System-Level Modeling of a Network-on-Chip

Ankur Agarwal ankur@cse.fau.edu
Assistant Professor, Department of Computer Science and Engineering
Florida Atlantic University,
Boca Raton, 33431, Florida

Abstract

This paper presents the system-level modeling and simulation of a concurrent architecture for a customizable and scalable network-on-chip (NoC), using system level tools (Mission Level Designer (MLD)). MLD supports the integration of heterogeneous models of computation, which provide a framework to model various algorithms and activities, while accounting for and exploiting concurrency and synchronization aspects. Our methodology consists of three main phases: system-level concurrency modeling, component-level modeling, and system-level integration. At first, the Finite State Processes (FSP) symbolic language is used to model and analyze the system-level concurrency aspects of the NoC. Then, each component of the NoC is abstracted as a customizable class with parameters and methods, and instances of these classes are used to realize a 4×4 mesh-based NoC within the MLD environment. To illustrate and validate the system-level operation of the NoC, we provide simulation results for various scheduling criteria, injection rates, buffer sizes, and network traffic patterns.

Keywords: Network-on-Chip, System-Level Design, Concurrency Modeling, System Modeling, Models of Computation

1. INTRODUCTION

System complexity, driven by both increasing transistor count and customer need for increasingly savvy applications, has increased so dramatically that system design and integration can no longer be an after-thought. As a consequence, system level design, long the domain of a few expert senior engineers, is coming into its own as a discipline. SystemC, introduced seven years ago, has made it possible to co-design and co-develop software and hardware, and hence, such components [1]. However, integration of an entire product at the SystemC level would take too long to complete. In this paper we propose a methodology to separately address design and optimization at a higher level, viz., Architect's level, where components pre-developed (using SystemC or any other equivalent system language) can be integrated with each other to build subsystems, and ultimately a system, with the (sub) system optimized across several more global QoS parameters. Note that for this to be practically feasible, the components have to be abstracted to a higher level, so system modeling and analysis, and final architecture selection, can be completed rapidly.

A system design process is inherently complex. The design involves multiple representations, multiple (design) groups working on different design phases, and a complex hierarchy of data and applications [2]. The different groups bring different perspectives towards system design. The system or product inconsistencies primarily arise out of lack of appropriate communication among various design teams. For example, the concerns of a hardware design engineer are different from that of a software designer [3]. Such constraints lead to an increase in product development cycle and product development cost, thereby reducing system design productivity [4]. To counter this, one will have to exploit the principle of "design-and-reuse" to its full potential [5]. Then, a system (subsystem) would be composed of reusable sub-systems (components).

The network-on-chip (NoC) architecture can improve this declining design productivity by serving as a reusable communication sub-system for an embedded device [6]. NoC provides a multi-core architecture for managing complexity by incorporating concurrency and synchronization. This NoC architecture may comprise components such as routers, input and output buffers, network interfaces, switches, virtual channel allocators, schedulers and switch allocators [7]. To develop a system from such reusable components, one has to design and develop variants of each component. For example, buffer size is a customizable parameter for an input buffer. Similarly, scheduling criteria provide customizability for schedulers. A system architect estimates performance and quality-of-service (QoS) parameters for various system configurations. Components need to encapsulate their performance metrics for various useful parameter combinations, in order to help the architect make informed decisions. We propose that a system be modeled in advance of the architect's design phase. Such a model is analyzed to ensure system functionality at an abstract level. This model can then be ported to the architect's design phase, for analyzing the performance of a system and for estimating the resources needed for mapping an application onto the system. This model must allow one to manipulate a set of parameters to fine tune system performance. Such a system model needs to have a high level representation of various performance and QoS parameters for subsystems and components. These performance and QoS parameters can then be traded-off against each other in the system model, to yield a global optimization. Such a model at the design phase will allow one to make key decisions and reduce the scope of the multidimensional search space. These key decisions may include the number of processors, hardware-software partitioning, estimated performance values for new components, and the use of existing components in software or hardware. Such design decisions have the potential to significantly enhance productivity of system design.

Such system modeling, analysis and design will not be an effective solution until we have a mechanism for modeling concurrency and synchronization issues [8, 9]. This requires representation of various activities and algorithms with appropriate MoC [10, 11, 12]. This defines our two goals for system-level designers: (1) To model the system functionality well in advance of building the actual computing system in order to provide a level of flexibility in system design. (2) To be able to manipulate an abstract set of design elements simultaneously to generate different sets of QoS parameters and performance metrics and fine tune the system model.

A model of computation (MoC) is a mathematical formalism that captures and allows us to reason about a computational system or concept independent of its implementation details. Different MoC have evolved to represent the reasoning in different areas of focus [13]. A synchronous local region of a NoC might require one or more such MoC to co-exist and interact. Further, to reduce simulation time and to integrate the subsystems into an integrated system model, other MoC may be needed. Thus, several MoC are needed for modeling an integrated system. In such a system, each component or subsystem of the system should be able to use any allowed MoC and more importantly should retain its behavior after integration of the system. Consider also the case in which a subsystem uses two or more local regions (or islands) of the NoC. These are connected by a switch in a NoC. For example consider a digital camera as a component, or as a subsystem of a much larger system, such as a wireless handheld device (system). It is possible that its design would not fit into one local region. Under such a scenario we should also be able to address the concurrency and synchronization issues because of shared resources (both local and global). We have integrated appropriate MoC to model our NoC architecture.

In this paper, we present a methodology to obtain a system-level model of a NoC which is quality-of-service driven, customizable, and parameterizable. The NoC implementation uses several MoC to model different regions. Due to the distributed nature of the NoC, the modeling of concurrency is essential for a robust design. The contributions of this paper are as follows:

1) We propose a methodology wherein the FSP symbolic language is first used to model and analyze the system-level concurrency aspects of the NoC, before any component-level modeling is done. This prevents the introduction of concurrency issues (such as deadlock and livelock) in the subsequent component-level modeling and integration phases.
2) The MLD environment is used to model each component of the NoC, using different MoC. Each component is abstracted as a customizable class with parameters and methods. Only the system-level functionality of each component is modeled, i.e. no gate-level design is done. Indeed, the goal is to obtain a rapid customizable system-level design, which the system architect can use in iterations until the final design.
3) Instances of these classes of components are used to realize a 4×4 mesh-based NoC, in the integration phase. To illustrate and validate the system-level operation of the NoC, we provide simulation results for various scheduling criteria, injection rates, buffer sizes, and network traffic patterns.
4) The NoC architecture is implemented on a field Programmable gate array (FPGA), and area results are abstracted.

The goal of this paper is not to propose yet another improved NoC architecture, but rather to develop a methodology to accelerate the development of NoC architectures, by performing system-level modeling of the NoC, taking into account concurrency issues. In the process, we show how to apply two complementary tools to the NoC modeling: FSPes and MLD. Once the NoC has been designed using these two tools, it can be easily ported to an FPGA, which allows us to rapidly obtain gate counts and other performance measures for the NoC. Hence, this methodology allows a system architect to rapidly evaluate the performance of a NoC by using system-level modeling, instead of tedious gate-level modeling, while still obtaining gate-level results.

After this Introduction, Section 2 gives an overview of the NoC architecture to be modeled, and discusses the MoC to be used at the system, subsystem and component levels of the NoC. Section 3 presents the system-level concurrency modeling of the NoC architecture using FSP. Section 4 introduces the MLD environment, and details each class of component, along with its set of customizable parameters. Section 5 presents and discusses numerical results for both the simulation of the MLD (in terms of latency) and its emulation on a FPGA (in terms of number of gates). Section 6 concludes the paper.

2. NETWORK-ON-CHIP MODELING USING MoC

To enhance the declining system design productivity one will have to introduce and adopt of new technology, and by addressing system-level design concerns at a higher level of abstraction. A system architect must be able to analyze system performance and complete “what-if-scenarios” from the executable specifications at the initial phases of the design cycle. This will help in reducing and possibly eliminating the re-spins in the design phase, thereby increasing the design productivity. To achieve such capability it is necessary to separate the design concerns, specifically computation from communication. This has led to the exploration of layered system architecture for embedded system development. The concerns related to each layer will be modeled separately, independent of the underlying architectures. This will help us increase the amount of component re-use by facilitating the portability of the components onto the different architectures. These layers are as shown in Figure 1. In this paper we discuss the components associated with the two following layers: the communication backbone and the communication

protocol. The components from these two layers have then been designed using system level modeling environment “Mission Level Designer” (MLD).

