	Const
Binary Tree	$2^K - 1$	3	$2(K-1)$	1	Var
Hypercube	2^K	K	K	$2^K - 1$	var

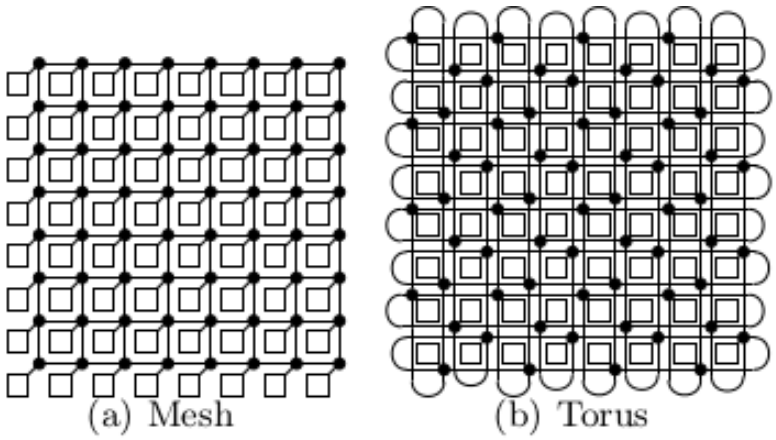

Figure 2.12: **2D Mesh and Torus Topology [37].** The Torus topology is a Mesh NoC with the ends connected and thus forming a cylindrical NoC. This reduces the network diameter, but increases the average length of the links and the area required for the NoC.

This topology uses the dimension order routing algorithm [36]. This follows the routing in one dimension at a time and according to the congestion in the network, the dimension is chosen.

2.7.2.3 Fat Tree Topology

In this topology, the topology is implemented such that the nodes of the tree are routers and the leaves are the processing elements [36]. The nodes above the leaves are called its ancestors and the nodes below the nodes are called its children. In a Fat Tree Topology, each node has replicated ancestors which provides multiple paths to move from each node to another a shown in figure 2.13.

The main disadvantage of the Tree Topology is that all the messages must route

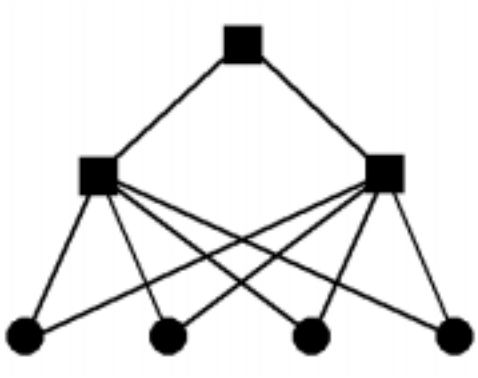

Figure 2.13: **Fat Tree Network [38].** The tree topology is implemented such that the nodes of the tree are routers and the leaves are the cores/IP. For the Fat tree topology, more links are used to connect multiple routers to the same node.

through the root of the tree, which makes it a bottleneck. This is avoided in Fat Tree Topology, but this requires extra links to be introduced in the system and thus it is not scalable.

2.7.2.4 Hypercube Topology

The hypercube has a cube like structure with the vertices as the nodes. The hypercube based topologies provide regularity, symmetry, high connectivity, small diameter and programmability as shown in figure 2.14. They appear as the most suitable ones for our image processing applications in MPSoC.

For a N node Hypercube the diameter of the network is $log_2 N$. The Hypercube Topology has good bisection width compared to other topologies and has logarithmic vertex degree of $log_2 N$. The drawback with this topology is its high number of routers and links which restricts it from supporting scalability.

2.7.3 Optical NoC

The Optical NoC uses optical telecommunication mechanisms for message transfer. It relies on the photonic technology instead of electrical signals. It provides better energy efficient communication and better performance compared to Electronic NoC. It provides low latency and high bandwidth and thus enhances the communication bounds present in Electronic signal communication [40].

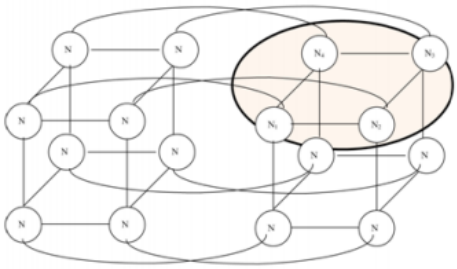

Figure 2.14: **Hypercube Network [39].** In a Hypercube topology, the vertices of a cube are taken as nodes, and all the vertices of a cube is connected to all the corresponding vertices of its neighbouring cubes.

The drawback of this system is that it is prone to thermal instability. With fluctuating temperatures, the systems becomes unstable. Presently ring resonators used for inter layer communication produces lots of losses, which causes interference in some environments. A lot of research is being done in this area for enhancement.

2.7.4 Wireless NoC

Wireless Network on Chip work on the principle of wireless transfer of messages. This creates a communication protocol where Network congestion can be avoided and thus provide less latency communication of messages. It depends on the *Transmitter/Receiver theory* and used antennas for communication [41].

The Wireless NoC has the drawback with the high area requirement for alignment of antennas. Another main disadvantage is that it has an upper bound on the bandwidth that can be sent over the network. It also uses high amount of energy if not used effectively.

2.7.5 Network Flow Control

Netowork Flow Control is the method by which message packets are transmitted inside the network [36]. The following modes are generally used in present day NoCs.

2.7.5.1 Store and Forward Routing

In Store and Forward Routing, the packets move in one piece and are stored in the routers memory. Then it gets transferred to the next router. So the memory of the router should be equal to the largest packet in the network. The Latency is the

time of receiving the packet, storing it and sending it again to the next router. A packet can be sent only after the whole packet has arrived to the router.

2.7.5.2 Virtual-Cut Through Routing

Virtual-Cut Through Routing is an improvement over the Store and Forward Routing technique. Here the packet can start transmission to next router as soon as the next router is ready to accept the packet. It need not wait for the full packet to arrive to the router. This method also need same memory as in Store and Forward technique but the latencies are reduced.

2.7.5.3 Wormhole Routing

In Wormhole Routing, the messages are divided into equal size flits. After the first flit is sent, the routing path is reserved to transmit the remaining flits of the message. This technique requires less memory compared to other two techniques as only one flit needs to be stored at a time in a router. Even though the latency is reduced, the probability of deadlock is higher in this routing technique. This can be avoided by introducing virtual ports by multiplexing one physical port. By doing this, the possibility of traffic congestion and blocking decreases.

2.8 3D Mesh NoC architectures

2.8.1 3D Mesh NoC

In a System on Chip, the on-chip interconnect is the principle bottleneck. The conventional 2D IC had limited floor planning choices and thus does not support the performance enhancements of the NoC. The three dimensional IC was introduced recently which contains multiple layers of active devices to connect layers using short vertical connections [42]. The NoC and 3D ICs together form the 3D NoC architecture as shown in figure 2.15. Using a cycle accurate simulator, it is shown that 3D Mesh NoC exhibits less execution and average latency than 2D Mesh NoC for fast fourier transform and matrix multiplication workloads [43].

With the development of 3D ICs, multilayer networks became prominent [44].

By introducing the 3D topologies, the network diameter was reduced, the length of the links is reduced and provides more scalability than 2D Mesh NoC. For the 3D Mesh NoC, the router configuration has to be changed from a 5 port router to a 7 port router. Even though this reduces the energy consumption of the system, this is compensated by the short links and thus provides lesser power consumption.

There have also been physical implementations of 3d NoC architectures and verified the working of the system.A 2 tier Tezzaron 3D technology to compare the 3D Mesh and 3D Stacked Mesh NoC have been implemented [45].The different types of 3D topology constraints are discussed and implementation in terms of wire delay, area and energy is observed [46]. Description on how to determine the best topology, compute paths and placement of the NoC components in each 3D layer is provided. 65 nm low power technology libraries are used for synthesis.

2.8.2 3D Stacked NoC

Through Silicon Vias technology have been developed for stacked NoC architecture. Staked NoC is a combination of bus based architecture and 3D NoC architecture where it uses a bus structure for inter layer communication as shown in figure 2.16. Even though the manufacturing of TSVs is difficult, it produces a faster interconnection mechanism between the layers of the NoC [47].

2.8.3 3D NoC in market

The research on 3D NoC development have been improving in the last 5 years. Although there have been experimental implementations for research in VHDL, it has not come up in the market due to insufficient 3D design and synthesis tools as well as the limitations of TSVs and extra manufacturing process such as wafer thinning of 3D IC [48]. This is yet to be generally accepted by the industry. The implementation paradigms for heat dissipation, floor planning, switch placement, NoC partitioning, router area and power consumption have been discussed in terms of cost and implementation.

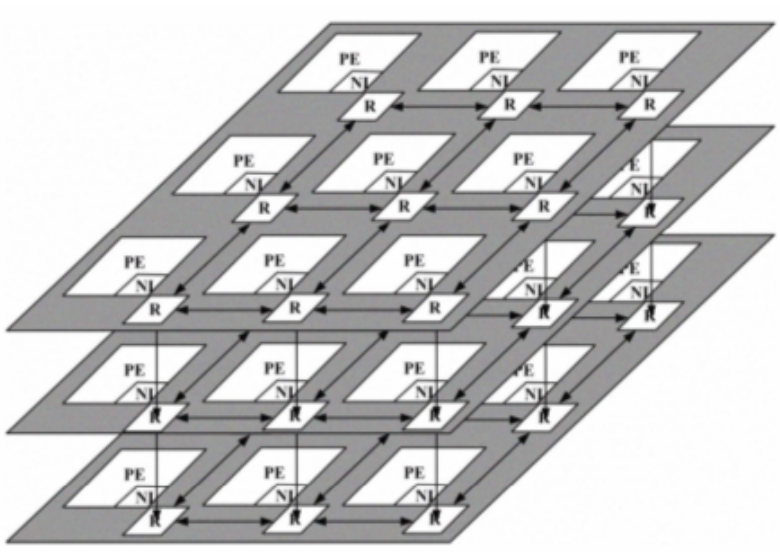

Figure 2.15: **The Mesh based 3D NoC architecture [15].** This is a three tier Mesh NoC, where three 2D Mesh NoCs are connected using links between the layers. Each router has seven ports, six for connecting to its neighbouring routers and one to connect to the processing element(PE).

There have been research going on specifically in the concentration of cost and implementation of 3D NoCs [15]. Various strategies for the router placement and implementation of hybrid routers[4] have been discussed to reduce the manufacturing cost and implementation. Even though 3D NoCs are not present in the market today, it is an emerging technology for scalable servers and supercomputers necessary for large scale multi core processors.

2.8.4 Analysis of 2D and 3D Mesh NoC

With the above mentioned analysis, the binary tree has low bandwidth and congestion at the root. Thus for increasing number of cores, the tree NoC is not scalable. Depending on the workload and the requirements of the system, the performance of each NoC is chosen. Since the scalability of the system is of prime importance in this research, we will analyse the NoCs with increasing number of cores.

