# NoC Performance Estimation tool based on formal models for Scientific Applications

Carlos Andrés Piedrahita Velásquez

Tesis de maestría presentada para optar al título de Magíster en Ingeniería

Asesor
Gustavo Adolfo Patiño Álvarez, Doctor (PhD)

Universidad de Antioquia
Facultad de Ingeniería
Maestría en Ingeniería
Medellín, Antioquia, Colombia
2022

| Cita | Piedrahita-Velásquez C.A. [1] |
| --- | --- |
| **Referencia**<br>Estilo IEEE (2020) | [1] Piedrahita-Velásquez C.A, "NoC Performance Estimation tool based on formal models for Scientific Applications", Tesis de maestría, Maestría en Ingeniería, Universidad de Antioquia, Medellín, Antioquia, Colombia, 2022. |




Maestría en Ingeniería.

Grupo de Investigación Sistemas Embebidos e Inteligencia Computacional (SISTEMIC).


Centro de Documentación Ingeniería (CENDOI)

**Repositorio Institucional:** http://bibliotecadigital.udea.edu.co

Universidad de Antioquia - www.udea.edu.co

**Rector:** John Jairo Arboleda Céspedes.

**Decano/Director:** Jesús Francisco Vargas Bonilla.

**Jefe departamento:** Augusto Enrique Salazar Jiménez.

ABSTRACT

As the number of Processing Elements (PEs) present in Systems-on-Chip (SoC) increases, the complexity of the applications running in these systems increases as well. A Network-on-Chip (NoC) provides the communication channels between the PEs, and its proper design is a decisive factor in meeting the performance requirements of an SoC application. However, NoC design is a complex task that involves the exploration of a wide design space involving subjects as NoC topology and application mapping. During the design process, a system architect explores several NoC configurations, trying to find a system prototype that meets the performance requirements of an application. This is done using detailed simulations of the hardware and software of the system (known as cycle-accurate simulations), and in general, they are very time-consuming to run. On the other side of the spectrum, high-level simulations of a system are faster to run, but due to the simplicity of their models, their estimations can be inaccurate. Thus, a tool capable of fast execution and accurate NoC performance estimations is of interest. In this thesis, an NoC performance estimation tool based on a formal model is proposed. This tool uses a recently released state-of-the-art traffic suite to model traffic patterns of several scientific applications running in an NoC. This differentiates the tool proposed in this thesis from similar tools reported in the literature, that can only model synthetic traffic patterns and thus, their performance estimations are of limited use. With the ability of modeling traffic patterns of real applications, the NoC performance estimations made by the proposed tool are similar to the ones obtained from a cycle-accurate simulation. Additionally, it will be shown that the tool proposed in this thesis runs faster than a cycle-accurate simulator, which makes it ideal for design space exploration at the early stages of system development.

# ACKNOWLEDGMENTS

I would like to thank my family for their continuous support and encouragement during the development of this thesis. Also, I thank the Universidad de Antioquia for granting me the “Estudiante Instructor” scholarship, which enabled me to pursue my Masters' studies. Additionally, I want to thank the members of the SISTEMIC group for the constructive discussions about the work done during this thesis. Finally, I want to thank my thesis’ supervisor, Ph.D. Gustavo Patiño for his guidance, encouragement, and patience, especially, in this personally difficult last year for me.

# CONTENTS

# List of tables

# List of Figures

# I. INTRODUCTION AND MOTIVATION

## A. *Problem Description*

The design of Multi-Processor System-on-Chip (MPSoC) consists of an increasingly complex task, due to the demanding requirements in terms of performance (latency and traffic rate) of the communication network (Network-on-Chip, NoC), common in this type of systems [1], [2]. Figure I-1 shows how this design complexity has increased through the years, and how system designers had tackled this complexity. With a wide design space, system architects have almost infinite possibilities for system definition (ISA selection, topology selection, network size, application mapping, etc.), and, therefore, it is important to estimate the performance metrics of the NoC from early design stages, since systems that do not meet the performance specifications of the application should be discarded quickly, to avoid wasting resources and time [3], [4]. Nonetheless, the estimation of these metrics is done through exhaustive simulations of the hardware and software of the system [5]–[7], simulations that for complex applications can take up several days to deliver results, making the design space exploration a cumbersome task. For example, Figure I-2 shows the average simulation time of four cycle-accurate simulators (gem5 [8], Sniper [9], PTLsim [10], and Multi2Sim [11]) running the MiBench [12] and SPEC2006 [13] benchmarks. From the figure it can be seen that for the SPEC2006 benchmarks, the average simulation time can be as high as 70000 seconds (19.4 hours), this, even though the simulators are using speed-up mechanisms like fast-forwarding and only executing a reduced number of instructions [14].