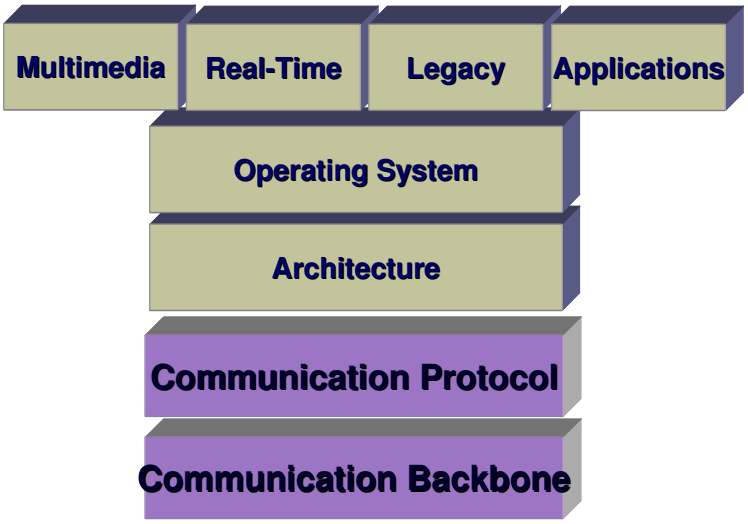


Figure 1: Layered Architecture for System Design.

A NoC is designed as a layered architecture and comprises two main layers: the communication protocol layer and the communication backbone layer. The communication protocol layer consists of the network interface (NI) and is responsible for decoupling communication from computation and packetization/depacketization. Resources such as a general purpose processor (GPP), digital signal processor (DSP), FPGA, application specific integrated circuit (ASIC), memory, or any other hardware element, are connected to the communication backbone layer components through the NI’s. A resource is called a producer (P) when data originates from it, and a consumer (C) when data is received by it. The communication backbone layer, on the other hand, is responsible for buffering, network routing, switching, flow-control, and prioritization of data. It is made up of routers, buffers and links. Figure 2 shows a 3×3 NoC architecture.

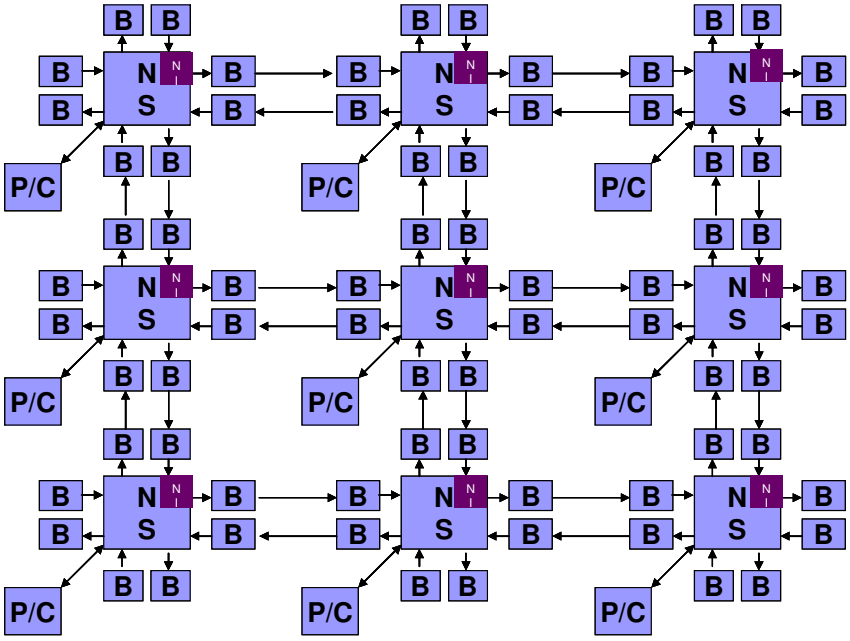


Figure 2: A 3×3 Network-on-Chip Architecture.

The Bi, Bo, P, C, N, S, and NI acronyms denote input buffer, output buffer, producer, consumer, node, scheduler and network interface, respectively. Links provide connections between routers. Routers are connected to each other and to the NI, as per a specific topology. The router block includes the node (N) and scheduler (S) components. The scheduler is responsible for controlling the traffic flow, while the actual data is routed via the node.

At the system (top) level, the NoC model must be defined at a high level of abstraction for it to be useful for performance evaluation [14]. Lower level descriptions, such as register transfer level (RTL) code and source code, while appropriate for the design level, would slow down the trade-off analysis. We should be able to adjust one of these parameters to fine tune system performance and yield different combinations of cost-performance-QoS-power dissipation. One may wish to do this on a dynamic basis for better power management of the system in the field. Thus, we can argue that at the system level, we would need some kind of manager to dynamically optimize the system-level design parameters. The local region for a NoC is again divided into two different domains: NoC at the subsystem level and NoC at the component level. At the sub-system level, we address local issues (as relating to a particular subsystem) rather than global issues. Such a subsystem will usually be some DSP subsystem, IP-core, FPGA, or ASIC. A component is another essential part of the local region. A set of components taken together would constitute the higher subsystem level. At this level, the designer would not have to worry about addressing the system-wide concurrency and synchronization issues. The design should be highly reusable in order to be utilized in other products and scalable in order to be mapped into the higher domains, i.e. the subsystems and the systems. This component level would comprise software components and a computation part, which in turn could be represented by electronic components and computer architecture components.

3. CONCURRENCY MODELING USING FSP

During the previous decade one could enhance the system performance by simply increasing the clock speeds. The International Technology Roadmap of Semiconductors (ITRS) predicts that the saturation point for these clock speeds is nearing [16, 17, 18]. Thus, we need to find other innovative ways of further enhancing performance. One way that could help in this direction is by exploiting concurrency [8]. However, if concurrency issues are not addressed properly, then it may lead any system into a deadlock state. In a multiprocessing environment there are various processes executing simultaneously. For example, in a NoC, at a given time, every resource may either be producing or consuming some data packets. Thus, it can be said that almost every element (node, link, buffer, router, scheduler, resource, etc…) might be communicating with another element at some point perhaps concurrently on the NoC platform. Therefore, there are several inter-processes communications in such a model. If these inter-process communications are not modeled properly then the system may fail. Such a failure may not be detected at the system integration phase. This is due to the fact that these failures are intermittent, which occur only under certain conditions, but nevertheless may be catastrophic. A system may be more prone to intermittent failures if concurrency concerns are not addressed properly. These intermittent failures may finally result into a deadlock or a livelock state. Thus, it is very important to model concurrency issues in such NoC systems.

The main issue is to be able to analyze whether after following all the steps for designing a concurrent system, the new system design will still have any concurrency concerns. There are various MoC which can achieve concurrency [19]. Some of these MoC's include communicating sequential processes (CSP) [20], pi-calculus [21], lambda calculus [22] and finite state machines (FSM) [23]. CSP, pi-calculus and lambda-calculus offer an effective mechanism of specification and verification. Pi-calculus is a well-defined process algebra that can be used to describe and analyze process systems. It allows mobility of communication channels, includes an operational semantics, and can be extended to a higher-order calculus, where not only channels but whole processes can be transferred. However, they are based on mathematical models; they are not only hard to understand for a software/hardware designer but also hard to apply practically.

FSM's have been extensively used in the past, especially in the domain of modeling of control applications. But these FSM's are not able to address the increasing size of the software content in the systems. Hierarchical FSM's have replaced FSM's in modeling concurrent applications. Several tools and methodologies have also been introduced which address these concurrency concerns. Unified Modeling Language (UML) addresses these concurrency concerns in the form of state diagrams and sequence charts [24]. But UML may not be useful for analyzing the system exhaustively for concurrency failures.

FSP and Labeled Transition System Analyzer (LTSA) provide a framework for modeling concurrency and analyzing it exhaustively [25]. FSP is a language based on CSP. But unlike CSP a system designer will not have to analyze the system specification in the form of a mathematical model in order to expose concurrency issues in the system. We can develop a simplified concurrency model in FSP and analyze the concurrency issues with LTSA graphically. LTSA also provides a capability of exhaustive analysis to uncover synchronization issues such as deadlocks and livelocks.