The Torus topology uses a lot of wiring area with the increase in number of cores. The hypercube adds more ports to the router with increase in number of cores, and

[4]Hybrid router is a combination of 2D and 3D routers on the same chip

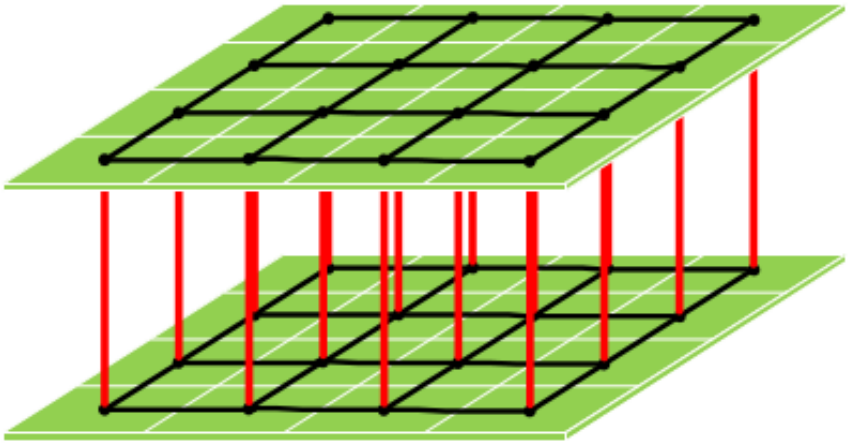

Figure 2.16: **A 2-tier stacked 3D NoC [15].** In a stacked NoC, the links between the layers are bus structures, where the messages are broadcast to reach multiple layers in one cycle. This reduces the inter layer communication delay and reduces the link count of the Noc.

thus does not support scalability. Thus the Mesh topology will be most suitable for scalability reasons. Even though the Mesh topology has the disadvantage that it has high worst case delay, when the Through Silicon Vias [5] is introduced, this drawback is overcome easily.

The comparison in terms of scalability has been done for the 2D and 3D Mesh NoC in [44]. The two NoCs are compared based on wiring area, energy dissipation, worst case delay and area for Through Silicon Vias. Wiring area is the percentage of wiring area required for the NoC compared to the whole System on Chip area. As shown in figure 2.20 the wiring area is observed to be very high for Torus NoC. As the number of cores increases, the percentage of area for Mesh and Torus decreases and becomes equal to the 2D Mesh NoC as shown in figure. This is because as the number oc cores increases, the System on Chip area increases more rapidly than the NoC area. So for high number of cores, the 3D NoC provides lesser network diameter and almost same percentage of area compared to 2D NoC. The energy dissipation is in nano

[5]Through silicon vias(TSV) is a bus structure that is used to connect the layers between the stacked NoC [49]. Using the TSVs the communication between the layers are broadcasted and thus take very less time for communication. TSVs are difficult to manufacture and occupy on chip area.

joules which is measured for the Network on Chip. As observed from the figure 2.19 the energy consumption in 3D NoC is lower as the network diameter is reduced and each message needs to spend lesser time on the network. Even though the energy of a 3D router is higher compared to a 2D router, the overall power consumption for Torus and Mesh NoC is reduced. For the Hypercube NoC, the number of router ports increases as the number of cores increases and thus the energy consumption is higher compared to other NoCs. With the introduction of TSVs, each TSV occupy considerable area on the chip. The on chip area occupied by the TSVs for Torus NoC is higher compared to Mesh and Hypercube as shown in figure 2.8.4. The TSVs provide a faster communication mechanism by broadcasting the messages to all the layers though the bus. This is used for quick communication between the layers of the NoC. The drawback with the Mesh 3D NoC compared to 2D is the worst case delay as shown in figure 2.18. The worst case delay is the maximum delay possible between two nodes in the system. As the number of cores increases, the delay increases exponentially. But this is being overcome with 3D stacked NoC and TSVs for the Mesh NoC. By implementing the stacked 3D NoC and TSVs, the length of the links gets shortened and the messages between the layers gets transmitted easily. Thus 3D Mesh NoC is a good performer considering the four factors namely wire delay, energy consumption, TSV area and usage of wiring area.

2.8.5 Router for NoCs

To implement the 3D router, we will first understand the working of a 2D router. Various router configurations have been introduced in the market, but the basic router configuration is the Hermes router [50] shown in figure 2.21. The router consists of the following signals:

- clock_ tx - Synchorizes data transmission
- tx - Indicates data availability

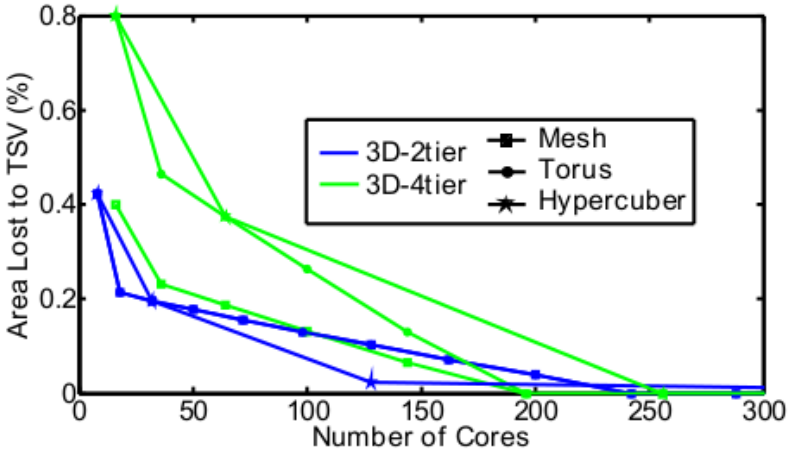

Figure 2.17: **Comparison of area lost to TSV in NoCs [44].** For the 3D fout tier plot, the Mesh topology requires less percentage of TSV area compared to Torus and Hypercube. The difference in TSV ares is high for the 3D-4 tier. For the 3D-2 tier, the difference is less, but the hypercube and Mesh requires the same area for higher number of cores. Thus the 3D requires almost same percentage of TSV area for higher number of cores

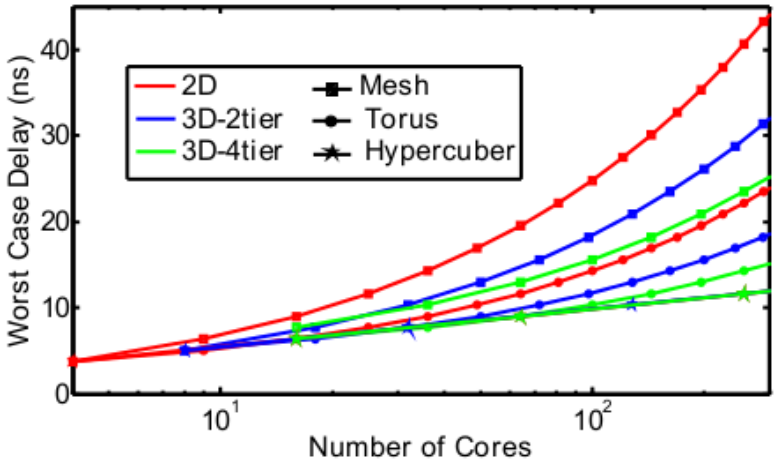

Figure 2.18: **Comparison of worst case delay in NoCs [44].** The worst case delay increases for the 3D-4tier plot with increase in number of cores compared to other 3D-4tire topologies. This is a drawback with the Mesh NoC, but with the implementation of Through Silicon Vias and 3D IC technology, this delay can be overcome.

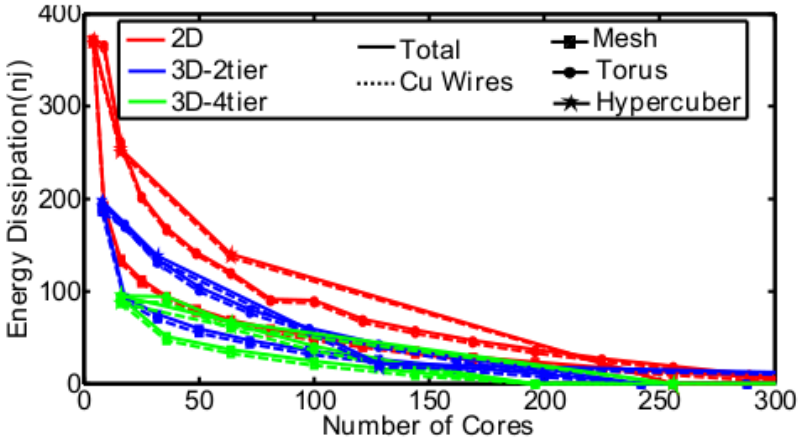

Figure 2.19: **Comparison of energy dissipation in NoCs [44].** For the 3D Mesh topology, the energy dissipated is least compared to the 3D Torus and Hypercube topologies. This is because the Mesh topology has smaller links and occupies lesser on chip area.

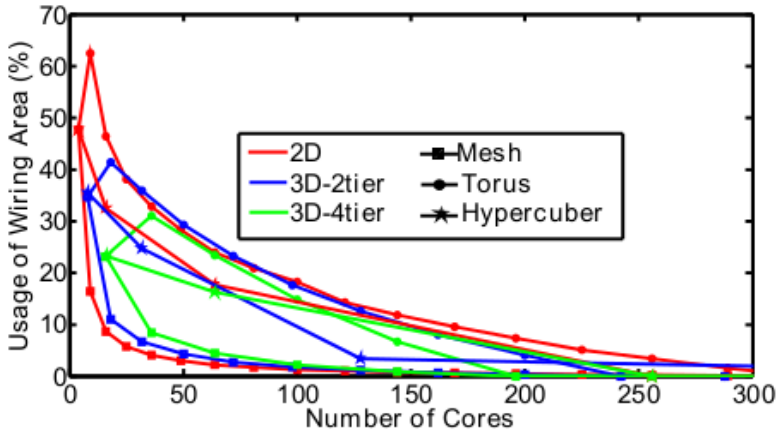

Figure 2.20: **Comparison of usage of wiring area in NoCs [44].** For the 3D-4tier Torus topology, the on chip area occupied is very high compared to other topologies. As the number of cores increases, the 3D-2tier and the 2D NoC occupies the same percentage of area as the percentage of NoC area becomes negligible compared to the total area. Thus 3D NoC occupies almost same area as 2D NoC for high core counts.

- lane_ tx - indicates the lane transmitting data
- data_ out - Data to be sent
- credit i - Indicated available buffer space for each lane

East, west, north and south are the two way ports for the router. Since this is a router for 2D Mesh NoC, it has four ports for connecting four routers and one port to connect to the core. Each port had n virtual lanes and thus n buffers. So each buffer can store upto d/n flits. As the flits are received in a lane, the credits are decremented and the flits are stored in the buffer indicated by $lane\_rx$.

When multiple packets come to the input port, round robin arbitration grants access to the packets. Once the incoming packet reaches the buffer, the XY routing algorithm is used to find the correct output lane. When the routing algorithm finds an available output port, the connection is established and the routing table is updated. An example routing table entry is shown in table 2.4 for the connection shown in figure 2.22.