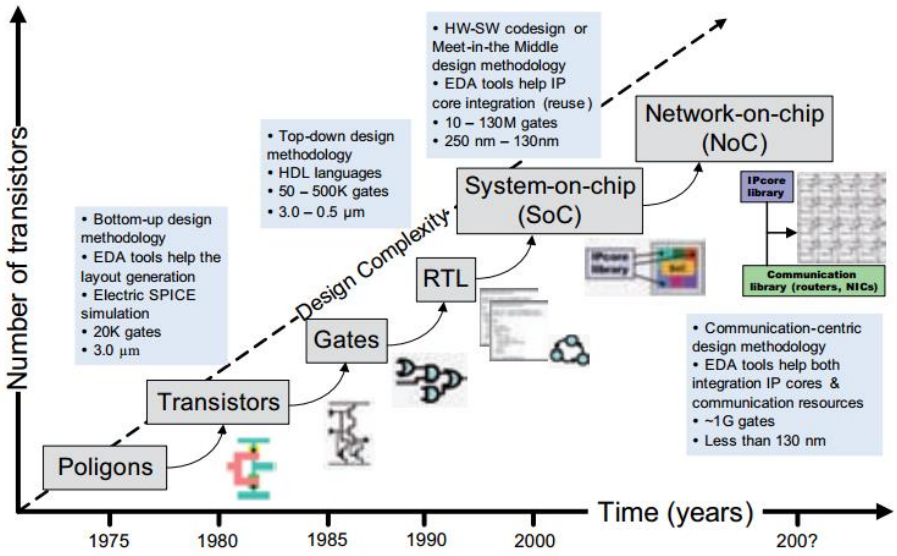


**Figure I-1.** Design Complexity in Electronic Systems. Taken from [15].

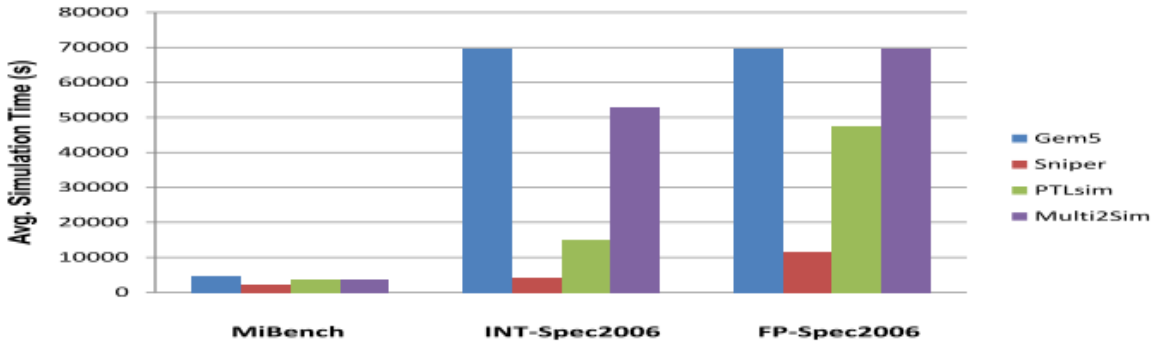


**Figure I-2.** Average simulation time of different cycle-accurate simulators. Taken from [14].

Also, cycle-accurate simulators like Noxim [16], BookSim [17], and gem5's garnet [18] offer limited flexibility in terms of NoC topologies that can be studied, thus, limiting the search space during the design process. As an alternative for cycle-accurate simulation, system architects have several options for system modeling, these include formal methods, static profiling and trace-based analysis, each of them with varying levels of abstraction of the system. However, these options imply trade-offs between estimation time and accuracy of the estimations as can be seen in Figure I-3 [19]. From this figure, cycle-accurate simulation offers the most accurate results, both its estimation time can be measured in days in the worst case. Also, from Figure I-3, as the level of abstraction increases, the estimation time is reduced greatly being measured in seconds in the best case, but the accuracy of the results decreases notably.

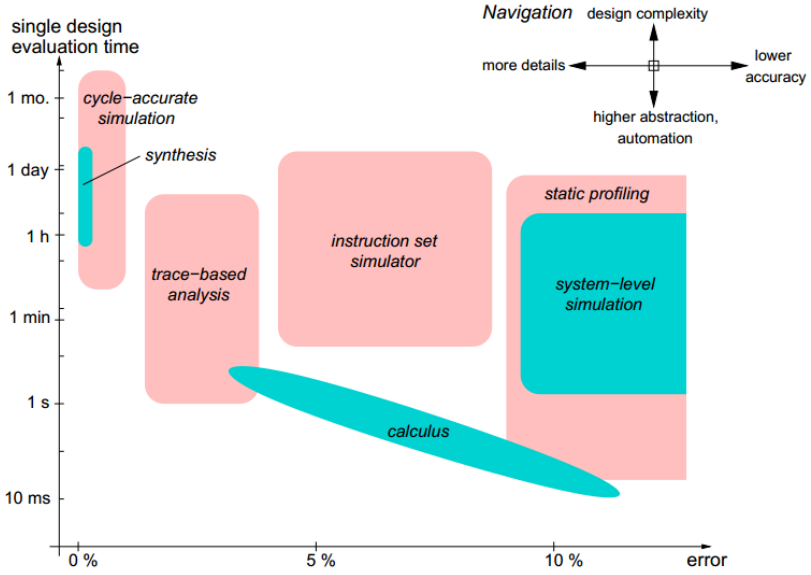


**Figure I-3.** Estimation time vs accuracy of the estimation. Taken from [20].

Examples of trace-based analysis tools can be found in [21]–[23], An example of an instruction set simulation tool is [19], and an examples of a static profiling tool is [24]. From Figure I-3, formal models (depicted as *calculus* in the figure) offer the greatest gains in estimation time but their inaccuracy can be rather high. The source of this inaccuracy is the high level of abstraction of these models, that includes the use of synthetic traffic patterns (e.g., uniform, tornado, hot spot, etc.) to model the traffic patterns of an application. Examples of Formal models used for performance estimation are Queueing Theory [25], Dataflow graphs [26], and Network Calculus [27]. However, recent developments in the modeling of traffic patterns of real applications, like the MCSL NoC traffic suite [28], makes possible the use of more realistic

traffic patterns with formal models, improving the insight that can be obtained, while preserving their fast estimation time. Additionally, the inherent inaccuracy of formal models can be acceptable during early stages of the Design Space Exploration process, where it is desirable to investigate multiple versions of a system rapidly to find suitable candidates that meet the design constraints. Later, these candidates can be further analyzed using cycle-accurate simulation to obtain more refined results, continuing this way with the design space exploration process [3].

*1) Research Motivation*

From the above, the necessity for a tool that permits the estimation of the performance (latency and throughput) of an NoC, in a reasonable time and with results as close as possible to those given by cycle-accurate simulation seems evident. Understanding this need, this thesis presents a tool based on Queueing Theory for estimating the performance of an NoC.

*2) Objectives*

*a) General Objective*

Develop a software tool based on formal models, for the estimation of latency and throughput in an NoC with a specific topology oriented to scientific applications.

*b) Specific Objectives*

a. Identify scientific applications whose implementation can benefit from a multi-processor architecture (MPSoC).
b. Evaluate the advantages and disadvantages of the formal models available in the literature, to identify the most suitable models for the characterization of latency and traffic rate on an NoC.
c. Implement in a software tool, the formal model identified as the best suited for estimating latency and traffic rate in an NoC.
d. Extend the functionality of a state-of-the-art cycle-accurate simulator, to validate the results obtained with the formal modeling of the estimation of latency and traffic rate in an NoC.
e. Validate the results obtained using the tool based on formal models, comparing them with the results obtained with the previously modified cycle-accurate simulation tool.
f. Identify through formal modeling of latency and traffic rate, the most suitable network topology for the implementation of various scientific applications.

## B. *Thesis Overview*

In this thesis, an alternative to cycle-accurate simulation is proposed: a tool based on Queueing Theory and estimate the performance metrics of an NoC. This tool can implement various NoC topologies and reproduce the stochastic traffic properties of various scientific applications, retrieving performance metrics of interest of the NoC (latency and throughput),without considering the implementation details observed in a cycle-accurate simulation of the hardware and software of the system, thus, making this tool "lightweight" from a computational point of view. This implies that the time needed to obtain results using the proposed tool can be less than the time required using a cycle-accurate simulation, which in turn, would reduce the time spent doing design space exploration at the early stages of system development.

## C. *Thesis contributions*

This thesis presents NoCSimulator, a tool based in Queueing Theory for the estimation of the performance of a NoC. The following are the contributions of this thesis:

- To the best of our knowledge, NoCSimulator is the only Queueing Theory based tool capable of estimate the performance of a NoC using traffic patterns based in real applications, thus, improving the quality of the results obtained when using a tool based in formal models.
- Also, to the best of our knowledge NoCSimulator is the only Queueing Theory based tool capable of model multiple NoC topologies as Mesh, Ring, Flattened Butterfly and Fat-tree. This widens the design space available to the system architect.
- Finally, NoCSimulator runs considerable faster than the state-of-the-art cycle-accurate simulator gem5, this, makes NoCSimulator suitable to be used at early stages of the design space exploration process.