3.1 Concurrency Model for NoC

To develop a concurrency model, we first write a high-level specification. This should not be platform or data dependent. We then identify the concurrent processes in our model. As the concurrency issues arise only among interacting processes, not in the (sequential) internal data processing, we model only the (external) interaction among various processes. This reduces the model complexity significantly; models execute faster due to reduced state exploration. Figure 3 shows a high-level concurrency modeling flow diagram.

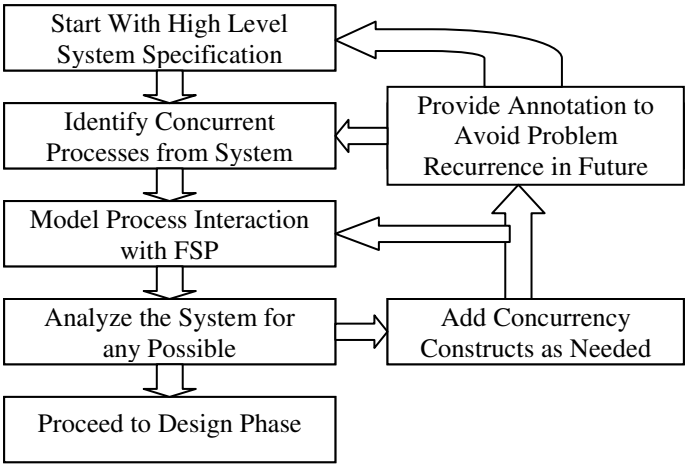


Figure 3: Concurrency Modeling Flowchart.

To model concurrency, we start with a very high-level specification. This high-level specification must not contain any internal details of the process behavior. It encompasses the following:

* Data will be received in serialized packet format;
* There will be several paths available arranged in a matrix fashion for the data packet to travel;
* Data may be buffered at each intersection;
* Further routing is available based on the availability and congestion of the links at the destination;
* Packets will contain destination addresses and priorities;
* Links may be unidirectional (one link for each direction) or bi-directional.

We further made two assumptions to simplify our concurrency model: links are unidirectional and nodes are arranged in a 2-D mesh. Concurrent processes in a system can be identified from these high-level specifications. In the NoC model, the identified concurrent processes were links, buffers, schedulers, nodes, producers and consumers. We then define detailed specifications for

all individual processes. To model concurrency one must follow an incremental approach. In the NoC model, we first developed the producer and link processes. We then checked for concurrency issues in these two processes before including other processes in the model. Since links are attached to buffers, we then added a buffer process to the model.

As concurrency concerns arise only when two or more processes interact with each other, and not from the internal actions of any processes, we should therefore only model interactions among various processes and not their internal actions. Further, a NoC model with nine nodes and schedulers along with thirty-six buffers, nine producers and consumers and hundreds of links will comprise of tens of thousands of states. Thus it will not only be difficult but almost impossible to model it with FSP and analyze it with LTSA. Therefore, we abstracted this model to represent only one representative scenario of interaction. If this interaction does not have any concurrency issues then other interactions may be replicated in a similar manner to avoid any deadlock or livelock in the system. This abstracted model is represented in Figure 4.

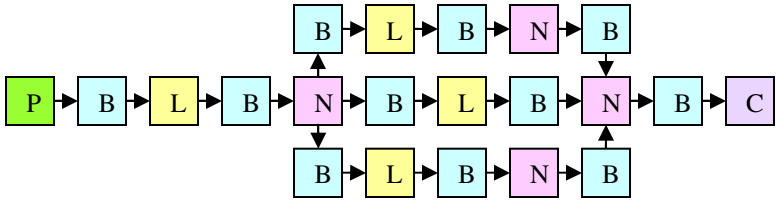


Figure 4: NoC Model for Interaction Between One Producer Process and One Consumer Process.

Further, from the implementation of the link process we realized that a link is responsible for just forwarding the data to the next process. In this it does not play any role which may cause any concurrency concerns. Thus we eliminated the link process from the model. We further reduced this model shown in Figure 3, into a more simplified model by removing unnecessary data paths. Instead of showing three data paths (three buffers connected to one node) we represented the model with two data paths (two buffers with one node). This is due to the fact that synchronization issues will arise when more than one process tries to interact with another process. But as long as the number of similar processes is more than one, we can have a general model to represent these interactions. It may be concluded that the number of buffers (as long as it is more one) will not make any difference in addressing concurrency issues. However, it should be noted that a node with one buffer will not have the same implementation as a node with two buffers. Figure 4 represents a further simplified model with a single data path for the NoC.

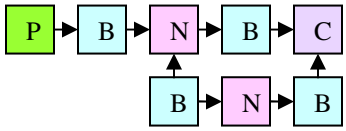


Figure 4: NoC Model for Interaction Between one Producer Process and one Consumer Process with Link Processes Eliminated.

The final abstracted model as shown in Figure 4 has been described in Figure 5. We can realize from the model implementation that there are one producer process - p1, two buffer processes - b1 and b2, one scheduler process – s1. The final composition process includes all of the above processes. These processes have been exhaustively tested. Figure 6 shows results for an exhaustive simulation. It can be seen from simulation results that there are 351 different possible states and 882 possible transitions among these states in the system. None of these states are involved in any deadlock and livelock.

```
PRODUCER = (buffAvail → (hiPriDataOut → PRODUCER | midPriDataOut → PRODUCER)).

const N      = 2
range Data_Range = 0..N
BUFFER = STATE[0][0],
STATE[a:Data_Range][b:Data_Range]
   = (when ((a+b)<N) buffAvail → (hiPriDataIn  → STATE[a+1][b] |midPriDataIn → STATE[a][b+1])
     |when (a>0) shortDataInBuff → nodeGrantFast → hiPriOut → (confirm → STATE[a-1][b]
                                                                |notConfirm → STATE[a][b])
     |when((a==0)&& b>0) dataInBuff → nodeGrantSlow → midPriOut → (confirm → STATE[a][b-1]
                                                                   |notConfirm → STATE[a][b])).
const M = 1
range Node_Requests = 0..M

SCHEDULER = STATE[0][0][0][0],
STATE[h1:Node_Requests][l1:Node_Requests][h2:Node_Requests][l2:Node_Requests]
                        = (when (((h1+l1+h2+l2)<M)) shortDataInBuff1                    → STATE[h1+1][l1][h2][l2]
                          |when (((h1+l1+h2+l2)<M)) dataInBuff1                         → STATE[h1][l1+1][h2][l2]
                          |when (((h1+l1+h2+l2)<M)) shortDataInBuff2                    → STATE[h1][l1][h2+1][l2]
                          |when (((h1+l1+h2+l2)<M)) dataInBuff2                         → STATE[h1][l1][h2][l2+1]
                          |when (h1>0) nodeGrantFast1 → (outBuffAvail1      → STATE[h1-1][l1][h2][l2]
                                                         |outBuffNotAvail1  → STATE[h1-1][l1][h2][l2])
                          |when (h2>0) nodeGrantFast2 → (outBuffAvail2      → STATE[h1][l1][h2-1][l2]
                                                         |outBuffNotAvail2  → STATE[h1][l1][h2-1][l2])
                          |when ((h1==0)&&(h2==0)&&(l1>0)) nodeGrantSlow1   → (outBuffAvail1
                                                                              → STATE[h1][l1-1][h2][l2]
                                                         |outBuffNotAvail1  → STATE[h1][l1-1][h2][l2])
                          |when ((h1==0)&&(h2==0)&&(l2>0)) nodeGrantSlow2   →  (outBuffAvail2
                                                                              → STATE[h2][l1][h2][l2-1]
                                                         |outBuffNotAvail2  → STATE[h2][l1][h2][l2-1])).

||FINAL1 = ({b1, b2}:BUFFER||s1:SCHEDULER||p1:PRODUCER)/{
            b1.hiPriDataIn/p1.hiPriDataOut,b1.midPriDataIn/p1.midPriDataOut,b1.buffAvail/p1.buffAvail,
            b1.shortDataInBuff/s1.shortDataInBuff1,b1.dataInBuff/s1.dataInBuff1,
            b1.nodeGrantFast/s1.nodeGrantFast1,b1.nodeGrantSlow/s1.nodeGrantSlow1,
            b2.shortDataInBuff/s1.shortDataInBuff2,b2.dataInBuff/s1.dataInBuff2,
            b2.nodeGrantFast/s1.nodeGrantFast2,b2.nodeGrantSlow/s1.nodeGrantSlow2,
            s1.outBuffAvail1/b1.confirm,s1.outBuffNotAvail1/b1.notConfirm,
            s1.outBuffAvail2/b2.confirm,s1.outBuffNotAvail2/b2.notConfirm}.
```