In the routing table, three vectors namely in, out and $free$ are used. The in vector connects the input lane to an output lane. The out vector connects an output lane to an input lane. The free vector is responsible for modifying the output lane to free(1) or busy(0) state. The in and out vector use identifier (id) using the port number(np) and the lane number (nl) by the equation

$$id = (np \times numberoflanes) + nl$$

. The ports are numbered from 0 to 4 for the East, West, North, South and Local ports respectively. In the table the North port L1 lane is busy(free = 0) and is being driven by the input lane L1 of the West port(out = 2). The input lane L1 of North port is driving the output L1 lane of the South port(in = 6). Once the output lane is decided, the flit is removed from the input buffer and the number of credits is

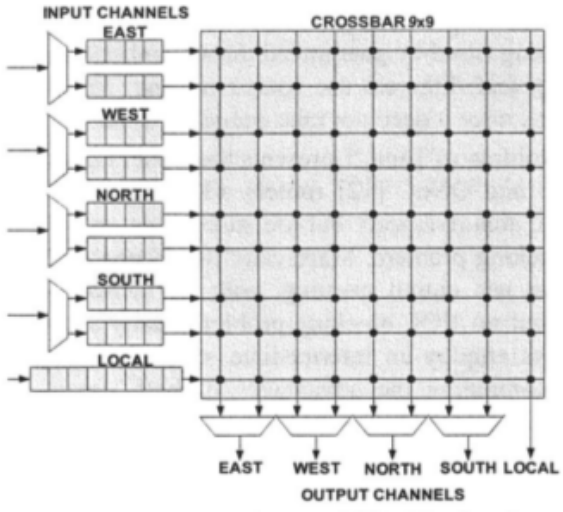

Figure 2.21: **Hermes Router with 2 virtual channels for 2D Mesh NoC [50].**

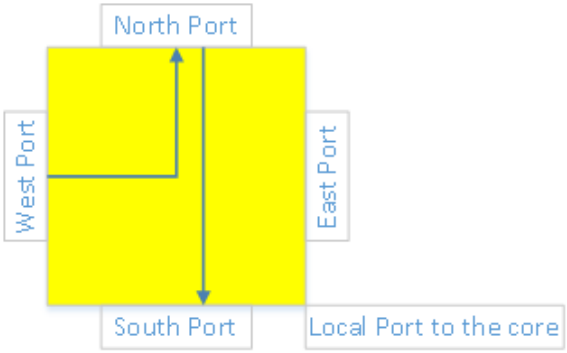

Figure 2.22: **A 2D Hermes router with connections established [50].**

incremented and the credit information is passed to the neighbouring router through *credit_o*.

In a 3D Mesh router, the number of ports increases by two to connect the top and bottom layers. This increases the number of switches as well as the power consumption of the router as shown in figure 2.23. To overcome this Through Silicon Vias was introduced to overcome the high energy consumption. The router placement on a NoC has various effects on the power consumption and there is a trade-off between performance and power consumption.

Table 2.4: Routing table entry [50]

Port	E1	E2	W1	W2	S1	S2	N1	N2	L
id	0	1	2	3	4	5	6	7	8
Free	1	1	1	1	0	1	0	1	1
In			4		6				
Out					2		4		

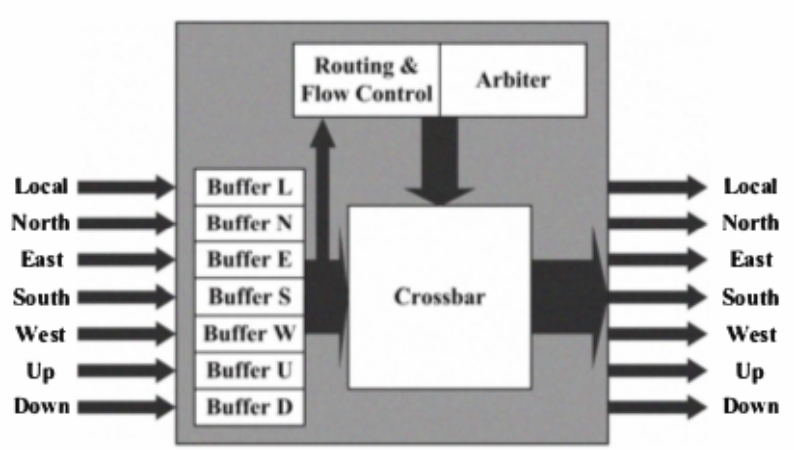

Figure 2.23: **Hermes Router modified into a 3D router [50].**

CHAPTER 3: DESIGN AND IMPLEMENTATION

3.1 Simulation Environment

So far we had discussed the necessary background to understand the theory behind this research. To implement this system, various simulation software are available. One of the efficient software is GEM5, which is a simulator platform for computer architecture simulation. This chapter discusses how this thesis is implemented in GEM5 by modifying the open source software for our requirements.

3.2 The GEM5 simulator

GEM5 is the merger of M5 and GEMS software. It supports various CPU architectures like ALPHA, ARM, SPARC and X86 processors. It supports an event driven model of simulation which includes caches, cache coherence protocols, network on chips and different I/O memory. It also provides Kernel Virtual Machine support which is used for binary translation. Its an open source software founded by collaboration from various institutes like University of Michigan, The University of Wisconsin Madison and various industries like ARM and AMD. GEM5 is based on command line interface and does not depend on the hardware it is installed in.

The GEM5 Ruby model supports three implementations namely Caches and memory, coherence protocol and inter connection network as shown in figure 3.1. The Ruby model works separate from the processors and thus provide flexibility to run any processor on any memory related simulation. The domain specific language Specification Language for Implementing Cache Coherence (SLICC) creates the state machines to connect the processor and memory system.

In GEM5, we have two models of CPU namely Atomic Simple CPU and Timing Simple CPU. The Atomic model does the memory access instructions atomically as

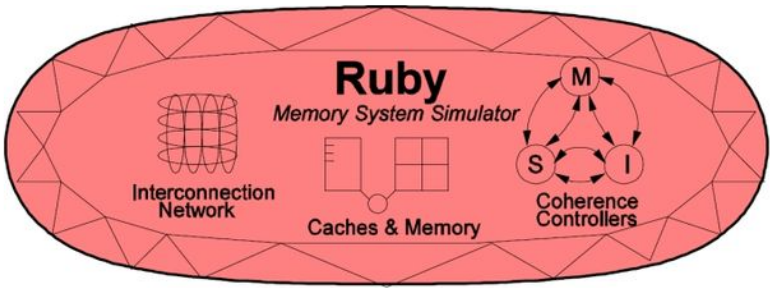

Figure 3.1: **GEM5 Ruby [51].**

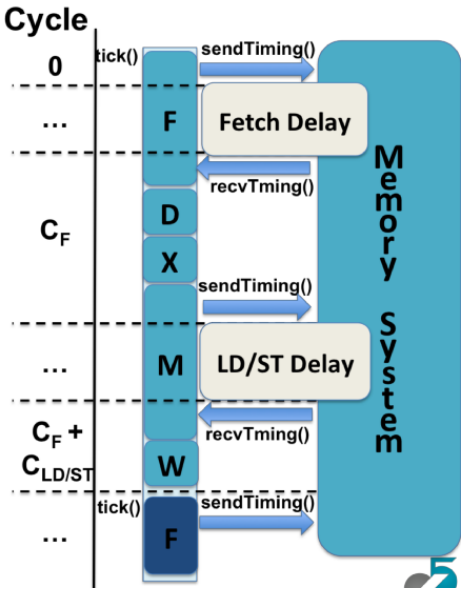

Figure 3.2: **CPU - Timing Simple [51].** In the Timing Simple CPU the sendTiming and recvTiming are used to depict the memory access. The CPU stalls till the memory access is finished.

shown in figure 3.3. In atomic mode the processor does not wait for the memory access to finish. It continues execution after sending the memory request to the memory system. Even though this method looks faster, a lot of dependency analysis has to be done beforehand to avoid stalls. The timing model waits for the every memory instruction to finish as shown in figure 3.2. Even though this method is independent of dependency analysis, there is lot of delay in the pipeline. Since this is a simulation, we will be able to observe the memory accesses clearly. Thus, here we will be using the Timing Simple CPU as we need the memory readings for our application.

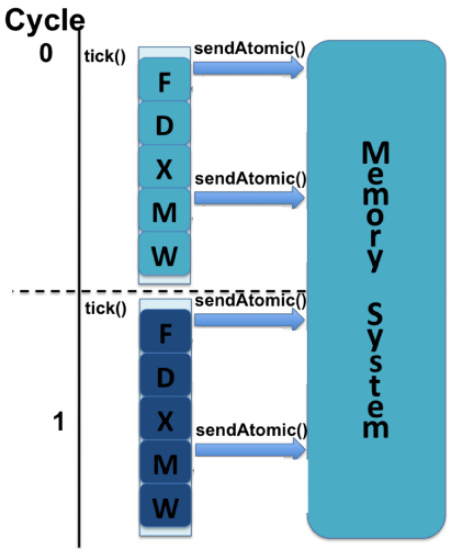

Figure 3.3: **CPU - Atomic Simple [51].** In the Atomic CPU, the CPU does not stall for the memory accesses and the memory access is done atomically and independently of the CPU pipeline.

GEM5 supports the MOESI Hammer CMP and MOESI Directory CMP based cache coherence protocols for single chip, 2 level caches. MOESI describes the different states of a cache block namely Modified, Owner, Exclusive, Shared and Invalid states. For the Hammer protocol, the cache hierarchy follows strictly-exclusive hierarchy whereas the Directory protocol follows the strictly-inclusive cache hierarchy. The states of the Hammer CMP protocol are explained in table 3.2 and that of Directory CMP protocol is explained in table 3.1. During a read operation, the cache block changes from modified or exclusive state to shared state. During a write operation, the cache block change from shared state to exclusive or modified or invalid state.

We will be using the GEM5 2D Mesh implementation for our thesis. This topology requires the number of directories to be equal to the number of cpus. The number of routers/switches is equal to the number of cpus in the system. Each router/switch is connected to one L1, one L2 (if present), and one Directory. It can be invoked from command line by –topology=Mesh. The number of rows in the mesh has to be specified by –mesh-rows. This parameter enables the creation of non-symmetrical

Table 3.1: States of cache blocks in MOESI Directory CMP protocol [51]

States	Invariants
MM	The cache block is held exclusively by this node and is potentially locally modified (similar to conventional "M" state).
O	The cache block is owned by this node. It has not been modified by this node. No other node holds this block in exclusive mode, but sharers potentially exist.
M	The cache block is held in exclusive mode, but not written to (similar to conventional "E" state). No other node holds a copy of this block. Stores are not allowed in this state.
S	The cache block is held in shared state by 1 or more nodes. Stores are not allowed in this state.
I	The cache line is invalid and does not hold a valid copy of the data.
MM_W	The cache block is held exclusively by this node and is potentially modified (similar to conventional "M" state). Replacements and DMA accesses are not allowed in this state. The block automatically transitions to MM state after a timeout.
M_W	The cache block is held in exclusive mode, but not written to (similar to conventional "E" state). No other node holds a copy of this block. Only loads and stores are allowed. Silent upgrade happens to MM_W state on store. Replacements and DMA accesses are not allowed in this state. The block automatically transitions to M state after a timeout.

Table 3.2: States of cache blocks in MOESI Hammer CMP protocol [51]

States	Invariants
MM	The cache block is held exclusively by this node and is potentially locally modified (similar to conventional "M" state).
O	The cache block is owned by this node. It has not been modified by this node. No other node holds this block in exclusive mode, but sharers potentially exist.
M	The cache block is held in exclusive mode, but not written to (similar to conventional "E" state). No other node holds a copy of this block. Stores are not allowed in this state.
S	The cache line holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. The cache line can be read, but not written in this state.
I	The cache line is invalid and does not hold a valid copy of the data.

meshes too [51].

To meet our needs, we have to understand the code implementation of 2D Mesh NoC and convert it into 3D Mesh NoC. Since GEM5 is an open source software, the documentation for the implementation of the 2D NoC was studied as described in appendix 5 and the modification of the code is discussed in the following sections.