## D. *Thesis Outline*

The thesis is organized into the following chapters:

- II – SoC and NoC Concepts.
- III - Review of formal models for Performance Estimation in Networks.
- IV - MCLS Traffic Suite and Selection of a Cycle-Accurate Simulator as Validation Tool.
- V – NoCSimulator overview.
- VI - Validation Tests for NoCSimulator.
- VII - Design Space Exploration Exercise.
- VIII - Conclusions

# II. SOC AND NOC CONCEPTS

**CONTENTS**



## A. *Introduction*

The main bottleneck of modern computer systems performance is the movement of data between the different components of the system (CPUs, memory, GPUs, etc.), and it influences other design objectives as well, for example, cost, occupied area, power consumption and reliability. Bus-based networks in Systems-on-Chip (SoC) have been used traditionally when the number of nodes is low, however, as the number of nodes increases the performance of bus-based networks suffers due to arbitration issues [2]. Thus, to keep up with the paradigm of SoC, system architects must explore new topologies for interconnection networks (in this context, Networks-on-chip, NoC), looking for desirable features as high throughput, low latency, low cost, less occupied area, low power consumption, etc. These topologies are characterized by the routing of data packets, rather than dedicated physical wires which connect the nodes directly, this because packet routing enables an efficient sharing of the available bandwidth of the network between the nodes. Then, a proper understanding of the design implications when selecting an NoC is essential for system success.

In the remainder of this chapter the main concepts necessary to give context to this thesis are presented. In section B, the main features of Systems-on-Chip are presented. In section C Network-on-Chip are discussed thoroughly, including concepts as topology, routing protocols, deadlock, and flow control. Section D is dedicated to the definition of the performance metrics of a NoC. Then, in section E Design Space Exploration (DSE) is defined. Also, in this section it is explained where in the DSE workflow, the tool presented in this thesis fits. Finally, some examples of DSE tools and techniques are discussed in section F.

B. *Systems-on-Chip (SoC)*

Moore's law states that the number of transistors in an integrated circuit (IC) is roughly doubled every two years [29]. Until the early 2000s, this meant that CPU manufacturers could improve CPU performance by building processors of increasing complexity that could run at higher and higher clock rates. However, smaller transistors working at lower voltages and high frequencies started to compromise CPU reliability. This drawback shifted the computer design paradigm from a single fast (and complex) processor on a chip to multiple slower (and simple) processors on a chip [30]. Nowadays, advancements in chip manufacturing technology allow the integration of several hardware components (processors, memory controllers, hardware accelerators, etc.) into a single integrated circuit known as System-on-Chip (SoC) reducing both manufacturing costs and system dimensions. As core counts increase, there is a corresponding increase in bandwidth demand to facilitate high core utilization. The processing power of cores is of no use unless data can be fed to them at the appropriate rates [31]. Figure II-1 shows an example of an SoC, where different hardware components like a CPU, a GPU, a camera controller, WiFi, and Bluetooth hardware, etc. are integrated into a single chip.

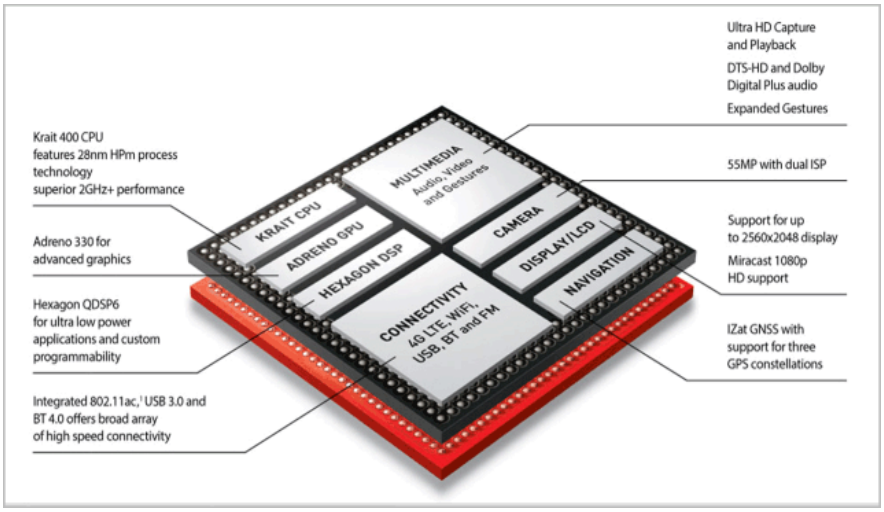


**Figure II-1.** Example of an SoC. Taken from [32].

Traditionally, Systems-on-Chip (SoC) designs utilize topologies based on shared buses. However, this type of interconnection usually scales efficiently only up to a few cores. For a larger number of processing elements, a more scalable and flexible solution is needed [31]. The solution consists of an on-chip data routing network consisting of communication links and routing nodes generally known as Network-on-Chip (NoC) [33], [34]. A generic NoC design divides a chip into a set of tiles, with each tile containing a processing element and a router. Each processing element is connected to the local router and each router is connected to other routers forming a packet-based on-chip network (see Figure II-2) [31].

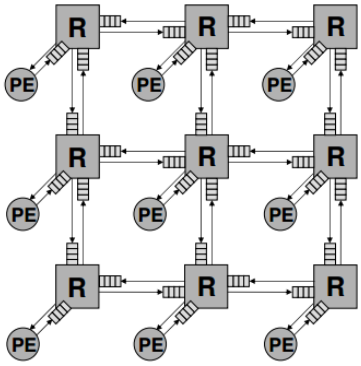


**Figure II-2.** A 3-by-3 mesh NoC. Taken from [35].

Multi-core architectures are now common in a variety of computing domains. For example, these architectures enable increased levels of server performance in data centers [36], [37]. Also, desktop applications, particularly graphics are already exploiting multi-core features [38], [39]. High-bandwidth communication is required for these throughput-oriented applications because communication latency can have a significant impact on the performance of multi-threaded workloads. Additionally, synchronization between threads will require low-overhead communication to scale to a large number of cores [1].

## C. *Networks-on-Chip (NoC)*

NoCs are replacing buses and crossbars in many-core chips systems. Such on-chip networks have routers at every node, connected to neighbors via short local on-chip links. Multiple communication flows are multiplexed over these links to provide scalability and high bandwidth. In multiprocessor systems-on-chip (MPSoCs) using an on-chip network promotes design isolation: MPSoCs utilize heterogeneous Intellectual Property (IP) blocks from a variety of vendors; with standard interfaces, these blocks can communicate through an on-chip network in a plug-and-play fashion [1].

When using an NoC, IP cores are attached to routers via dedicated network interfaces (see Figure II-3). Network interfaces are generally available for any core, thus, computation and communication concerns are decoupled at the network interface level, enabling a modular and plug-and-play oriented approach to system design [31].