Figure 5: FSP Implementation for Abstracted NoC Model.

```
Compiled: BUFFER
Compiled: SCHEDULER
Compiled: PRODUCER
Composition:
FINAL1 = b1:BUFFER || b2:BUFFER || s1:SCHEDULER || p1:PRODUCER
State Space:
 24 * 24 * 9 * 2 = 2 ** 15
Composing...
-- States: 351 Transitions: 882 Memory used: 3043K
Composed in 110ms
FINAL1 minimising.....
Minimised States: 351 in 46ms
No deadlocks/errors
Progress Check...
-- States: 351 Transitions: 882 Memory used: 3397K
No progress violations detected.
Progress Check in: 31ms
```

Figure 6: Simulation Result for Abstracted NoC Model.

4. COMPONENT MODELING WITH MLD

Our NoC model has been designed in an object oriented manner, wherein each component is an instance of a class with properties and methods. In this section we describe the different classes defined in order to build the NoC model. These classes are (where the capital letters indicate that a class is being referred to): Producer, InputBuffer, Scheduler, Router, OutputBuffer, and Consumer. While being self-contained as per the principles of object-oriented design, these classes need to interact with one another.

The NoC model supports component customization, which eases the task of a design architect. The latter can now change the actual system model to understand the impact of various design parameters without the need for changing the actual design. Thus, it has the potential to provide a more effective analysis of the result by investing less time as compared to traditional performance analysis. The customization parameters are discussed as part of the description for each class. The MLD tool was used to design the classes. The next section provides a brief overview of MLD and the motivations for choosing it for our NoC design.

4.1 System-Level Design with MLD

MLD is a system-level modeling environment which allows one to model a system at an abstract level. MLD supports modeling in different domains such as discrete event (DE), synchronous data flow (SDF), finite state machine (FSM), dynamic data flow (DDF), and synchronous reactive (SR), among others. Multiple domains can be combined to represent a system model. The performance analysis of a model is done at an abstract level, therefore simulations run faster as compared to other modeling environments, which are C, C++ or SystemC-based.

System models are implemented in MLD through either a graphical editor or the Ptolemy Tool command language, PTcl. The functionality of the modules may be specified by a hierarchical block diagram, a finite state machine, a module defined in C/C++ language, or by a PTcl module definition.

4.2 Producer Class

A producer is instantiated from the Producer class. It comprises a resource and a resource network interface. A producer generates the required traffic pattern and packetizes the data into flits. A flit is the smallest unit of communication supported in the NoC. The Producer class has been implemented with a synchronous data flow model of computation as it is responsible for continuous data flow.

Referring to Figure 2, a producer outputs a flit when buffAvail is asserted by the corresponding buffer. A flit is time-stamped at the time of its generation. The timestamp is used to determine the latency involved in delivering the flit. The source and destination address fields of the flit header are updated at this time. The flit header has fields for its priority, timestamp, X-direction of source address, Y-direction of source address, X-direction of destination address, and Y-direction of destination address. The priority of this flit is governed as per a statistical distribution block. For example, in the case of a uniform distribution pattern, every third flit will be a high priority flit. Once the new flit has its timestamp, source and destination addresses and priority fields updated, it is then forwarded to the output through dataOut.

The customizable parameters for the Producer class are: (1) the distribution pattern of the data; (2) the packet injection rate, i.e. the amount of data generated as a function of time; (3) the priorities of the generated data - High, Mid or Low. Each set of parameters is described below. We used three statistically generated traffic distribution patterns – uniform traffic with linear destination, random, and application-specific patterns.

The packet injection rate represents the number of flits per cycle injected into the network for transmission. Defining it as a customizable parameter, allows us to test the NoC model for varying load conditions. The injection rate is changed from 0.1 to 1.0 in equal increments of 0.1 to check the resultant network latency. An injection rate of 1.0 is equivalent to a producer outputting a flit every clock cycle. Similarly, a rate of 0.33 represents the injection of a flit in every three clock cycles.

We provided three priority levels for data packets in our NoC model: High priority, Mid priority and Low priority. High priority supports control signals such as Read (RD), Write (WR), Acknowledge (ACK), and interrupts. Therefore, high priority data is a short packet (single flit packet). Mid priority supports real-time traffic on the system, while Low priority supports non-real time block transfers of data packets. We have defined control signals as High priority because the data must respond immediately to a control signal. Therefore, a control signal must reach its destination in time to manage the communication flow of the network. Real-time data must be delivered in real-time bounds. Therefore, we have assigned Mid Priority to real-time data. The rest of the data on the network belongs to the Low priority class. The number of priority levels is a customizable parameter.

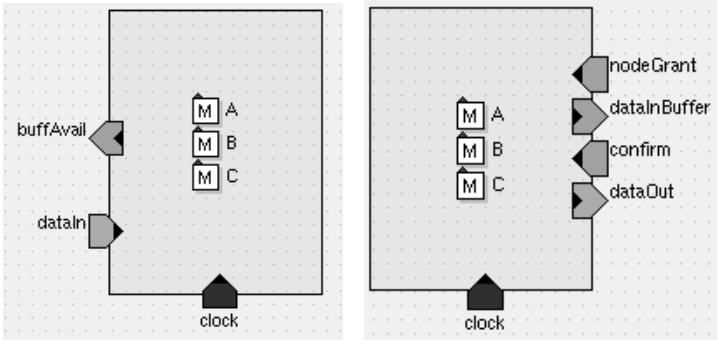
4.3 InputBuffer Class

An input buffer is instantiated from the InputBuffer class. It contains a buffer, a buffer scheduler, and a virtual channel allocator. An input buffer stores the incoming flits, generates the proper handshaking signals to communicate with the scheduler and forwards the flits to the router. An input buffer has an input block and an output block. These two blocks are controlled by a state machine. Thus, we have implemented InputBuffer in the DE and FSM domains. Two concurrent FSM's are responsible for storing the input data at the input terminals of the input buffer and forwarding the data to a router at the output terminal of the input buffer. The DE domain is used for implementing a handshaking protocol. A data forwarding path has been implemented based on a "request-grant" signaling approach (other NoC implementations refer to it as flow control logic). Incoming flits corresponding to all the priority levels (High, Mid and Low) are stored in a common buffer. Let buffSize represent all the available space in an input buffer. buffAvail indicates whether there is space available in the input buffer. The stored data flits are forwarded to the router based on their scheduling criteria. For example, in case of priority-based scheduling, High priority flits are forwarded before the Mid or Low priority flits, etc. Table 1 shows the classification of flit types.

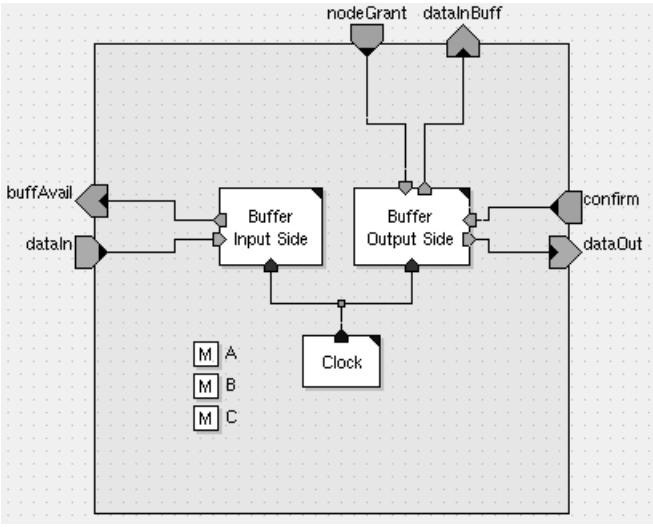
Bit combination	Flit type
00	No flit
01	High priority flit
10	Mid priority flit
11	Low priority flit

T

Table 1: Flit Priority Bit Combinations.