3.3 GEM5 source code

GEM5 Ruby is implemented using python and C++ languages. As shown in figure 3.4 the python program provides the user interface objects and the C++ is used for the low level design of the system. The SLICC combines the memory and the CPU models in a state machine to run the system. The source code consists of configuration and implementation files. The configuration files are implemented in python and the implementation files are implemented in C++. Each file consists of a class implementing a modular programming structure.

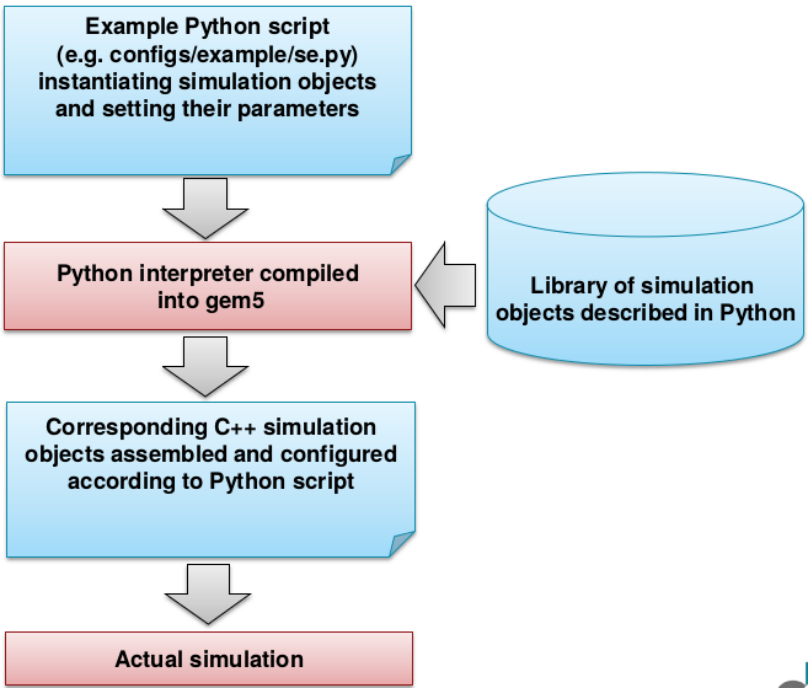

Figure 3.4: **GEM5 source code layout [52].**

3.4 GEM5 code documentation

For modifying the GEM5 2D NoC structure for our requirements, a proper code analysis of the implementation of 2D Mesh NoC was done. Since its a modular object oriented implementation, all the parts of the network is implemented as classes in separate Python and C++ languages. The Scons compiler was used to compile both types of files. This section briefly explains the top level implementation of the code that is pre implemented in GEM5 by the code developers. The analysis of every line of code for the pre implemented 2D Mesh NoC was done and the brief working of each class is described.

3.4.1 Link class

The link class uses two types of links namely ExtLinks and IntLinks. ExtLinks are used to connecting the router to the core and IntLinks are used for connecting the routers together. Each link class object contains the information for a link like the bandwidth, latency and weight of the link. For each link, an object is created and the

link details are updated. Each link had a corresponding link ID which is numbered based on the type of link. The ExtLinks going from core to router are numbered from 0 to $numberofcores - 1$. The internal links are numbered from $numberofcores$ to $2 \times numberofcores - 1$. The ExtLinks from router to core is numbered from $2 \times numberofcores$ to $3 \times numberofcores - 1$. See code 3.1 for the BasicLinkParams structure which shows the structure parameters of a basic link class.

Listing 3.1: Structure of a Link

```
struct BasicLinkParams : public SimObjectParams
{
        BasicLink * create();
        int bandwidth_factor; //Bandwidth
        Cycles latency;                    //Latency
        int link_id;
        int weight;
};
```

3.4.2 Message class

The Message class contains the parameters required for a message. It keeps track of the tick time, the last enqueued time, delayed ticks and the lane of the router link that the message has to be routed to as shown in the code 3.2. This class is used by the Messagebuffer class for enqueue and dequeue operations. An object of the message class is used as input and output for all the networking classes.

Listing 3.2: Private parameters of Message class

```
class Message
{
  private:
```

```
    const Tick m_time;
    Tick m_LastEnqueueTime; // my last enqueue time
    Tick m_DelayedTicks; // my delayed cycles
    uint64_t m_msg_counter;
     // Variables for required network traversal
    int incoming_link;
    int vnet;
}
```

3.4.3 MessageBuffer class

The MessageBuffer class keeps the information required for all the buffers that act as intermediate storage between the nodes and the routers. It keeps track of the available buffer size, the type of master and slave port connection and the order of extraction of messages from the buffer as shown in code 3.3. The Throttle class uses the MessageBuffer class to move the message from input vnet/buffer to output vnet/buffer(input and output buffers are passed and changes the bandwidth. For every iteration it checks if the previous message was sent and then pushes the message.

Listing 3.3: Parameters of MessageBuffer class

```
struct MessageBufferParams : public SimObjectParams
{
    MessageBuffer * create();
    unsigned buffer_size;     //Size of Buffer
    bool ordered;
    bool randomization;       //Order of message extraction
    unsigned int port_master_connection_count;
    unsigned int port_slave_connection_count;
};
```

3.4.4 SimpleNetwork Class

The basic network layout is done in the SimpeNetwork class. It uses the above defined Link, Message, Router and MessageBuffer classes to implement a simple network. The simple network models hop by hop network traversaland abstracts out the detailed implementation of of the switches. The switches are modelled in the PerfectSwitch class while the links are modelled in Throttle class. The flow control is implemented by monitoring the available buffers and available bandwidth in output links before sending. There are two types of routing that are followed:

- Adaptive Routing - In adaptive routing the path is a function of network instantaneous traffic. It increases the number of possible paths usable by a packet to arrive to its destination. However deadlock and livelock situations can happen in fully adaptive algorithms which limits its usage.

- Deterministic Routing - In deterministic routing, the path is completely specified from the relative position of source and target address.

The basic parameters of the class is shown in code 3.4. The RubyNetworkParams has the basic parameters like the links, routers and the type of ruby system used. The SimpleNetworkParams inherits the RubyNetworkParams and provides the variables for the network. The create function initializes the classes to the corresponding values. The adaptive routing specifies the type of routing that has to be followed. The MessageBuffer vector is used to store the messages in the network.

Listing 3.4: Parameters of SimpleNetwork class

```
struct RubyNetworkParams : public ClockedObjectParams
{
    int control_msg_size;
    std::vector< BasicExtLink * > ext_links;
    std::vector< BasicIntLink * > int_links;
```

```
    std::vector< ClockedObject * > netifs;
    unsigned number_of_virtual_networks;
    std::vector< BasicRouter * > routers;
    RubySystem * ruby_system;
    std::string topology;
    unsigned int port_master_connection_count;
    unsigned int port_slave_connection_count;
};

\\Inherit RubyNetworkParams
struct SimpleNetworkParams : public RubyNetworkParams
{
    SimpleNetwork * create();  //Initialization
    //sets the Routing method-static or adaptive
    bool adaptive_routing;
    int buffer_size;
    int endpoint_bandwidth;
    std::vector< MessageBuffer * > int_link_buffers;
};
```

3.4.5 PerfectSwitch Class

The PerfectSwitch class works on the round robin scheduling of incoming messages to a router and the scheduling of messages at the output port of the router based on the routing table entry. The PerfectSwitch class uses the Swith class discussed in next section, to model the switch. So for a 5 port router, the switch will connect input port to all the corresponding lanes of the output port. The PerfectSwitch class also operates the MessageBuffer for each router-node pair using the Throttle class

Table 3.3: The implementation of the weight matrix

Types of Links	Horizontal	Vertical	Interlayer	No Connection
Weight	1	2	3	10000

and thus operates the network efficiently.

3.4.5.1 Switch class

The Switch in a router is modelled in this class. It is used to initiate the PerfectSwitch class by getting the routing table entry from Topology class and passing it to the PerfectSwitch class. The Switch class sets up the switch by populating the input vectors and the output vectors for the routers. This class calls the addOutPort and addLinks functions in the PerfectSwitch and Throttle class respectively to set up the ports and links for the system.

3.4.6 Topology class

The Topology class is the main class that implements the routing algorithm and runs the network. *Topology_weights*, *component_latencies* and *component_inter_switches* are 2d integer matrices that has src routers/nodes as rows and destination routers/nodes as columns. These three matrices are passed to shortest path function to find the shortest path matrix whose algorithm is defined in figure 3.5 using Hash Map data structure. Makelink function creates the inlinks and outlinks using makeOutLink,makeInLink and makeInternalLink functions. The table 3.3 explains how the weight matrix is populated for each kind of link connection. The weight between a link connecting the node and the same node is zero depicted as the diagonal elements. The weight for the links connecting the horizontal and another horizontal axis node is 1, between horizontal node and vertical node is 2 and the links between the layers connecting the two planes are depicted with weight 3.

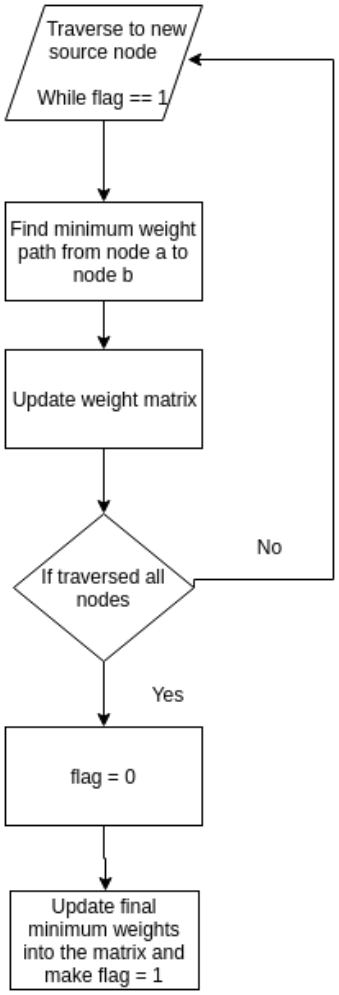

Figure 3.5: **All Pairs Shortest Path Algorithm [52].** The algorithm follows the flow shown in this figure. First the flag is set to 1 to indicate that the routing algorithm is being calculated. Then the minimum weight in the matrix for every node is calculated using the Dijkstra and Bellman-Ford algorithms. This is done till all the nodes receive the shortest path weight and then the fag is set to zero, to allow message passing through the path.