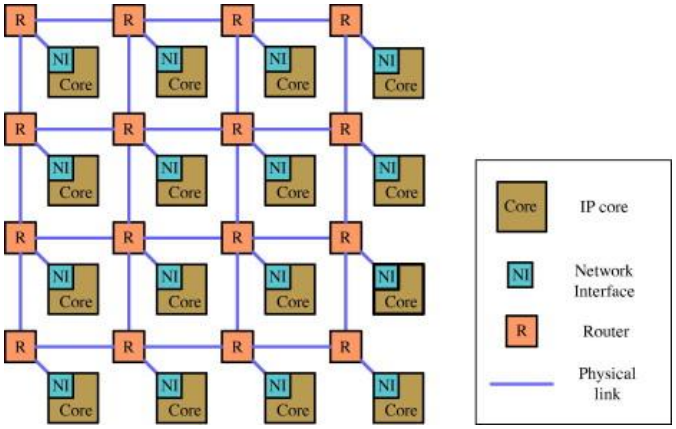


**Figure II-3.** Components of an NoC. Taken from [40].

Generally, NoCs can be designed, optimized, and verified by automated means [41], [42], leading to large savings in design time. Moreover, NoCs can be tuned in a variety of parameters (topology, buffering, data widths, arbitrations, routing choices, etc.), leading to higher chances of optimally matching design requirements [31].

*1) NoC Topology*

NoCs are composed of a set of router nodes and channels, and the topology of the network refers to the arrangement of these nodes and channels. The topology of an NoC can be compared to a roadmap. The channels (like roads) carry packets (like cars) from one router node (intersection) to another [2]. For example, the network in Figure II-3 consists of 16 nodes (routers), where each node has bidirectional channels that connect it to its neighbors. This particular network is a mesh topology.

The effect of topology on overall network cost-performance is profound. Topology determines the number of hops (or routers) a message must traverse as well as the interconnect lengths between hops, thus influencing network latency significantly [43]. As traversing routers and links incur energy, a topology's effect on hop count also directly affects network energy consumption. Furthermore, the topology dictates the total number of alternate paths between nodes, affecting how well the network can spread out traffic and hence support bandwidth requirements. The implementation complexity cost of a topology depends on two factors: the number of links at each node (node degree) and the ease of laying out a topology on a chip (wire lengths and the number of metal layers required) [43].

Figure II-4 shows how topologies can be classified. Bus-based topologies are used in applications where only a few PEs are present, due to its poor latency and throughput scaling with an increasing number of PEs [44]. Specifically, in this thesis, the following topologies are studied: Ring [45], Fat-tree [46], Butterfly (flattened) [47], and Mesh [48]. Each of these topologies has its advantages and disadvantages in terms of performance and they will be described in Chapter 6 of this thesis.

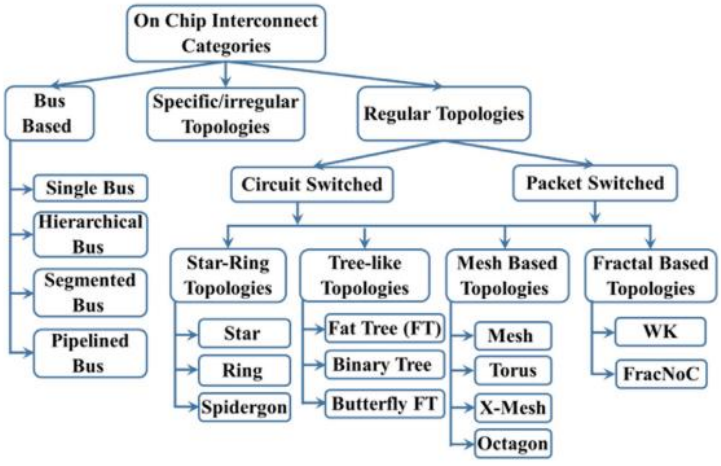


**Figure II-4.** Classification of topologies. Taken from [49].

A topology that provides multiple shortest paths between a given source and a destination node has greater path diversity than a topology where there is only a single path between a source and a destination node (path diversity can be measured for a given topology, for example, using a modified version of the Dijkstra algorithm [50]). Path diversity within the topology gives the routing algorithm more flexibility to load-balance traffic, which reduces channel load and thus increases throughput. Also, it enables packets to

potentially route around faults in the network [43]. For example, the Ring topology in Figure II-5 provides no path diversity, because there is only one shortest path between any pairs of nodes. On the other hand, the Mesh topology in Figure II-5 provides multiple shortest paths between source and destination nodes.

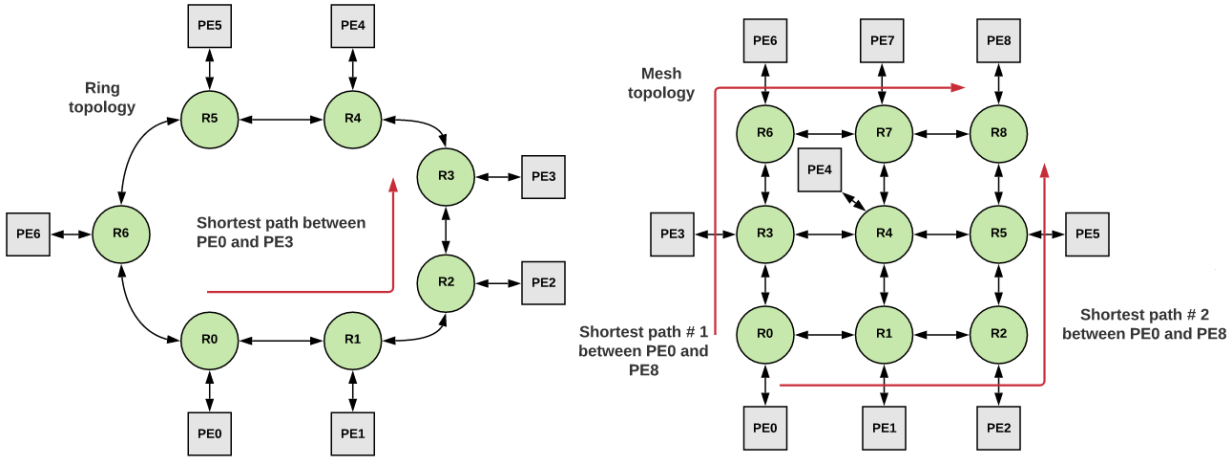


**Figure II-5.** Path diversity in Ring and Mesh topologies.

*2) NoC Routing Protocols*

While numerous routing algorithms have been proposed [51]–[54], the most commonly used routing algorithm in on-chip networks is Dimension-Ordered Routing (DOR) due to its simplicity. Dimension-ordered routing is an example of a deterministic routing algorithm, in which all messages from node A to node B will always traverse the same path. With DOR, a message traverses the network dimension-by-dimension, reaching the ordinate matching its destination before switching to the next dimension [55]. For example, in the Mesh topology of Figure II-6, if X-Y routing is used, packets are sent along the X-dimension first, and then in the Y-dimension. Thus, a packet traveling from (0,0) to (2,2) will first traverse 2 hops along the X dimension, arriving at (2,0), before traversing 2 hops along the Y-dimension to its destination.

Routing algorithms can be classified as minimal and non-minimal. Minimal routing algorithms select only paths that require the smallest number of hops between the source and the destination. Non-minimal routing algorithms [56], [57] allow paths to be selected that may increase the number of hops between the source and destination. In the absence of congestion, non-minimal routing increases latency and power consumption as additional routers and links are traversed by a message. With congestion, the selection of a non-minimal route that avoids congested links may result in lower latency for packets [55].