(a) (b)



(c)

Figure 7: MLD Implementation of the InputBuffer Class: (a) Input Side; (b) Output Side; (c) Combined Input and Output Sides.

The input buffer is virtually divided into three different buffers A, B and C (see Figures 7 (a) and (b)). We have provided flexibility in the size of these virtual buffers (the combined size is a fixed user-defined parameter). The input side (i.e. Figure 7 (a)) is responsible for checking the available buffer space and allocates memory space for an incoming flit. The output side (i.e. Figure 7 (b)) forwards a flit and releases the allocated memory.

We have implemented a handshaking protocol for forwarding a flit. The availability of data flits in the input buffer for further transmission is indicated by dataInBuff. If a grant comes in response to this request (nodeGrant), the flit stored in the buffer is forwarded to the corresponding Router. Table 2 shows the bit allocation of nodeGrant.

Bit combination	Flit type
00	No flit
01	High priority flit
10	Mid priority flit
11	Low priority flit

Table 2: nodeGrant Bit Allocation.

On receipt of nodeGrant, a data packet is forwarded by the output side of the input buffer through dataOut. Figure 7 (c) shows the complete implementation of the input side and the output side of the InputBuffer class. Three virtual buffers as shown in Figures 7 (a) and (b) are represented as memories (M) in Figure 7 (c). A flit is not removed from the input buffer until a confirmation (via confirm) is received from the scheduler (from the output side of Figure 7 (c)). If confirm is not received, the data flit is not removed from the input buffer; however, it will be queued for later forwarding. We provided three virtual channels (VC) per buffer. A VC controller inside the input buffer updates these virtual channels. We implemented the InputBuffer class with a discrete-event model of computation. The DE domain facilitates the signaling protocol and is thus a suitable choice for implementing handshaking protocols.

The customizable parameters for the InputBuffer class are the buffer size and the scheduling criteria. We can change the buffer size to any value. By changing the buffer size, we can understand its impact on latency, area and therefore the silicon cost. A buffer forwards data to the

next block based on its scheduling criteria. We provided three scheduling criteria (also referred to as service levels, SL): first-come-first-serve (FCFS), priority-based (PB), and priority-based-round-robin (PBRR). In the FCFS scheduling criterion, all the data packets are treated in the same way. A data flit is forwarded to the next block based on its arrival time. The data flit with the earliest arrival time will be forwarded first. In the PB scheduling criterion, the input buffer first forwards all data packets with High priority, then those with Mid priority. It forwards Low priority data packets only when there are no High or Mid priority data packets present in the input buffer. In the PBRR scheduling criterion, the input buffer forwards data packets with different priorities in a specific rotation pattern. It first forwards a High priority packet, then a Mid priority packet, followed by a Low priority packet. This cycle is repeated throughout the simulation.

4.4 Scheduler Class

A scheduler is instantiated from the Scheduler class. The data and control parts of a node have been separated (the Router class handles the data part and the Scheduler class handles the control signals) to manage concurrency issues and make the design more scalable and reusable. The actual data flows from one node to another through a router. The path of the actual flow of this data is defined by a scheduler. A scheduler is also responsible for synchronizing the input buffer with the router and the router with the output buffer while receiving and transmitting the data further. It schedules the incoming requests for data transmission to the next node, by checking for the availability of the output data path and by arbitrating the requests from various input buffers associated with it. The Scheduler class has been implemented in the DE domain. A scheduler is mainly responsible for synchronization, and thus the DE domain is the ideal MoC for its implementation. Figure 8 shows the MLD implementation of the Scheduler class interfaced with five buffers on input and output side. The scheduler is connected to five instances of InputBuffer (one for each direction in the 2-D mesh network and a fifth buffer for the local Producer class connected through a NI) and, similarly, five instances of OutputBuffer on the output side.

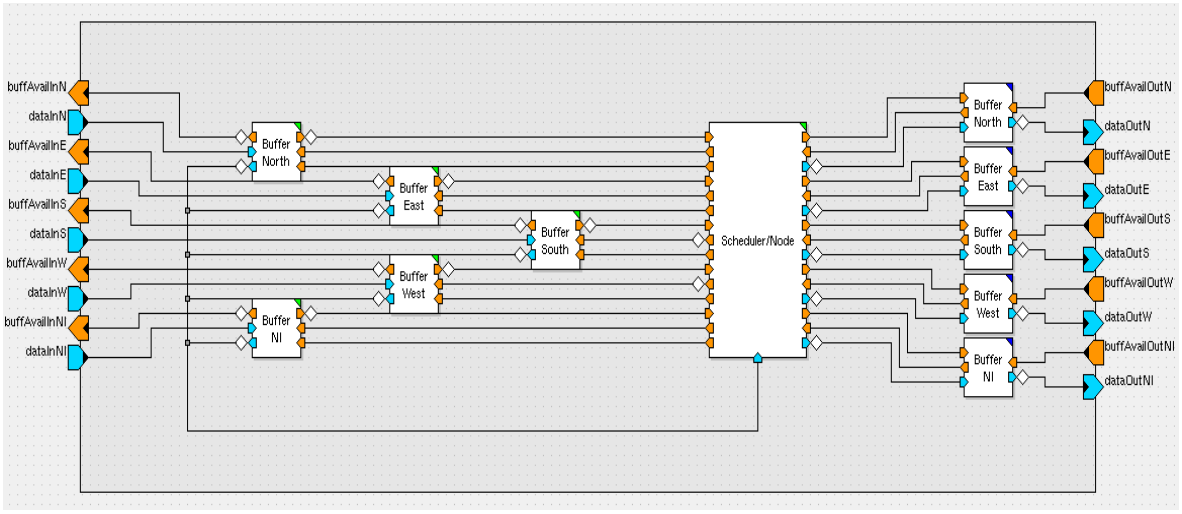


Figure 8: MLD Implementation of the Scheduler Class Interfaced with Five Input and Output Buffers.

A scheduler accepts the requests from an input buffer via dataInBuff (the input signal on the left side of Figure 8) and allocates the data path by asserting nodeGrant (the output signal on the left side of Figure 8). The data path allocation is based on the availability of an output buffer and the route. We have embedded multiple algorithms in the Scheduler class as discussed in the previous section. A scheduler will select an input buffer from multiple input buffers requesting for transmission of a data packet. The router informs the scheduler about the physical output path for flit transmission via outputPort. Availability of the data path is acknowledged by assertion of confirm. This interaction between the Scheduler and Router classes is shown in Figure 9. The

Scheduler class is implemented in two different MoC. The control part of the scheduler has been implemented with FSM. This FSM interacts with the DE domain for proper handshaking with the input buffer and router on the input side and the output buffer on the output side of the scheduler.

The customizable parameter for the Scheduler class is the scheduling criterion. The Scheduler class supports different scheduling criteria: first-come-first-served (FCFS), round-robin (RR), priority-based (PB), and priority-based-round-robin (PBRR). Thus, in a network we can have a combination of scheduling algorithms.

4.5 Router Class

A router (shown in Figure 9), instantiated from the Router class, determines the output path and handles the actual data transfer on the implemented backbone. The router receives a certain number of data flits per unit time and is constrained by the data bandwidth for transmitting a fixed number of flits at the output. Thus, the Router class has been implemented in the SDF domain. A dimension-order routing protocol was implemented in the Router class for determining the output path. We have provided customization in routing algorithms as discussed earlier. Upon receipt of data, a router extracts the destination information and determines the physical output port for transmitting the data. This output port address is sent to the corresponding scheduler, which determines the availability of this port. Upon its availability, data flits are then forwarded to the corresponding output buffer for this port.

The type of routing algorithm is a parameter for the Router class. It can be set to: X-direction-first, Y-direction-first, and XY-random. In the X-direction-first algorithm, the data is routed to the X-direction first provided there is the possibility of the data to be routed to the Y-direction as well. The Y-direction-first algorithm works similarly. In the XY-random algorithm, if there is a possibility for the data to be routed to the X-direction as well as the Y-direction, the direction is chosen in a randomized fashion with the same likelihood for the two choices.