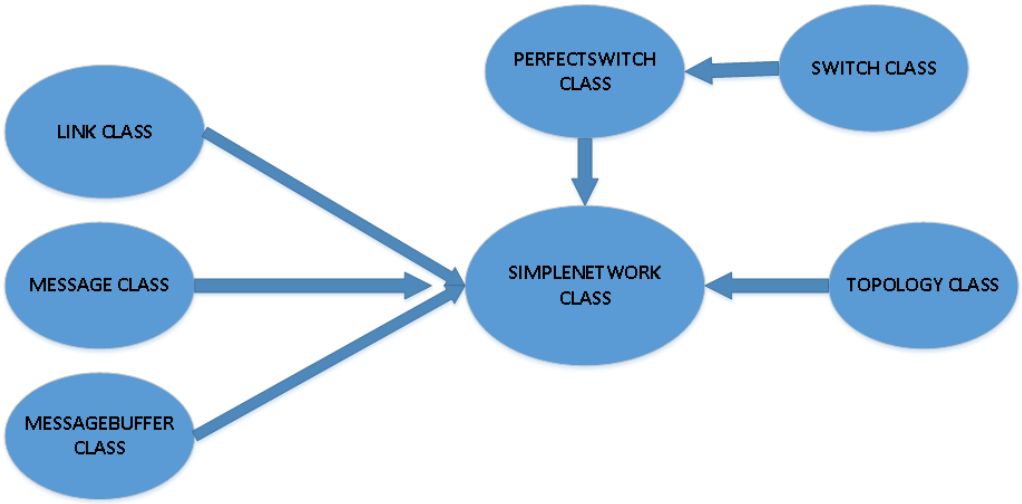

Figure 3.6: **Function flow graph of pre implemented GEM5 code.**

3.4.7 Function flow graph

The function flow of the pre implemented GEM5 code is depicted in figure 3.6 based on the discussion of the classes in the above sections. Here the SimpleNetwork class acts as the main class that inherits all the other classes. The Message class, MessageBuffer class and the Link class acts as the basic parameters used to build the network. The Switch class provides the framework for the PerfectSwitch class, which works on throttling the links of the input and output ports of the router. The Topology class runs the shortest path algorithm and populates the routing table entry, which is fed to the other classes for running the network.

3.4.8 Execution of pre implemented 2D Mesh NoC in GEM5

The following command is used to execute the 2D Mesh NoC:

$./build/X86\_MOESI\_hammer/gem5.debug configs/example/ruby\_network\_test.py -- num-cpus = 2 -- num-dirs = 2 -- topology = Mesh -- mesh-rows = 1$

whose output is shown in table 3.4. The rows of the matrix has the source nodes and the columns are the destination nodes. As for each node the simulation has a cache controller and a directory controller, each node is depicted as two nodes . Also, for each of these nodes, uni direction links are simulated and thus there should be

Table 3.4: Output for 2 core X86 simulation

	col0	col1	col2	col3	col4	col5	col6	col7	col8	col9
row0	0	10000	10000	10000	10000	10000	10000	10000	1	10000
row1	10000	0	10000	10000	10000	10000	10000	10000	10000	1
row2	10000	10000	0	10000	10000	10000	10000	10000	1	10000
row3	10000	10000	10000	0	10000	10000	10000	10000	10000	1
row4	10000	10000	10000	10000	0	10000	10000	10000	10000	10000
row5	10000	10000	10000	10000	10000	0	10000	10000	10000	10000
row6	10000	10000	10000	10000	10000	10000	0	10000	10000	10000
row7	10000	10000	10000	10000	10000	10000	10000	0	10000	10000
row8	10000	10000	10000	10000	1	10000	1	10000	0	1
row9	10000	10000	10000	10000	10000	1	10000	1	1	0

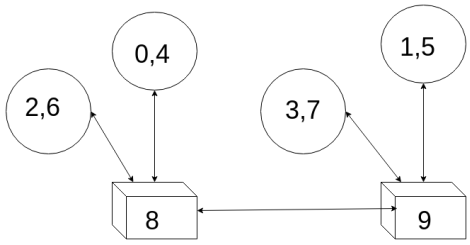

Figure 3.7: **Numbering of the cores and routers in the matrix.**

one node for outgoing and one node for incoming from the network. For a two node network, there will be four nodes (two for cache controller and two for directory controller) for receiving the data from network and four nodes for sending the data to the network. These are numbered from 0 to 7, and the two routers are numbered as 8 and 9 as shown in figure 3.7.

3.5 3D Mesh NoC implementation

Based on the study of the pre implemented 2d Mesh NoC, the changes in the source code was made. The Mesh.py python file was changed and the above classes was verified to work correctly by displaying the weight matrix in Topology class. The code listing in 3.5 was added to the Mesh.py file of 2D Mesh NoC. The links between the planes are added by calling the intlinks and extlinks functions, which in turn calls the underlying C++ functions for the Link class. The Link class in turn calls the rest of the functions to make the changes.

Table 3.5: Output of weight matrix for 3D NoC

	col0	col1	col2	col3	col4	col5	col6	col7	col8	col9
row0	0	10000	10000	10000	10000	10000	10000	10000	1	10000
row1	10000	0	10000	10000	10000	10000	10000	10000	10000	1
row2	10000	10000	0	10000	10000	10000	10000	10000	1	10000
row3	10000	10000	10000	0	10000	10000	10000	10000	10000	1
row4	10000	10000	10000	10000	0	10000	10000	10000	10000	10000
row5	10000	10000	10000	10000	10000	0	10000	10000	10000	10000
row6	10000	10000	10000	10000	10000	10000	0	10000	10000	10000
row7	10000	10000	10000	10000	10000	10000	10000	0	10000	10000
row8	10000	10000	10000	10000	1	10000	1	10000	0	3
row9	10000	10000	10000	10000	10000	1	10000	1	3	0

Listing 3.5: Code added to Mesh.py

```
#Creating the links between the planes
for row in xrange(num_rows):
        for col in xrange(num_columns):
                for ht in xrange(num_height):
                        if (ht + 1 < num_height):
                        down_id = (ht*num_rows*num_columns)
                          + col + (row * num_columns)
                        up_id = ((ht+1)*num_rows*num_columns)
                          + col + (row * num_columns)
            int_links.append(IntLink(link_id=link_count,
              node_a=routers[down_id],
              node_b=routers[up_id],
              weight=3))
            link_count += 1
        network.int_links = int_links
```

The input to the hash map in Topology class is changed according to the input from the Mesh.py file. Thus the code is verified and the weight matrix is taken as a

verification parameter. The number of links and the link ids are displayed to verify the number of links for a 3D NoC. For example, for a $4 \times 4 \times 4$ 3D Mesh NoC, 144 internal links are generated, whereas for 4×16 2D Mesh NoC, we get 108 internal links. The following command is used to run the 3D NoC and the weight matrix is displayed for verification:

$./build/X86\_MOESI\_hammer/gem5.debug configs/example/ruby\_network\_test.py -- num-cpus = 2 --num-dirs = 2 --topology = Mesh --mesh-rows = 1 --mesh-height = 2$

whose output is shown in table 3.5. The links that has weight 3 are the links that are used for inter layer communication. This is verified for higher number of cores and thus the implementation of the 3D Mesh NoC is verified to be correct.

CHAPTER 4: EVALUATION

The simulation is carried out for the SPEC CPU2006 benchmark for different system specifications to obtain the required output. First the Hammer and Directory protocols are compared based on the network traffic. Then the simulator is used to obtain the traffic pattern in 2D and 3D mesh NoCs. This uses the modification to the software described in chapter 3. Then the whole system with Hammer protocol and 3D Mesh NoC is simulated to obtain the performance parameters. All these simulations will try to prove the thesis statement described in chapter 1.

4.0.1 Experimental Setup

Simulation Parameters:

- Software: GEM5 modified for 3D Mesh NoC implementation
- CPU : X86 Timing CPU with private L1 and Logically shared L2 cache
- Flags used:-
  - num-cpus - Number of cores
  - num-dirs - Number of directories (Equal to number of cores for Mesh NoC)
  - topology = Mesh
  - mesh-rows - Specify the number of rows of the NoC
  - mesh-height - Specify the height/number of tiers of the NoC - typically limited to a max value of 8 due to lack of hardware support

CPU2006 SPEC benchmark: CPU2006 is the latest Standard Performance Evaluation Corporation (SPEC) benchmark [53]. This new suite will exercise new corners

Table 4.1: CPU2006 integer benchmark

States	Invariants
400.perlbench	Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
401.bzip2	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
403.gcc	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
445.gobmk	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	A highly-ranked chess program that also plays several chess variants.
462.libquantum	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
471.omnetpp	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	Pathfinding library for 2D maps, including the well known A* algorithm.
483.xalancbmk	A modified version of Xalan-C++, which transforms XML documents to other document types.

of CPUs, memory systems and compilers. This benchmark consists of real life applications, rather than using synthetic and artificial loop kernels. For our evaluation, we will be running the integer and floating point benchmark in GEM5 for our simulation. The integer benchmarks used in this simulation is described in table 4.1 and the floating point benchmarks used is described in table 4.2.

The omentpp and sphnix3 benchmarks have traffic generators which produce high traffic for simulation. In these kind of benchmarks, the 3D NoC is predicted to produce better improvement in network traffic than 2D NoC.

Table 4.2: CPU2006 floating point benchmark

States	Invariants
410.bwaves	Computes 3D transonic transient laminar viscous flow.
416.gamess	Gamess implements a wide range of quantum chemical computations. For the SPEC workload, self-consistent field calculations are performed using the Restricted Hartree Fock method, Restricted open-shell Hartree-Fock, and Multi-Configuration Self-Consistent Field
433.milc	A gauge field generating program for lattice gauge theory programs with dynamical quarks.
434.zeusmp	ZEUS-MP is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for the simulation of astrophysical phenomena.
435.gromacs	Molecular dynamics, i.e. simulate Newtonian equations of motion for hundreds to millions of particles. The test case simulates protein Lysozyme in a solution.
436.cactusADM	Solves the Einstein evolution equations using a staggered-leapfrog numerical method
437.leslie3d	Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme.
444.namd	Simulates large biomolecular systems. The test case has 92,224 atoms of apolipoprotein A-I.
447.dealII	deal.II is a C++ program library targeted at adaptive finite elements and error estimation. The testcase solves a Helmholtz-type equation with non-constant coefficients.
450.soplex	Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models.
453.povray	Image rendering. The testcase is a 1280x1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function.
454.calculix	Finite element code for linear and nonlinear 3D structural applications. Uses the SPOOLES solver library.
459.GemsFDTD	Solves the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method.
465.tonto	An open source quantum chemistry package, using an object-oriented design in Fortran 95. The test case places a constraint on a molecular Hartree-Fock wavefunction calculation to better match experimental X-ray diffraction data.
470.lbm	Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D
481.wrf	Weather modeling from scales of meters to thousands of kilometers. The test case is from a 30km area over 2 days.
482.sphinx3	A widely-known speech recognition system from Carnegie Mellon University

Table 4.3: Network traffic for Hammer and Directory protocol in 2D NoC

Number of Cores	Hammer traffic(Bytes/instruction)	Directory traffic(Bytes/instruction)
2	6	4
4	8	6
8	14	10
16	22	15
32	28	20
64	34	23

4.0.2 Comparison between Hammer and Directory Protocol

The Sphnix3 benchmark is executed for various number of cores for the X86 processor using the 2D NoC in GEM5 and it is observed that Hammer Protocol produces about 30% more traffic than directory. The output obtained in table 4.3 is plotted in figure 4.1. The x axis is the number of cores simulated and the y axis is the NoC traffic in Bytes per Instruction. The system is simulated for the 2D Mesh NoC. This extra traffic is due to the high communication traffic produced by broadcasting the cache coherence messages between the cores.

4.0.3 Comparison between 2D and 3D NoCs

To compare the 2D and 3D NoCs, the execution time and latency of messages on the network are plotted. The 64 core X86 processor with private L1 and logically shared L2 cache is used for simulation in GEM5. It is observed that for 3D Mesh produces lesser execution time than 2D Mesh for the sphinx3 benchmark as shown in figure 4.2 and table 4.4. 3D Mesh produces a fixed improvement in the instruction latency compared to 2D Mesh NoC as shown in figure 4.3 and table 4.5.