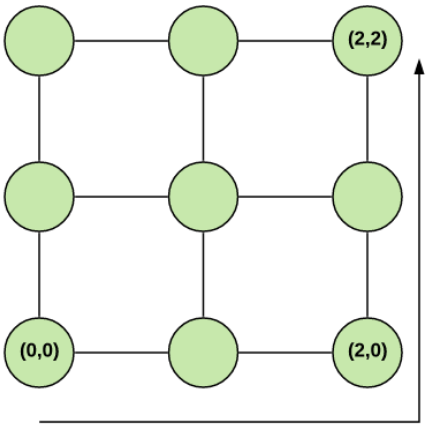


**Figure II-6.** DOR (X-Y) routing in Mesh topology.

*3) Deadlock avoidance*

When selecting a routing algorithm, not only its effect on latency, power consumption, and throughput must be considered, it also must guarantee deadlock freedom. A deadlock occurs when a cycle exists among the paths of multiple packets [55]. Figure II-7 shows four deadlocked packets waiting for links that are currently held by other packets, preventing any packet from making forward progress. In the figure, the packet entering router A from the South input port is waiting to leave through the East output port, but another packet is holding onto that exact link while waiting at router B to leave via the South output port, which is again held by another packet that is waiting at router C to leave via the West output port and so on [55].

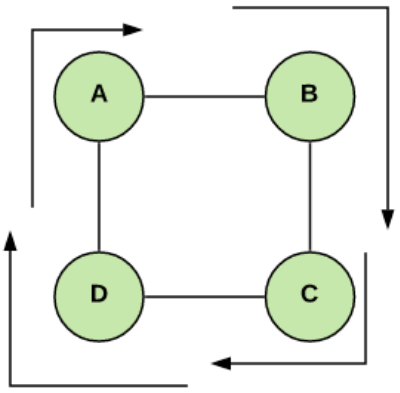


**Figure II-7.** Deadlock mechanism in routing protocols.

Deadlock is catastrophic to a network. After a few buffers are occupied by deadlocked packets, other packets are blocked on these buffers, paralyzing network operation. To prevent this situation, networks must either use deadlock avoidance methods (that guarantee that a network cannot deadlock, for example [58], [59]) or deadlock recovery (in which deadlock is detected and corrected, for example [60], [61]) [62].

*4) Flow Control*

Flow control governs the allocation of router buffers and links. It determines when buffers and links are assigned to packets, the granularity at which they are allocated, and how these resources are shared among the many packets in transit on the network. A good flow control protocol lowers the latency experienced by packets at low loads by not imposing high overhead in resource allocation, and drives up network throughput by enabling effective sharing of buffers and links between packets [63].

*a) Packets and Flits*

When a message is injected into the network, it is first segmented into packets, which are then divided into fixed-length flits (short for flow control units). For instance, a 128-byte cache line sent from a processing element (processor, memory controller, etc.) to another processing element will be injected into the network as a packet by the sender's network interface. The packet consists of a head flit that contains the destination address and another control information, body flits (that carry the cache line data), and a tail flit that signals the end of the packet [63]. Figure II-8 depicts a packet and its respective flits.

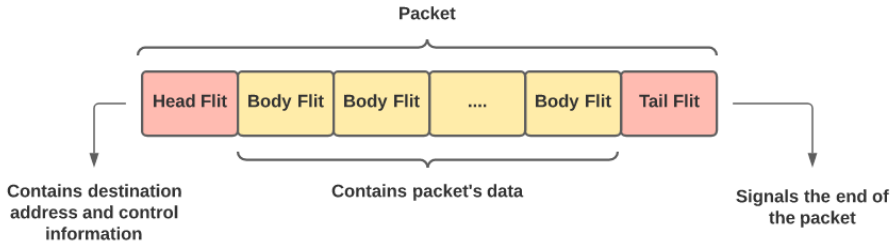


**Figure II-8.** A packet and its Flits.

*b) Wormhole Flow Control*

Wormhole flow control allows flits to move on to the next router before the entire packet is received at the current router. In wormhole flow control, a flit can depart the current router as soon as there is sufficient buffering for this flit in the next router. However, unlike other types of flow control protocols like store-and-forward and virtual cut-through [63], wormhole flow control allocates storage and bandwidth to flits rather than entire packets. This allows relatively small flit buffers to be used in each router, even for large packet sizes. While wormhole flow control uses buffers effectively, it makes inefficient use of link bandwidth. Tough it allocates storage and bandwidth in flit-sized units, a link is held for the duration of a packet's lifetime in the router. As a result, when a packet is blocked, all of the physical links held by that packet are left idle. Since wormhole flow control allocates buffers on a flit granularity, a packet composed of many flits can potentially span several routers, which will result in many idle physical links. Throughput suffers because other packets queued behind the blocked packet are unable to use the idle physical links [63].

Wormhole flow control can be implemented with fewer buffers than packet-based techniques (for example, store-and-forward and virtual cut-through), and due to the tight area and power constraints of NoCs, wormhole flow control is the common choice of systems architects [63].

*c) Virtual Channels*

A virtual channel (VC) is a separate queue in the router; multiple VCs share the physical link between two routers. By associating multiple separate queues with each input port, head-of-line blocking[1] can be reduced. Virtual channels arbitrate for physical link bandwidth on a cycle-by-cycle basis. When a packet holding a virtual channel becomes blocked, other packets can still traverse the physical link through other virtual channels. Thus, VCs increase the utilization of the physical links and extend overall network throughput [63]. Figure II-9 depicts a router with two virtual channels per input port. Thus, up to two different packets can pass through the same input port without blocking.

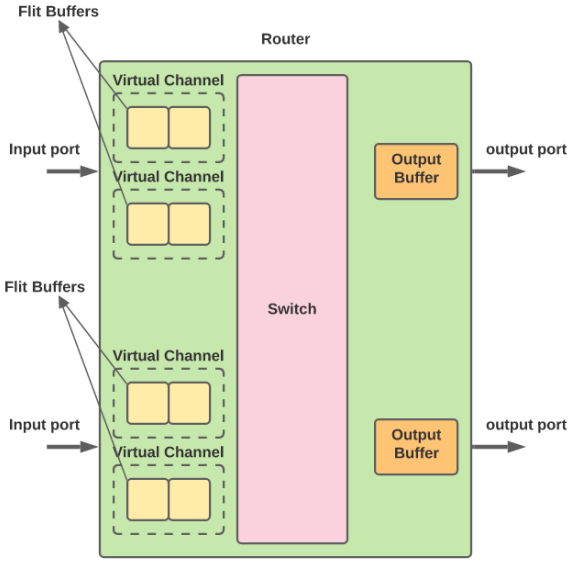


**Figure II-9.** Virtual Channels in a Router.

## D. *NoC Performance metrics*

The performance of an NoC is evaluated mainly by two metrics: latency and throughput. The latency of a network is defined as the time required for a packet to traverse the network from a source node to a destination node, and it includes delays due to waiting and processing of the packet on each router on its way. Latency depends not only on the topology of the NoC but also on the routing protocol and the flow control method implemented in the routers. The throughput of a network is the number of data units per unit of time that the network accepts at its input ports. Throughput is a property of the network and as with latency it also depends on the topology of the NoC, the routing protocol, and the flow control method used in the routers [1], [2]. The throughput per PE is a measure of the amount of traffic a PE adds to the total throughput of the network. Also, it can be used as an indication of the amount of processing work a PE does (for example, if a PE has a low throughput, it can be assumed that it is underutilized).