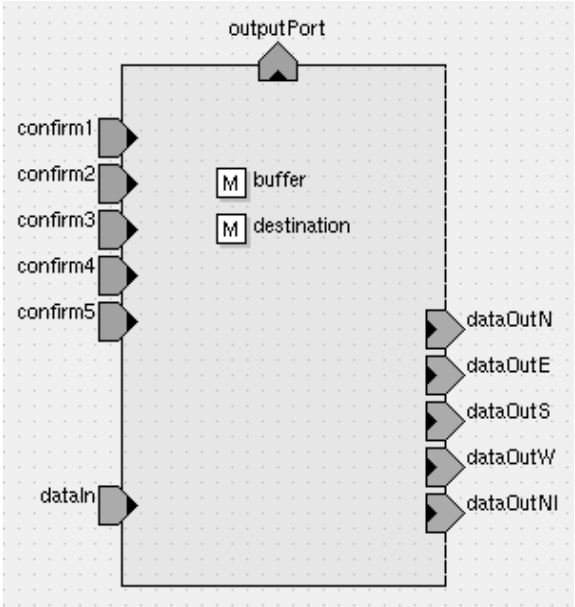


Figure 9: MLD Implementation of the Router Class.

4.6 OutputBuffer Class

An output buffer (shown in Figure 10), instantiated from the OutputBuffer class, accepts the incoming flits from the router and forwards these flits to the input buffer of the next node. It is implemented in the form of two concurrently executing state machines. The received flits are stored in the output buffer memory. The input state machine accepts and stores the data flits

while there is available memory in the output buffer. Upon the request of the scheduler by reqBuffAvail, the availability of buffer space in the output buffer is signaled by buffAvail. The output state machine senses the request for transmitting data from the output buffer of the next router via outputBuffAvail of that output buffer and forwards the data flit, if that signal was asserted.

The customizable parameters for the OutputBuffer class are the same as those for the InputBuffer class.

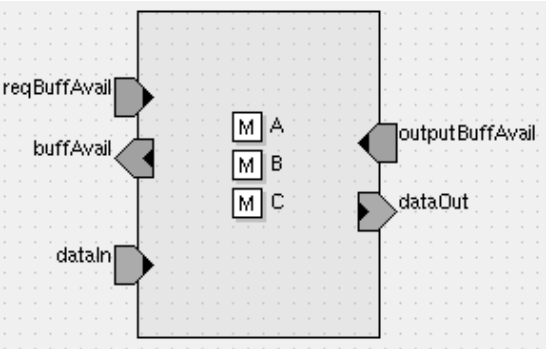


Figure 10: MLD Implementation of the OutputBuffer Class.

4.7 Consumer Class

A consumer, instantiated from the Consumer class, comprises a computing resource and a network interface (NI). A consumer accepts data flits, strips off the header information, and forwards the remainder of the data to its internal computing resources. A consumer consumes data packets. Thus, as we did for the Producer class, we have implemented the Consumer class in the SDF domain.

5. NUMERICAL RESULTS

Figure 11 shows the resulting 4x4 mesh-based NoC after system integration. We have simulated this NoC model with different packet injection rates (varying from 0.13 to 1.0), buffer sizes (1 through 10), and scheduling criteria (PB, PBRR, FCFS, RR). From these simulations we have estimated the High, Mid, and Low priority latencies. We have further implemented the NoC on a FPGA and estimated the total area and areas for each component.

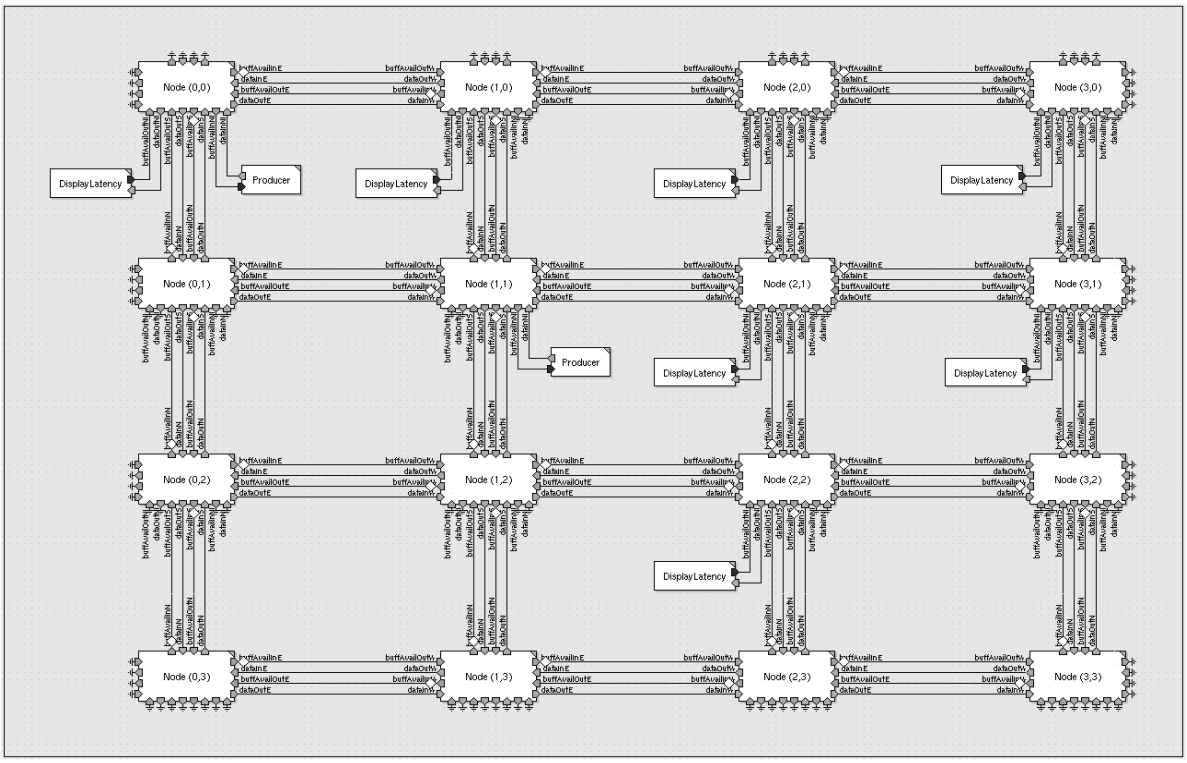


Figure 11: MLD Implementation of 4×4 Mesh NoC.

5.1 Latency Results via Simulation

The minimum latency for a single hop will be 6 clock cycles due to synchronization signals (concurrency cost) among different components: the 1st clock cycle is for storing data flits into the input buffer (buffAvail); the 2nd clock cycle is for requesting the data output to the scheduler (dataInBuff); the 3rd clock cycle is for receiving the grant signal from the scheduler (nodeGrant); the 4th clock cycle is for forwarding the data to the router (dataOut); the 5th clock cycle is for confirming the output path availability (reqBuffAvail); the 6th clock cycle is for forwarding the data packet to the output buffer (dataOut).

To simulate the NoC, we injected 10,000 flits into the network. High priority flits are used for transmitting control signals such as MemRD (Memory Read), MemWR (Memory Write), IORD (Input/Output Read), IOWR (Input/Output Write), and interrupts among others, while Mid priority flits and Low priority flits are used for transmitting real-time data and non-real time block transfers, respectively. For the network simulation, control signals were assumed to account for 10 % of the total data traffic flow, with real-time data and non-real time data accounting for 20 % and 70 %, respectively. However, this traffic load can be altered as it is a customizable parameter.

The final implementation of the NoC network was simulated for various scheduling criteria (FCFS, PB, PBRR, and RR) for varying buffer sizes (from 1 to 10) and varying packet injection rations as well. The FCFS scheduling algorithm does not prioritize the data packets. Thus, packets with all three priorities suffer almost the same latency. As the buffer size increases from 1 to 10, the data latency also increases from 14 to 107. With a larger buffer size, a data packet has to wait in the buffer for a longer time. Thus, larger buffer sizes lead to higher data latencies. In PBRR scheduling, High priority data packets must secure the lowest latency while Low priority packets have the highest latency. The data packet latencies for PBRR scheduling criteria vary from 8 to 30 clock cycles for High priority, from 15 to 36 clock cycles for Mid priority, and from 15 to 140 clock cycles for Low priority. The main advantage of the PBRR algorithm is its capability to serve all the five input buffers connected with the scheduler at the same time. Each input buffer

receives an equal scheduler response time. The latency results for PBRR scheduling are very similar to those for RR scheduling: indeed, the PBRR scheduling algorithm rotates and serves each input buffer in a RR fashion. However, it provides better results when more than three input buffers are producing data at the same time. In such a scenario, RR scheduling will not be able to deliver the High priority data in time. However, PBRR will work effectively under such a condition.
In a real-time embedded system, a system must obey real-time requirements. The NoC must deliver real-time data and control signals required for this data processing in a timely manner. Consequently, we must select scheduling criteria that can perform this task. Figure 12 shows High priority data latency results against buffer sizes for different scheduling criteria.