4.0.4 CPU2006 Benchmark output

The 64 core X86 processor with private L1 and logically shared L2 cache is used for simulation of the benchmarks in GEM5. Consistently 3D Mesh produces less Network traffic. For high traffic workloads like omnetpp and sphinx3 as shown in table 4.6 and table 4.7, the 3D NoC reduces traffic by 50% as shown in figure 4.4 and

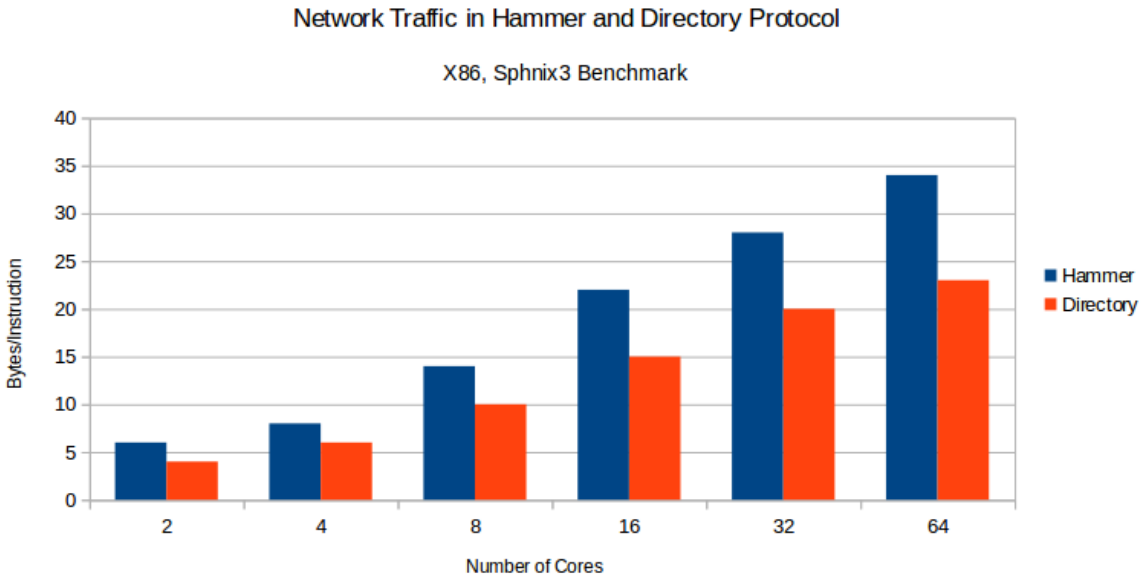

Figure 4.1: **Network Traffic for Hammer and Directory Protocols.**In the graph, the Hammer protocol produces 30% more traffic than Directory protocol for all the core counts. The higher difference is observed for the 64 core count than the 32 core count, and thus the traffic increases with the increase in number of cores.

Table 4.4: Execution time for 2D and 3D NoCs

Number of Cores	2D NoC(Cycles)	3D NoC(Cycles)
2	19894	17986
4	38567	34783
8	60451	49659
16	85042	69518
32	105892	85128
64	130695	97371

Table 4.5: Average latency for 2D and 3D NoCs

Number of Cores	2D NoC(Cycles)	3D NoC(Cycles)
2	6	5
4	12	10
8	18	15
16	24	19
32	29	25
64	35	30

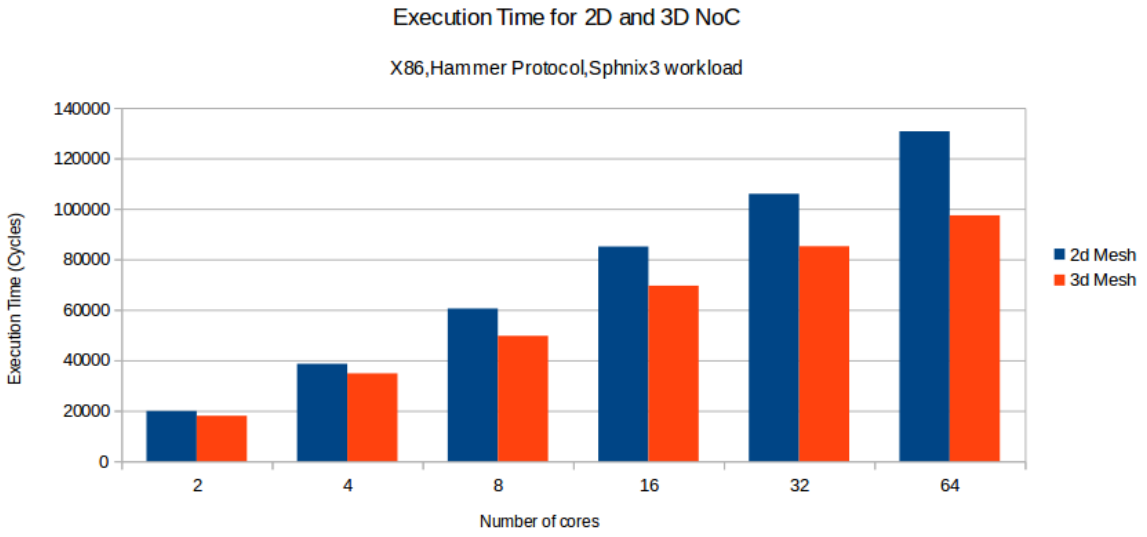

Figure 4.2: **Execution time for 2D and 3D NoCs.** In the graph, the difference in 2D and 3D Mesh NoC execution time is higher for 64 core than 32 core count. Similarly for higher core count, better 3D execution time is observed.

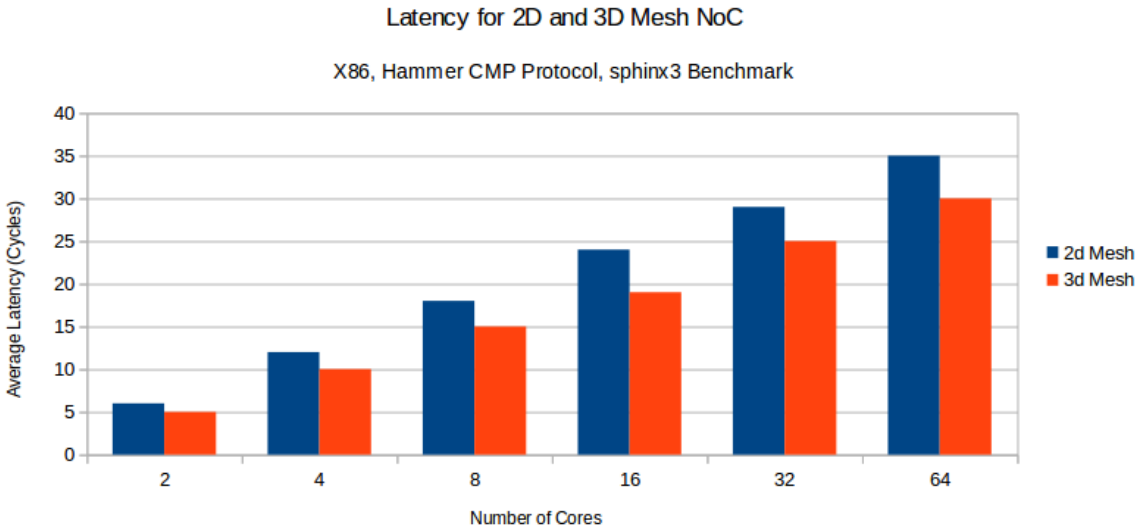

Figure 4.3: **Average Latency for 2D and 3D NoCs.** The 3D Mesh NoC produces lesser latency than the 2D Mesh NoC for higher number of core count. For lower core count like for 2 cores, the latency is almost same and the 2D and 3D NoC structures are similar for lower core count.

Table 4.6: Network traffic in 2D and 3D NoC for integer CPU2006 benchmark

Benchmark	2D NoC(Bytes/Instruction)	3D NoC(Bytes/Instruction)
perlbench	18	14
bzip2	19	16
gcc	35	28
mcf	15	12
gobmk	34	25
hmmer	22	15
sjeng	4	4
libquantum	27	25
h264ref	5	4
omnetpp	38	25
astar	28	23
xalancbmk	16	8

figure 4.5. The 3D NoC reduces traffic by around 11% on an average compared to 2D NoCs for both integer and floating point CPU2006 Benchmark.

4.0.5 Scalability of 3D NoC

It is observed that the percentage of improvement is increasing and thus is expected to reduce the extra traffic produced in hammer protocol with the increase in number of cores as shown in figure 4.6, which is equal to the traffic produced by the Hammer CMP Protocol. As the simulation in GEM5 does not support more than 64 cores in full system emulation mode, the polynomial equation for the growth should be identified by implementing in a simulator that handles simulation parameters for more number of cores.

Table 4.7: Network traffic in 2D and 3D NoC for floating point benchmark

Benchmark	2D NoC(Bytes/Instruction)	3D NoC(Bytes/Instruction)
bwaves	45	32
gamess	63	49
milc	53	50
zeusmp	38	26
gromacs	70	51
cactusADM	56	43
leslie3d	40	18
namd	66	50
dealII	71	40
soplex	38	35
povray	52	45
calculix	39	30
GemsFDTD	52	49
tonto	49	32
lbm	62	38
wrf	54	43
sphinx3	67	40

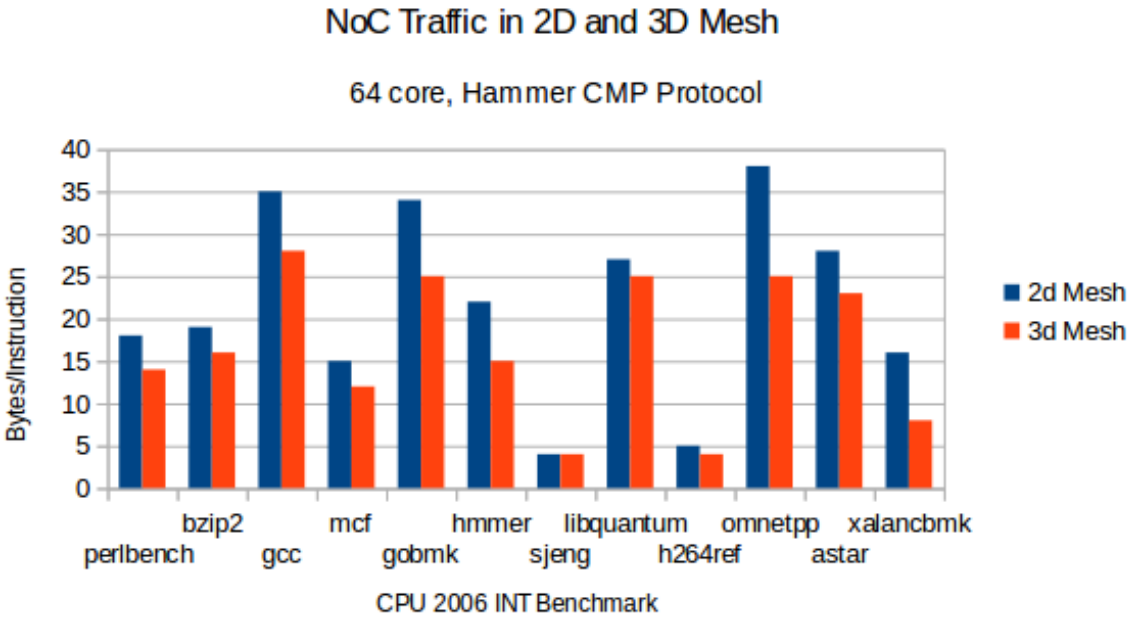

Figure 4.4: **NoC Traffic in CPU2006 integer benchmark.** From the graph for high traffic workloads like omnetpp, 3D Mesh NoC shows a lot of traffic improvement compared to 2D Mesh NoC. Whereas for less traffic workloads like sjeng, both NoCs have similar performance.