Figure II-10 shows the relationship between latency and traffic rate. As the throughput increases an increase in latency is observed due to the contention of packets in the network. The term zero load latency describes the minimum latency of the network and is defined as the time required to send a packet from a source node to a destination node when no other node in the network is sending packets. Saturation throughput indicates the maximum packet rate before experiencing a considerable latency in packets transiting the NoC [1].

[1] Head-of-line blocking: When upstream packets are blocked because a buffer is acquired by other packet.

Ideally, the system architect tries to design an NoC that offers minimum latency and high throughput for a given application under power, area, and cost constraints [33].

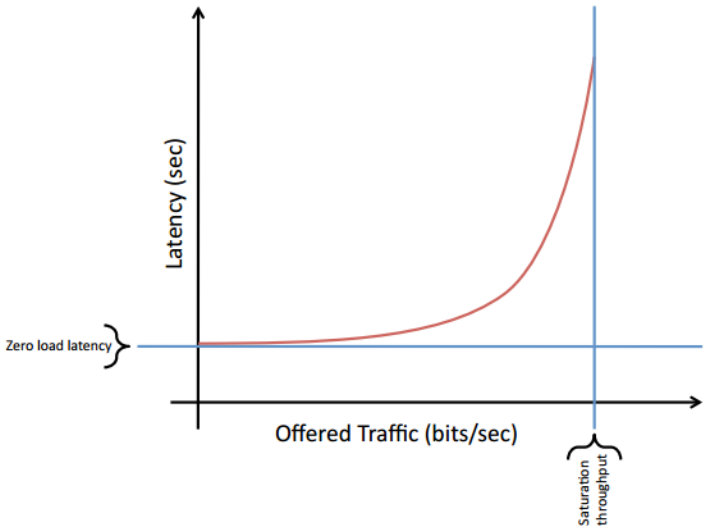


**Figure II-10.** Latency vs Throughput. Taken from [1].

## E. *Design Space Exploration*

Design Space Exploration (DSE) is defined as the characterization of various versions of a system in terms of previously defined metrics, to identify which of these versions meets design specifications [64]. One of the main challenges in SoC design is to estimate the essential characteristics of the system from the early design stages. This is because market pressures do not allow too much time (and money) to be spent on detailed implementations of the system that in the end might not meet the design specifications. Some of the questions to be answered while doing DSE are: What tasks should be implemented in hardware and which in software? What hardware components should be chosen and how should they be merged (architecture, topology)? How the tasks of the application should be mapped to the chosen hardware? What are the performance metrics of the communication network? Among others [64].

In general, DSE consist of a multi-objective optimization problem. The given constraints restrict the design space to feasible implementation options, whereas each possible design represents a System-on-Chip (SoC) solution. The key components of the decision vector are the software space (including algorithmic decisions and task-level partitioning aspects), the hardware architecture, and the temporal and spatial task mapping as shown in Figure II-11[3]. The complexity and size of the design space along with the time for evaluating a single design point prevents an exhaustive search to find the optimal solution. Consequently, only part of the complete design space can be elaborated, which naturally results in suboptimal solutions [3].

However, the number of investigated design points (system versions) closely relates to the time required for the individual evaluation process. This process consists of two major parts, given by the time spent in developing and describing the intended design point, as well as the time for finally evaluating the anticipated design. The first defines the modeling efficiency, whereas the latter depends upon the analysis and/or simulation time for the pure design evaluation [3].

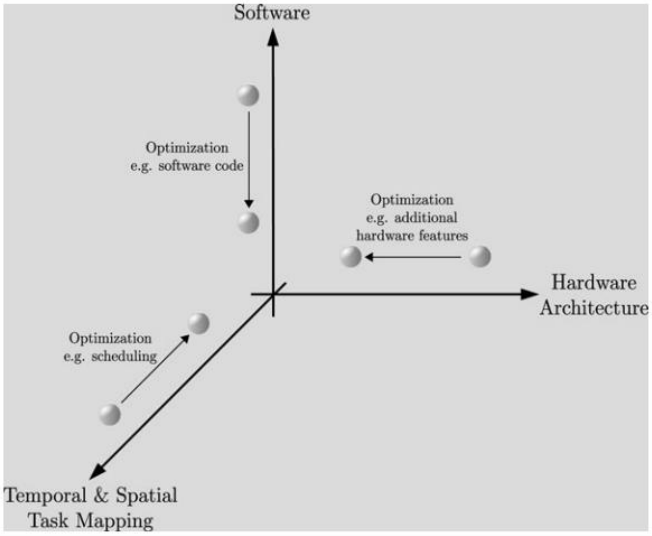


**Figure II-11**. The main axis of DSE. Taken from [3].

NoC Performance analysis is part of the activities of DSE, where different alternatives of the used topology should be evaluated to determine the most appropriate, to meet design specifications under certain restrictions such as costs, occupied area, energy consumption, etc.; restrictions that often come into conflict [64]. Figure II-12 presents the design flow used for the evaluation of different versions of an SoC, where for each version of the system (hardware + application), an estimate is made of the performance metrics (latency and throughput). These metrics are then analyzed to verify if they meet the system design specifications. If they do not comply, new versions of the system are proposed, for example, different topologies or application mappings, and the process starts again.

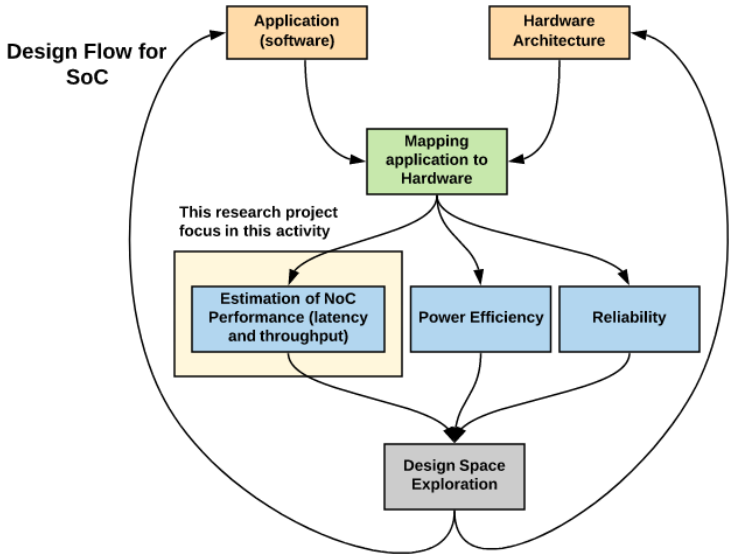


**Figure II-12.** DSE flow diagram for SoC design. Adapted from [64].

As indicated in Figure II-12, this research thesis is focused on NoC performance estimation, and it starts from an application that has been mapped to a specific hardware architecture (topology) where the tool proposed in this research thesis is used to estimate the performance of the NoC (latency and throughput).