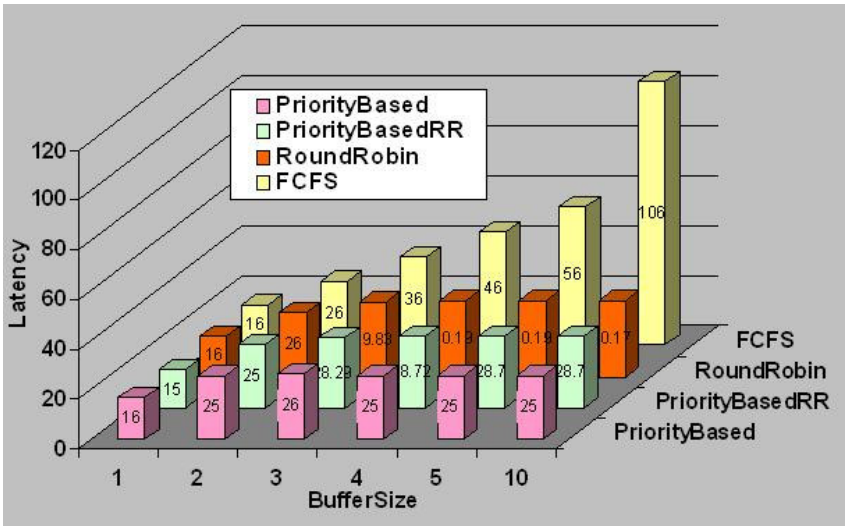


Figure 12: High priority data latency vs buffer size for different scheduling criteria.

It can be seen that the FCFS and RR scheduling algorithms may not always deliver High priority data packets in real-time bounds. Thus, we should use the PBRR or PB scheduling criteria for this NoC architecture. We used Low priority data latency results to finally choose between PBRR and PB scheduling. Figure 13 shows Low priority data latency against buffer sizes for PBRR and PB scheduling.

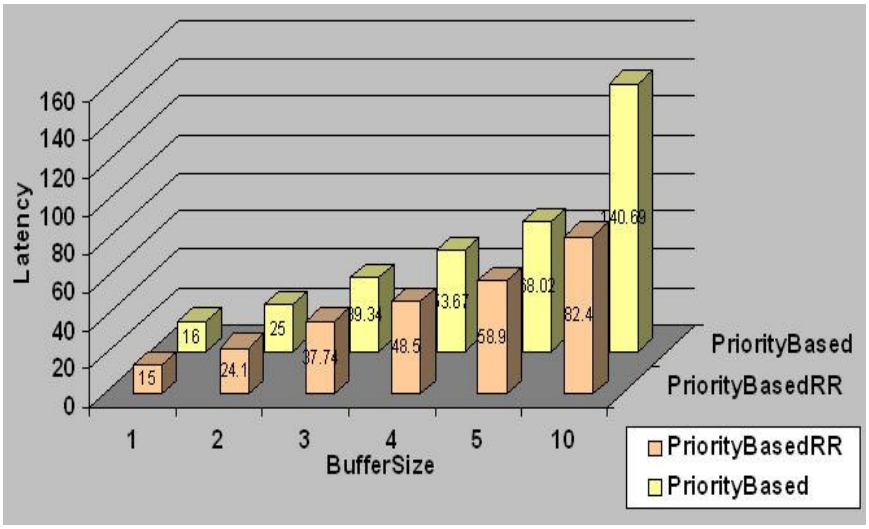


Figure 13: Low Priority Data Latency vs Buffer Size for Different Scheduling Criteria.

PB scheduling delivers the High priority data and Mid priority data in a timely manner. However, it has a higher latency for low priority data. Thus, it is best to use PBRR as the scheduling criterion for this NoC architecture.

5.2 Area Results via FPGA Implementation

We implemented all the Network-on-Chip (NoC) classes with Field Programmable Gate Arrays (FPGA) to extract the total NoC area and area information of each component. Table 3 shows the area result for a 32-bit and 64-bit Input Buffer and Output Buffer implementations. These area results are not directly proportional to the buffer size i.e. a linear relationship between buffer size and number of gates does not exist. This is mainly because of two reasons:

1. Our Input Buffers and Output Buffers have scheduling capability, along with the functions of storing and forwarding the data packets, as discussed in chapter 5. This scheduling circuit does not depend upon the buffer size. Therefore, buffer of a size two will not have double the number of gates as compared to a buffer of size one.

2. The internal architecture of an FPGA is divided into several Configurable Logic Blocks (CLBs) along with other components such as Digital Clock Manager (DCM) and block Random Access Memory (BRAM). A CLB is further divided into four slices. Architecture of each CLB is identical. The number of gates is calculated by multiplying the number of occupied slices with the total number of gates in a single slice. Thus, the equivalent FPGA implementation will count the total number of gates for a slice even if it is not fully occupied. Similarly, if we use a part of BRAM for storing the data packet/flits, then we account for total number of gates for that BRAM.

Consequently, we will not see a linear relationship between the buffer size and the number of gates.

Buffer Size	No. of Gates for 32-Bit Buffer	Number of Gates for 64-Bit Buffer
1	7,863	10,686
2	9,162	11,581
3	9,622	12,423
4	9,812	12,310
5	9,924	12,510
10	10,680	13,273

Table 3: Total Number of Gates for Different Buffer Sizes

Table 4 shows area result for different scheduling criteria (Round Robin (RR), Priority Based (PB), First Come First Serve (FCFS), and Priority Based Round Robin (PBRR)) and Router. Router handles the actual data transfer, thus has dedicated lines for sending and receiving data bits. Thus, Router takes more number of gates than Scheduler.

Component	Number of Gates
PBRR Scheduler	5,954
FCFS Scheduler	3,155
RR Scheduler	3,674
PB Scheduler	3,554
Router	9,589

Table 4: Total Number of Gates for Different Scheduling Algorithms in Scheduler and Router.

5.2.1 Area Results via FPGA Implementation: Impact of Buffer Size on NoC Area: There are eighty input buffers and eighty output buffers in a 4×4 mesh based NoC. Thus, buffer size is a key parameter for deciding the number of gates used in a NoC architecture. Figure 14 shows the High, Mid and Low data latencies against different 64-bit buffer sizes for PBRR scheduling.

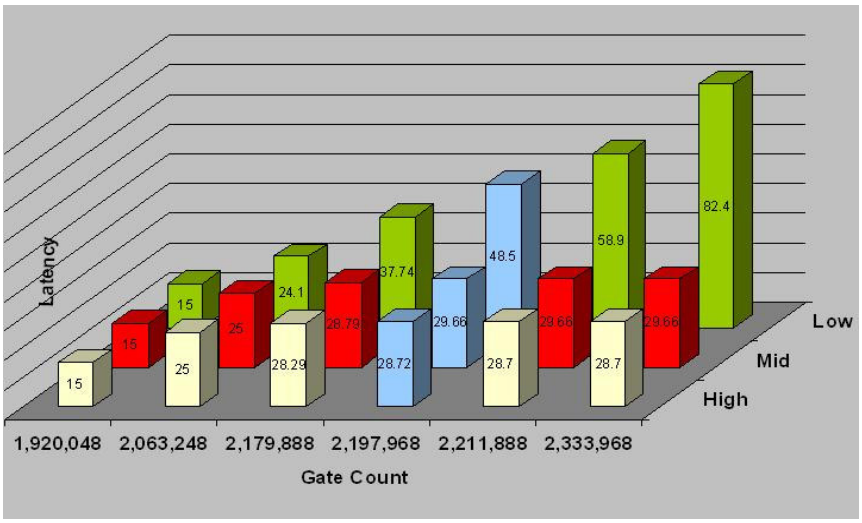


Figure 14: Latency vs NoC area with PBRR scheduling and different buffer sizes.