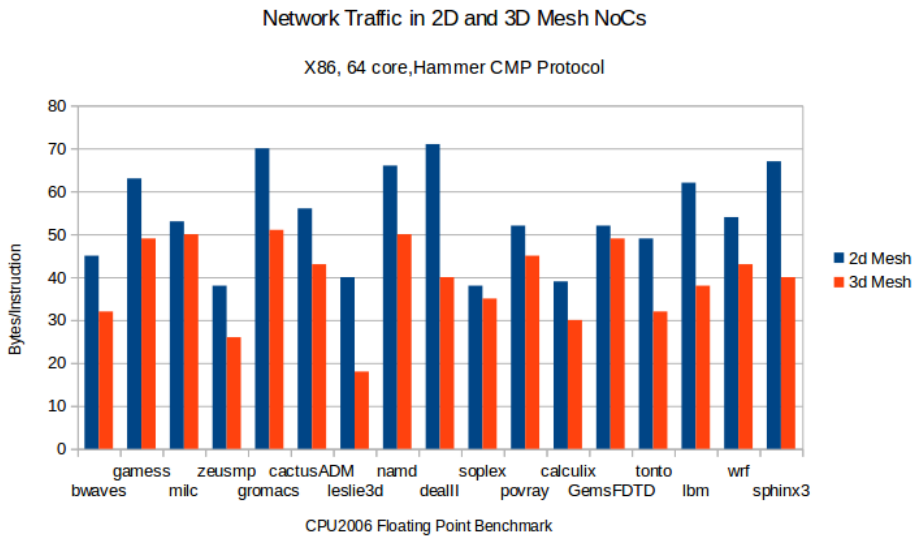

Figure 4.5: **NoC Traffic in CPU2006 floating point benchmark.** From the graph for high traffic workloads like sphnix3, 3D Mesh NoC shows a lot of traffic improvement compared to 2D Mesh NoC. Whereas for less traffic workloads like soplex, both NoCs have similar performance. The floating point benchmark generates more traffic than the integer benchmark and thus the 3D Mesh NoC shows better improvement in traffic for floating point benchmarks.

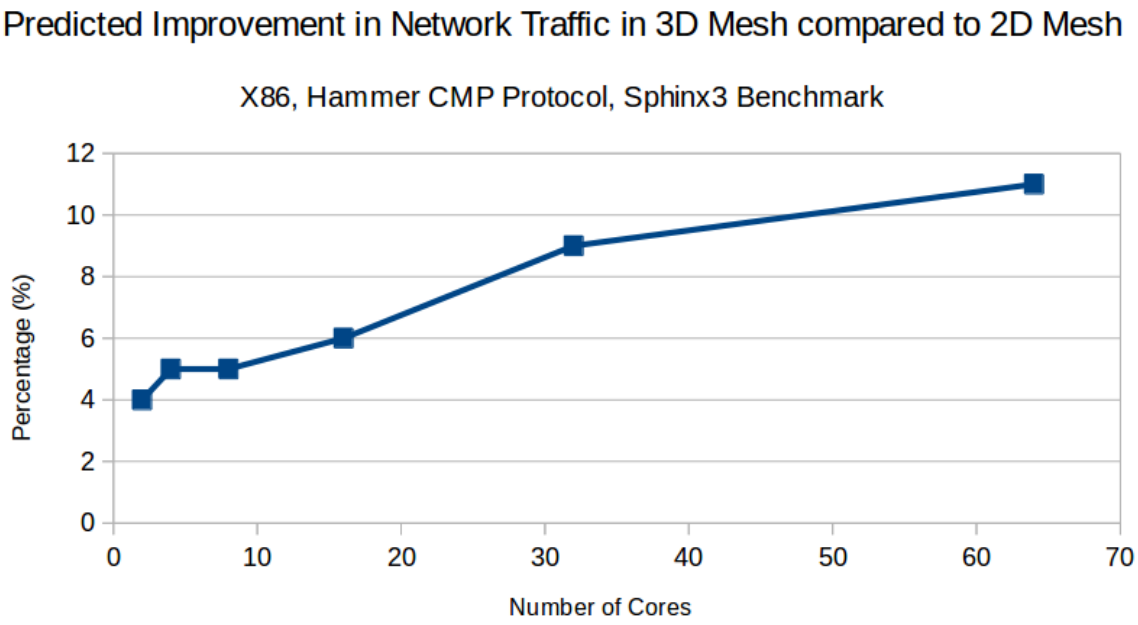

Figure 4.6: **Percentage of traffic reduction in 3d NoC compared to 2D NoC.** With the increase in number of cores, the traffic improvement increases, which is observed by the 11% improvement for 64 core count. For small core count, the improvement is diminishing. Thus the 3D Mesh NoC is expected to compensate the extra traffic produced by the Hammer cache coherence protocol for high core count.

CHAPTER 5: CONCLUSION

The increase in number of cores have lead to bottlenecks in various supercomputing systems. These problems were discussed and the high communication traffic was taken as a challenge in this research. Various architectures were researched and the most scalable Cache Coherence Protocol and Network on Chip were chosen. The Hammer Protocol was chosen as the scalable protocol due to its low on chip memory area overhead. But it produced more communication traffic and thus a 3D Mesh interconnect was proposed to reduce the traffic. Due to the unavailability of hardware resources, the system was simulated using the open source GEM5 computer architecture software by implementing a 3D Mesh NoC code package.

On executing the simulation for our system, the Hammer Protocol produced consistently 30% more traffic compared to Directory Protocol in 2D Mesh NoC. By implementing the 3D Mesh NoC and through the regression modelling, it was observed that for high traffic load, the 3D Mesh NoC reduces NoC traffic by 30% for around 500 cores. So the extra traffic produced by the Hammer Protocol is compensated by the 3D Mesh NoC implementation.

Thus the aim of this research to improve scalability of many core systems is proved theoretically and in simulation that a Many Core System on Chip can be made scalable by using the Hammer Cache Coherence Protocol and 3D mesh Network on Chip for high traffic workloads.

REFERENCES

[1] B. Bentley, "Validating the intel (r) pentium (r) 4 microprocessor," in *Design Automation Conference, 2001. Proceedings.* IEEE, 2001, pp. 244–248.

[2] M. W. Welker and O. A. Place, "Amd processor performance evaluation guide," 2003.

[3] R. B. Garner, "The scalable processor architecture (sparc)," in *The SPARC Technical Papers.* Springer, 1991, pp. 3–31.

[4] E. Hagersten and M. Koster, "Wildfire: A scalable path for smps," in *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On.* IEEE, 1999, pp. 172–181.

[5] J. Laudon and D. Lenoski, "The sgi origin: a ccnuma highly scalable server," in *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2. ACM, 1997, pp. 241–251.

[6] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[7] B. S. Ang, D. Chiou, D. L. Rosenband, M. Ehrlich, L. Rudolph *et al.*, "Start-voyager: a flexible platform for exploring scalable smp issues," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing.* IEEE Computer Society, 1998, pp. 1–13.

[8] "Top500 tianhe-2 (milkyway-2)," 2016. [Online]. Available: http://www.top500.org/system/177999

[9] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter, "Characterization of scientific workloads on systems with multi-core processors," in *Workload Characterization, 2006 IEEE International Symposium on.* IEEE, 2006, pp. 225–236.

[10] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas, "Performance impact of resource contention in multi-core systems," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* IEEE, 2010, pp. 1–12.

[11] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *Computer*, no. 9, pp. 79–85, 1997.

[12] M. A. Al-Mouhamed and K. A. Daud, "Experimental analysis of smp scalability in the presence of coherence traffic and snoop filtering," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on.* IEEE, 2012, pp. 81–88.

[13] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian, "Towards scalable, energy-efficient, bus-based on-chip networks," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–12.

[14] P. Lotfi-Kamran, B. Grot, and B. Falsafi, "Noc-out: Microarchitecting a scale-out processor," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 177–187.

[15] T. C. Xu, P. Liljeberg, J. Plosila, and H. Tenhunen, "A high-efficiency low-cost heterogeneous 3d network-on-chip design," in *Proceedings of the Fifth International Workshop on Network on Chip Architectures*. ACM, 2012, pp. 37–42.

[16] P. Foglia, C. A. Prete, M. Solinas, and G. Monni, "Re-nuca: Boosting cmp performance through block replication," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*. IEEE, 2010, pp. 199–206.

[17] M. O. Agyeman and A. Ahmadinia, "Optimising heterogeneous 3d networks-on-chip," in *Parallel Computing in Electrical Engineering (PARELEC), 2011 6th International Symposium on*. IEEE, 2011, pp. 25–30.

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

[19] L. R. Vittanala and M. Chaudhuri, "Integrating memory compression and decompression with coherence protocols in distributed shared memory multiprocessors," in *Parallel Processing, 2007. ICPP 2007. International Conference on*. IEEE, 2007, pp. 4–4.

[20] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[21] R. S. Patti, "Three-dimensional integrated circuits and the future of system-on-chip designs," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1214–1224, 2006.

[22] A. Ros, J. M. Garcia, and M. E. Acacio, *Cache coherence protocols for many-core CMPs*. INTECH Open Access Publisher, 2010.

[23] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory system characterization of commercial workloads," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 3–14, 1998.

[24] F. J. Pollack, "New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1999, p. 2.

[25] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory system characterization of commercial workloads," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 3–14, 1998.

[26] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, *Memory consistency and event ordering in scalable shared-memory multiprocessors.* ACM, 1990, vol. 18, no. 2SI.

[27] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, *Operating system support for improving data locality on CC-NUMA compute servers.* ACM, 1996, vol. 31, no. 9.

[28] A. Psathakis, V. Papaefstathiou, N. Chrysos, F. Chaix, E. Vasilakis, D. Pnevmatikatos, and M. Katevenis, "A systematic evaluation of emerging mesh-like cmp nocs," in *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on.* IEEE, 2015, pp. 159–170.

[29] S. Al-Hothali, S. Soomro, K. Tanvir, and R. Tuli, "Snoopy and directory based cache coherence protocols: A critical analysis," *Journal of Information & Communication Technology*, vol. 4, no. 1, pp. 01–10, 2010.

[30] M. M. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on.* IEEE, 2003, pp. 182–193.

[31] A. Ros, M. E. Acacio, and J. M. Garcia, "A direct coherence protocol for many-core chip multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 12, pp. 1779–1792, 2010.

[32] K. Tatas, K. Siozios, D. Soudris, and A. Jantsch, *Designing 2D and 3D Network-on-chip Architectures.* Springer, 2014.

[33] J. Balfour and W. J. Dally, "Design tradeoffs for tiled cmp on-chip networks," in *Proceedings of the 20th annual international conference on Supercomputing.* ACM, 2006, pp. 187–198.