Ideally, the tools used for DSE should be characterized by [64]:

- **Simple configurability** of hardware architecture and application mapping. That is, the task of obtaining different versions of the system should be simple.
- As far as possible, **short analysis time**, to test various alternative versions of the system, in a reasonable period of time.
- **Robustness in the presence of incomplete design information**, since generally, the low-level details of the system have not yet been defined.

F. *Survey of some DSE Tools and Techniques*

In [4] was proposed a four-step methodology for efficient design space exploration and tunable parameters optimization for multicore/many-core architectures. In the first step, an initial optimization algorithm based on one-shot searching was used to determine initial settings for each tunable parameter to within 51.26% of the final best setting for that parameter. The second step consisted of an intelligent set partitioning algorithm that separated the parameters into different ordered sets based on the significance of the parameters to the objective function and the exhaustive search threshold factor supplied by the system designer. The third and fourth steps consisted of exploring the subsets obtained from the second step by exhaustive search and greedy search, respectively, which improve on the initial settings obtained by the first step of the methodology, to find best settings to within 1.35% - 3.69% of the best settings obtained from a fully exhaustive search of the design space. The authors used the ESESC simulator [65] to run the different system configurations during the DSE process. [66] proposes a DSE methodology for NoCs composed of buffered and bufferless routers. Several router placement plans with a different number of buffered routers and positions were evaluated. Simulation results showed that intelligent router placement can achieve significant gains in performance under the same resource budget. Also, two techniques that take advantage of the NoC's heterogeneous nature (buffered and bufferless routers) for further performance improvement were proposed. BRAM[2] is an application mapping technique that can map applications close to buffered routers to improve performance, and BRAR[3] is a routing algorithm that sends data packets along the buffered routers to reduce the chances of lost packets. Simulations showed that applying such techniques can reduce data latencies by an average of 15%. [67] presents a framework to enable hardware acceleration of performance-critical parts of an application, by addressing the problem of automated hardware/software partitioning under power and area constraints to minimize the overall program latency. It implemented a flow in the LLVM compiler [68] to automate the detection and refactoring of hot regions (loops) of the application's code and making them candidates for acceleration on custom logic (hardware implementation). Then, a scalable hybrid approach based on an iterative search on Pareto frontiers in the design space, combined with an ILP[4] formulation was used to select a hardware/software mapping considering communication cost within and across processing elements and resource reuse among the hardware components. Experimental results on five benchmarks demonstrated that the proposed framework finds optimal solutions for a set of benchmarks whose optimal solutions were known while producing such solutions in less time. [49] introduced an approach for customizing on-chip interconnect (OCI) for SoC applications. The objective of this work is to build a framework for OCI (more specifically, custom topologies) evaluation without considering application traffic patterns, only available resources, for example, the number of routers available, or the number of links. To this end, the cycle-accurate simulator

[2] BRAM: Buffered Router Aware Mapping.
[3] BRAR: Buffered Router Aware Routing.
[4] ILP: Integer Linear Programming.

NIRGAM [69] was modified. Simulation results show that customizing OCIs, based on available resources, achieves better performance compared to the basic OCI architectures. Simulation results showed that the customization approach gives better performances for almost all application traffic patterns (Transpose, Uniform, Shuffle, and Bit-Reversal). These results show that WK and FracNoC (Fractal NoC) topologies perform well almost in all traffic patterns because of their properties, such as high degree of regularity, symmetry, scalability, and ease of extendibility. Another interesting property is their fractal structure, i.e., the network can be constructed hierarchically by grouping basic modules.

From above, the DSE process can be summarized as the use of optimization techniques to find the most suitable system configuration that satisfies a set of given constraints. In this process, cycle-accurate simulation is used to obtain performance data of each iteration of the system's candidates. This process is depicted in Figure II-13.

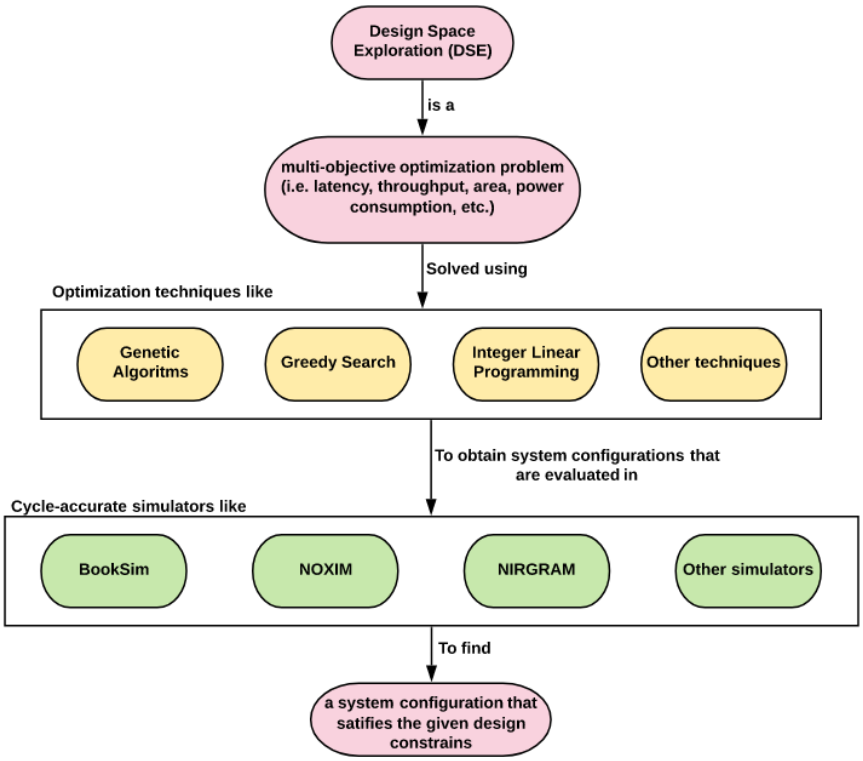


**Figure II-13.** Summary of the DSE process.

Figure II-13 can be thought of as a detailed view of Figure II-12, where the term "system configuration" in Figure II-13 corresponds to the mapping of an application in a given architecture, and the performance metrics in Figure II-12 are obtained using cycle-accurate simulation. Additionally, the optimization techniques shown in Figure II-13 enable the iteration loops shown in Figure II-12.

The tool developed and presented in this thesis can replace the cycle-accurate simulators used in DSE providing the following advantages:

1. Flexibility in system configuration: different topologies can be evaluated, not just Mesh topology. This is very important to widen the search space.

2. Capability to reproduce traffic patterns based on real applications, not just synthetic patterns. This is important to obtain more relevant results during the design process.
3. The performance of the NoC is estimated using formal models in contrast to cycle-accurate simulation. This can reduce the time spend doing DSE.

## G. *CONCLUSION*

In this chapter, important concepts to give context to this thesis were presented. These concepts included a description of the paradigm of System-on-chip (SoC) computer design. Also, NoC properties, like topology, routing protocols, performance metrics, etc. were exposed. Finally, the Design Space Exploration (DSE) process was described, and how of the tool developed in this thesis (NoCSimulator) fits in the DSE workflow was indicated.