We also provided the total NoC area (gate count) for different buffer sizes with PBRR scheduling criteria. A buffer size of 10 produces a high value of Low priority data latency (82.4 clock cycles). Thus, we do not recommend using a buffer size of 10. Buffer sizes of 1 and 2 will have lower values of High priority data latency, however they may not provide sufficient amount of buffering needed in a network. The High and Mid priority data latencies for buffer sizes of 3, 4, and 5 are almost similar. A buffer size of more that 5 will have a higher number of gates; consequently the leakage current will also increase. Therefore, for this NoC it is recommended to use buffer sizes of 3, 4, or 5.

5.3 Leakage Power Consumption Results for NoC Parameters

FPGAs have three main factors affecting the power consumption. These are leakage current, dynamic power, and inrush current. Leakage current is a small amount of current that exists even when the gate is idle. Such power consumption is referred as leakage power consumption. Its value depends upon the technology. In 90 nm technology this leakage current is the dominant factor. As the process gets smaller (due to fabrication technology advances), the core voltage decreases. Therefore, even if the frequency increases, the rate of increase of the dynamic power drops. However, static power has grown exponentially with the introduction of new process technologies. Thus, leakage power that was ignored earlier needs to be accounted in total power consumption.

Inrush current is the result of the internal circuitry contention when the FPGAs are powered up. These spikes can measure in multiple amperes, thus causing huge amount power consumption. However, with the advancement of FPGA technology and introduction of 90 nm FPGA technology this issue has been resolved. Further with recent advances, parts of FPGAs can be put to sleep state consuming a minimum amount of leakage power.

FPGA architecture mainly consists of CLBs, IOB (Input/Output Blocks), carry-chain adder, multipliers, BRAMs, and DCM among others. In this dissertation, we provide some insight on leakage power consumption for 1.2 V, SRAM (Static Random Access Memory) FPGAs built in 90 nm technology. The embedded version of this FPGA is also available and consists of only CLBs. We detail our leakage power consumption on this embedded version of FPGA. As explained earlier that the FPGA architecture consists of CLBs and is symmetrical in nature. So we focus our study on single CLB.

Leakage current may change by a factor of three based on the input to a circuit. For example, a 2:1 multiplexer has three inputs (2 inputs and 1 select line). Its leakage current value ranges from

2.88 mW (when all the inputs are zeros) to 0.91 mW (when all the inputs are ones). However, the total value of leakage current will also depend upon the utilization factor. Unutilized logic will consume less leakage power as compared to utilized logic. Usually, we attain a high utilization factor for CLBs and a low utilization factor for IOB. At 25o C, an FPGA consumes 4.2 µW per CLB. Table 5 lists the number of slices used for designing NoC components and the total leakage power consumption for NoC implementation with different buffer sizes and scheduling algorithms. This leakage power will entirely depend upon the FPGA technology.

.

NoC Components	Number of Slices	Leakage Power (W)
Buffer Size 1	481	0.0020202
Buffer Size 2	546	0.0022932
Buffer Size 3	594	0.0024948
Buffer Size 4	601	0.0025242
Buffer Size 5	612	0.0025704
Buffer Size 10	671	0.0028182
Router	778	0.0032676
Scheduler (PB)	290	0.001218
Scheduler (RR)	274	0.0011508
Scheduler (PBRR)	356	0.0014952
Scheduler (FCFS)	201	0.0008442

Table 5: Leakage Power Consumption for NoC Components

6. CONCLUSION

We have developed a methodology to realistically model and optimally design the communication backbone of a NoC. The model is built with reusable and customizable building blocks, which are abstract enough to facilitate rapid analysis. We used MLD simulation environment because of its support for multiple MoC, which helps with both fast model building and simulation.

Acknowledgement

This work reported in this paper was carried out as part of the “One-Pass-to-Production” research project, funded by the iDEN division of Motorola, Plantation, FL.

7. REFERENCES

1. A. Jantsch, H. Tenhunen, *“Network on Chips”* Kluwer Academic Publishers, Boston, (2003)

2. G. Desoli and E. Filippi, “*An outlook on the evolution of mobile terminals: from monolithic to modular multi-radio, multi-application platforms”*, IEEE Circuits and Systems Mag., 6(2): 17-29, 2006.
3. W. C. Rhines, *“Sociology of design and EDA”*, IEEE Trans. on Design and Test, 23(4): 304-310, 2006.

4. E. A. Lee and Y. Xiong, *“System level types for component-based design”*, Workshop on Embedded Software, California, 2001.

5. Y. Xiong and E. A. Lee, *“An extensible type system for component-based design”,* International Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Germany, 2000.

6. D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, *“NoC synthesis flow for customized domain specific multiprocessor SoC”*, IEEE Trans. on Parallel and Distributed Systems, 16(2): 113-129, 2005.

7. S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, *"A Network on Chip architecture and design methodology"*, IEEE Symposium on VLSI, 117-124, 2002.

8. A. Agarwal and R. Shankar, *"Modeling concurrency on NoC architecture with symbolic language: FSP"*, IEEE International Conf. on Symbolic Methods and Applications to Circuit Design, 2006.

9. J. Burch, R. Passerone, and A. L. Sandivanni-Vincentelli, *"Overcoming heterophobia: modeling concurrency in heterogeneous systems"*, IEEE International Conf. on Applications of Concurrency to System Design, 13-32, 2001.

10. E. A. Lee and A. Sangiovanni-Vincentelli, *"Comparing models of computation"*, IEEE/ACM International Conference on Computer-Aided Design, 234-241, 1996.

11. A. Jantsch and I. Sander, *"Models of computation and languages for embedded system design"*, IEEE Proceedings on Computers and Digital Techniques, 114-129, 2005.

12. A. Girault, B. Lee; E.A. Lee, *"Hierarchical finite state machines with multiple concurrency models"*, IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems, 18(6): 742-760, 1999.

13. A. Agarwal and R. Shankar, "A Layered Architecture for NoC Design methodology", IASTED International Conf. on Parallel and Distributed Computing and Systems, pp. 659-666, 2005.

14. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, *"Performance evaluation and design trade-offs for network-on-chip interconnect architectures"*, IEEE Transaction on Computers, 54(8):1025-1040, 2005.

15. A. Agarwal, R. Shankar, C. Iskander, G. Hamza-Lup, *"System Level Modeling Environment: MLdesigner"*, 2nd Annual IEEE Systems Conference, Montreal, Canada, 2008.

16. Bertozzi and L. Benini, *"Xpipes: A network-on-chip architecture for gigascale systems-on-chip"*, IEEE Circuits and Systems Magazine, 4(1):18-31, 2004.

17. S. J. Lee, K. Lee, S. J. Song, and H. J. Yoo, *"Packet-switched on-chip interconnection network for system-on-chip applications"*, IEEE Transaction on Circuits and Systems II, 52(6), :308-312, 2005.
18. W. J. Dally and B. Towles, *"Route packets, not wires: on-chip interconnection networks"*, IEEE International Conference on Design and Automation, 2001.
19. J. Burch, R. Passerone, A.L. Sandivanni-Vincentelli, "Overcoming heterophobia: modeling concurrency in heterogeneous systems", IEEE International Conference on Application of Concurrency to System Design, pp. 13-32, 2001

20. G.H. Hilderink, *"Graphical modeling language for specifying concurrency based on CSP"*, IEEE Proceedings on Software Engineering, 150(2): 108 – 120, 2003.

21. S. Chrobot, *"Modeling communication in distributed systems"*, IEEE International Proceeding in Parallel Computing in Electrical Engineering, 2002

22. T. Murphy, K. Crary, R. Harper, F. Pfenning, *"A symmetric modal lambda calculus for distributed computing*", 19th Annual IEEE Symposium on Logic in Computer Science, 2004.

23. A. Girault, B. Lee; E.A. Lee, *"Hierarchical finite state machines with multiple concurrency models*", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 18(6): 742-760, 1999.

24. M. Barrio, P. De La Fuente, *"IEEE International Computer Science Conference on Software Engineering*", 1997.

25. J. Magee, J. Kramer, *"Concurrency State Models and Java Programs*", West Sussex England, John Wiley & Sons, (1999).