[34] D. Seo, A. Ali, W.-T. Lim, N. Rafique, and M. Thottethodi, "Near-optimal worst-case throughput routing for two-dimensional mesh networks," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 432–443.

[35] J. Wu, D. Xie, L. Tang, and H. Wang, "Cost evaluation of three-dimensional network-on-chip," in *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on.* IEEE, 2013, pp. 133–136.

[36] R. Ville, L. Teijo, and P. Juha, "Network on chip routing algorithms," *TUCS Technical Report*, 2006.

[37] M. Hafizur Rahman, A. A. Y. Hag, R. M. Nor, and T. M. T. Sembok, "Topaz simulator on mesh and torus network," in *Information and Communication Technology for The Muslim World (ICT4M), 2014 The 5th International Conference on*. IEEE, 2014, pp. 1–5.

[38] D. Ludovici, F. Gilabert, S. Medardoni, C. Gomez, M. E. Gómez, P. Lopez, G. N. Gaydadjiev, and D. Bertozzi, "Assessing fat-tree topologies for regular network-on-chip design under nanoscale technology constraints," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 562–565.

[39] T. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins, "Topology adaptive network-on-chip design and implementation," in *Computers and Digital Techniques, IEE Proceedings-*, vol. 152, no. 4. IET, 2005, pp. 467–472.

[40] Q. Feng, J. Cao, Y. Qian, and W. Dou, "An analytical approach to modeling and evaluation of optical chip-scale network using stochastic network calculus," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 2012, pp. 1039–1046.

[41] C. Wang, W.-H. Hu, and N. Bagherzadeh, "A wireless network-on-chip design for multicore platforms," in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. IEEE, 2011, pp. 409–416.

[42] C.-H. Chao, K.-Y. Jheng, H.-Y. Wang, J.-C. Wu, and A.-Y. Wu, "Traffic-and thermal-aware run-time thermal management scheme for 3d noc systems," in *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*. IEEE, 2010, pp. 223–230.

[43] M. Xie, D. Zhang, and Y. Li, "Meshim: A high-level performance simulation platform for three-dimensional network-on-chip," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*. IEEE, 2011, pp. 349–352.

[44] F. Radfar, M. Zabihi, and R. Sarvari, "Comparison between optimal interconnection network in different 2d and 3d noc structures," in *System-on-Chip Conference (SOCC), 2014 27th IEEE International*. IEEE, 2014, pp. 171–176.

[45] M. H. Jabbar, D. Houzet, and O. Hammami, "Impact of 3d ic on noc topologies: A wire delay consideration," in *Digital System Design (DSD), 2013 Euromicro Conference on*. IEEE, 2013, pp. 68–72.

[46] S. Murali, C. Seiculescu, L. Benini, and G. De Micheli, "Synthesis of networks on chips for 3d systems on chips," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. IEEE Press, 2009, pp. 242–247.

[47] I. Loi, F. Angiolini, and L. Benini, "Supporting vertical links for 3d networks-on-chip: toward an automated design and analysis flow," in *Proceedings of the 2nd international conference on Nano-Networks*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, p. 15.

[48] M. O. Agyeman, "A study of optimization techniques for 3d networks-on-chip architectures for low power and high performance applications," *International Journal of Computer Applications*, vol. 121, no. 6, 2015.

[49] M. O. Agyeman, A. Ahmadinia, and A. Shahrabi, "Low power heterogeneous 3d networks-on-chip architectures," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE, 2011, pp. 533–538.

[50] A. Mello, L. Tedesco, N. Calazans, and F. Moraes, "Virtual channels in networks on chip: implementation and evaluation on hermes noc," in *Proceedings of the 18th annual symposium on Integrated circuits and system design*. ACM, 2005, pp. 178–183.

[51] "Gem5 cache coherence documentation," 2016. [Online]. Available: http://www.m5sim.org/Ruby

[52] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*. IEEE, 2012, pp. 1–7.

[53] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

APPENDIX A: GEM5 CODE DOCUMENTATION

NetDest.cc:-

Summary:- creates a bitset container that keeps tracks of every type (DMA, Cache,directory) of controller nodes.

Add function to enable a node

AddNetDest - Sets nodes in m_bits that are already set in "netDest"

SetNetDest function:- Copies the "set" to m_bits[machine]

Remove function:- Removes the "oldElement.num th" node in m_bits[oldelement.type] set

Removeset function:- This function clears bits that are =1 in the parameter set

Clear function:- Clears the set

Broadcast function:-this function sets all bits in the set

Count function:-This function returns the population count of 1's in the set

IsEqual function:-This function checks for set equality

OR function:- return the logical OR of this set and orSet

And function:- return the logical AND of this set and andSet

IntersectionIsEmpty function:-Returns true if the intersection of the two sets is empty

IsSuperset function:-Returns false if a bit is set in the parameter set that is NOT set in this set

ie. Checks if the set bits is superset of test.bits

Subset function:-checks if test is a subset of the set bits

IsElement function:-check if the node in "element th"

position is set or not

IsBroadcast:−this function returns true iff all bits in use are set

IsEmpty function:−Checks if all bits are set to zero

SmallestElement function:−To find the nodeID of the first node set in the bits bitset

ElementAt function:−returns the value at bits[index]

Getsize function:−returns the number of nodes in the set

SetSize function:−Sets the size of the set and resets the set (ie. Makes all bits as zero)

PerfectSwitch.cc:−

Summary:−adds the input and output port and performes switching ie. Allocating input
port and lane to appropriate output port and lane
depending on the routing_table_entry in
addoutport function.

Constructor:−

Initializes

m_round_robin_start =0, //KEEPS TRACK OF ROUND ROBIN FOR INPUT PORT

m_wakeups_wo_switch = 0 and

m_virtual_networks = 5 //VIRTUAL NETWORK PER PORT

Consumer = sw

M_switch_id = sid //KEEPS TRACK WHICH SWITCH IS CONSIDERED

M_switch = sw

Init function :- Copies input network into m_network_ptr

M_pending_message_count is created as a zero vector of size

5AddInPort:-adds incoming port vector to m_in vector

AddOutPort

Create links with m_value = 0 and m_link = port number

Add output port to M_out :- vector of ports vectors

Update m_routing_table with incoming routing_table_entry

Operate Vnet

Round robin of input ports

Selects the vnet in the input port and puts that in buffer variable

Calls operateMessageBuffer function

OperateMessageBuffer:-

Output_links – stores the output links that are allotted to

Output_link_destinations – stores the destinations

Network.cc:-

Summary:- sets up the buffers for the network and initializes the network by calling class

topology contructor and other network variables like

M_network and m_net_ptr

Constructor:-

Assigning values to

m_virtual_networks;//number of virtual networks

M_control_msg_size//size of control message
M_nodes//number of DMA controllers
Creates topology by instantiating class topology
Resize m_toNetQueues To number of m_nodes,//keeps track of
queues for messages that are
going from component/nodes to network
m_fromNetQueues to number of m_nodes //keeps track of queues
for messages that are going
from network to nodes/components
m_ordered to number of virtual networks//keeps track of lanes
(virtual lanes) whether they are
true or false-if it is true then it has a message in it. If
false, then it does not have any message in
it.
Registers "this" network ie. M_network = this; in
Rubysystem.cc sets the m_net_ptr to this network
Initialize netqueues
Init Function:- Sets m_data_msg_size = payload +
control message size
Messagesizetype_to_int:-If input is control message, it
returns control message size; if input is
data message, it returns data_msg_size
CheckNetworkAllocation function:-setting the network_num the
virtual network value to true in
m_ordered sets the vnet_type in m_vnet_type
Settonetqueue function:-Calls the networkallocation
function to set the lane to true and specify

the type of vnet Adds the buffer to the network_numth element in m_toNetQueues[id]

SetFromNetQueue function:- Calls the networkallocation function Adds the buffer to the network_numth element in m_fromNetQueues[id]

Throttle.cc:-

Summary:- moves the message from input vnet/buffer to output vnet/buffer(input and output biffers are passed to the calss function from somewhere(may be perfectswitch.cc) and changes the bandwidth. For every iteration it checks if the previous message was sent and then pushes the message.

Constructor:-

consumer = em,

m_switch_id = sID,

m_switch = em,

m_node = node

m_vnets = 0

m_link_bandwidth_multiplier = link_bandwidth_multiplier;

m_link_latency = link_latency;

m_endpoint_bandwidth = endpoint_bandwidth;

m_wakeups_wo_switch = 0;

m_link_utilization_proxy = 0;

Addlinks function:-

Gets the input and output buffer and puts them in m_in and m_out vector

Increases m_vnets by 1

m_units_remaining.push_back(0)//m_units_remaining keeps track of virtual links. Each

element of the vector stores the size of message to be transmitted in this lane.

Set consumer and description

Operatevnet function:−

Checks if the previous message was sent from the virtual lane and then dequeues the message in the input queue and enqueues it into output queue. Accordingly updates the bandwidth remaining (bw_remaining).

Wakeup function:−

Decides the order in which vnets/lanes are passed to operatevnet

Switch.cc:−

Constructor function:− instantiates perfectswitch class as m_perfect_switch; initializes m_id(router−id), m_port_buffers and m_num_connected_buffers

Init function:− passes the m_network_ptr to init function of perfectswitch.cc and m_pending_message vector is changed to a size of 5

AddInPort function:− send the "in" vector to the addinport

```
 function in perfectswitch.cc which adds the in buffer vector
 to m_in vector
Input:- in (& const vector<MessageBuffer*>)
Addoutport function:-
 m_throttles.push_back(throttle_ptr)
Hook the queues to the PerfectSwitch
        m_perfect_switch->addOutPort(intermediateBuffers,
        routing_table_entry); //intermediatebuffers gets passed
        as "out" vector, which gets added to m_out as a port.

 // Hook the queues to the Throttle
        throttle_ptr->addLinks(intermediateBuffers, out);
        //intermediatebuffer gets added as in_vector in AddIinks
        in throttle.cc

Getthrottle function:- returns the throttle object according
to link_number



SimpleNetwork.cc:-
Constructor
Input:- p(pointer of type SimpleNetworkParams)
Summary:-
Initializes the nerwork parameters (m_buffer_size,
 m_endpoint_bandwidth,
m_adaptive_routing, m_int_link_buffers,
```

```
m_num_connected_buffers) and
passes it to init function in perfectswitch.cc
Resizes m_endpoint_switches to m_nodes (//number of
DMA controllers in
Network.cc)////m_endpoint_switches tells the
destination node that which
source switch is responsible for supplying data
Stores the routers(switches) in m_switches
Init function:- Sets the m_data_msg_size to
payload_control_msg_size
in Network.cc
creates the topology using routers.size,
ext_links and int_links, that
were passed to the simplenetwork constructor.
Makeoutlink:-
// From a switch to an endpoint node
Typecast BasicLink to SimpleExtLink and pass
it to src switch's
addoutport ie. We are adding an output port
which hooks the
m_fromNetQueues[dest] (destination port
MakeInLink function:-
// From an endpoint node to a switch
Calls the AddInPort function which adds the
m_toNetQueues[src] which
selects the src port
makeinternallink function:-
```

```
// From a switch to a switch
m_switches[dest]->addInPort(queues);
 m_switches[src]->addOutPort(queues, routing_table_entry,
                                    simple_link->m_latency,
                                    simple_link->m_bw_multiplier);
```