# III. REVIEW OF FORMAL MODELS FOR PERFORMANCE ESTIMATION IN NETWORKS

**CONTENTS**



## A. *Introduction*

In section 1) it was shown that formal models offer an alternative to cycle-accurate simulation. The main feature of formal models in general, is its faster execution time when compared with cycle-accurate simulators. However, this comes with a trade-off in the accuracy of the results obtained using these models, due to their high level of abstraction and limited flexibility [20]. Nonetheless, this trade-off can be acceptable during the early stages of the design exploration process, where hundreds of configurations of a system must be explored to find suitable ones that meet design constraints like execution time, power consumption, throughput, latency, etc. Using a tool based in formal models can speedup this process, and once some suitable system prototypes have been filtered out, these ones can be further studied using a cycle-accurate simulator, to obtain more refined results. In this thesis, a tool based in formal models is proposed for this purpose, and in this chapter, a review of three formal models used for performance estimation of networks in general (not only NoCs) is presented. These formal models are Synchronous Dataflow Graphs (SDFGs), Queueing Theory, and Network Calculus. This review will expose the discussion and analysis that were carried out to select Queueing Theory as the formal model to be used in the tool proposed in this thesis. This chapter is intended to meet specific objective *b* (see section b)).

The remainder of this chapter is divided as follows: section B is devoted to Dataflow graphs, and includes its main concepts and a review of the state of the art. Then, Queuing Theory is discussed in section C, also including a review of the state of the art. Later, section D is about Network Calculus and a review of its state of the art. Finally in section E, an analysis of the advantages and disadvantages of each of these three formal models is done, and from this analysis Queueing Theory is selected as the most suitable to be used in this thesis.

B. *Dataflow Graphs*

A dataflow graph is a directed graph, where the vertices (actors) represent computation, and edges (arcs) represent FIFO (first-in-first-out) queues that take data values from the output of one computation to the input of another. Thus, edges represent data precedence between computations. Actors consume data (or tokens) from their inputs, perform computations on them (fire), and produce certain numbers of tokens on their outputs [70]. Several models based on dataflow with restricted semantics have been proposed (Petri Nets [71], Kahn Process Networks [72], Synchronous Dataflow Graph [73], etc.); these models lose the descriptive power of general dataflow in exchange for properties that facilitate formal reasoning about programs specified in these models, and are useful in practice, leading to simpler implementations of the specified computation in hardware or software.

*1) Synchronous Dataflow Graphs [74]*

The Synchronous Data Flow (SDF) model of computation was proposed in [75]. The SDF model poses restrictions on the firing of actors: the number of tokens produced (consumed) by an actor on each output (input) edge is a fixed number that is known at compile time. The number of tokens produced and consumed by each SDF actor on each of its edges is annotated in illustrations of an SDF graph by numbers at the arc source and sink respectively. In an actual implementation, arcs represent buffers in physical memory. The arcs in an SDF graph may contain initial tokens, which are referred to as delays. Arcs with delays can be interpreted as data dependencies across iterations of the graph. Delays are represent using bullets (•) on the edges of the SDF graph; more than one delay on an edge is indicated by a number alongside the bullet [76]. An example of an SDF graph is illustrated in Figure III-1.

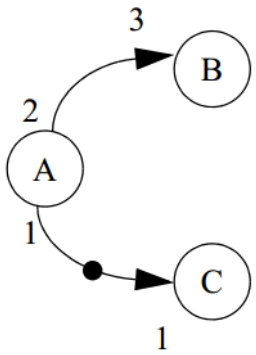


**Figure III-1.** Example of an SDF graph. Taken from [76].

Unbounded buffers imply a sample rate inconsistency, and deadlock implies that all actors in the graph cannot be iterated indefinitely. Thus, correctly constructed SDF graphs are those that can be scheduled periodically using a finite amount of memory. The main advantage of imposing restrictions on the SDF model (over a general dataflow model) lies precisely in the ability to determine whether or not an arbitrary SDF graph has a periodic schedule that neither deadlocks nor requires unbounded buffer sizes [76].

*a) Properties of SDFGs*

An SDF graph is compactly represented by its topology matrix. The topology matrix, known as $\Gamma$, represents the SDF graph structure; this matrix contains one column for each vertex and one row for each edge in the SDF graph. The $(i,j)th$ entry in the matrix corresponds to the number of tokens produced by the actor $j$

onto the edge $i$. If the $jth$ actor consumes tokens from the $ith$ edge, i.e., the $ith$ edge is incident into the $jth$ actor, then the $(i,j)th$ entry is negative. Also, if the $jth$ actor neither produces nor consumes any tokens from the $ith$ edge, then the $(i,j)th$ entry is set to zero [76].

For example, the topology matrix for the SDF graph in Figure III-1 is:

$$\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ 1 & 0 & -1 \end{bmatrix} \qquad \text{Equation III-1}$$

Where the actors A, B, and C are numbered 1, 2, and 3 respectively; the edges (A, B) and (B, A) numbered 1 and 2 respectively. **The repetitions vector $\boldsymbol{q}$** for an SDF graph with actors numbered 1 to $s$ is a column vector of length $s$, with the property that if each actor $i$ is invoked a number of times equal to the $ith$ entry of $q$, then the number of tokens on each edge of the SDF graph remains unchanged. Furthermore, $q$ is the smallest integer vector for which this property holds [76].

The repetitions vector for an SDF graph with consistent sample rates is the smallest integer vector in the null space of its topology matrix. That is, is the smallest integer vector such that $\Gamma q = 0$ holds. $q$ can be obtained by solving a set of linear equations; these are also called balance equations since they represent the constraint that the number of tokens produced and consumed on each edge of the SDF graph be the same after each actor fires a number of times equal to its corresponding entry in the repetitions vector [76]. For the example for Figure III-1 and Equation III-1:

$$q = \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix} \qquad \text{Equation III-2}$$

Then, if actors A, B, and C are invoked 3, 2, and 3 times respectively, the number of tokens on the edges remains unaltered (no token on (A, B) and one token on (A, C)). Thus, the repetitions vector in Equation III-2 brings the SDF graph back to its "initial state."

*b) Homogenous SDFG*

An SDF graph in which every actor consumes and produces only one token from each of its inputs and outputs is called a Homogeneous SDF Graph (HSDFG). An HSDFG actor fires when it has one or more tokens on all its input edges; it consumes one token from each input edge when it fires, and produces one token on all its output edges when it completes execution [76].

The repetitions vector defined in the previous section can be used to convert any general SDF graph to an equivalent HSDFG (for brevity the procedure won't be shown here, but it can be found in [76]). The resulting HSDFG has a larger number of actors than the original SDF graph. In fact, it has a number of actors equal to the sum of the entries in the repetition vector. In the worst case, the SDF to HSDFG transformation may result in an exponential increase in the number of actors [76], [77]. However, this transformation is necessary when constructing periodic multiprocessor schedules from multi-rate SDF graphs in a way that takes into account the available parallelism among different actor invocations [76].

*c) Inter-processor Communication Graph (IPC) and Maximum Cycle Mean (MCM)*

A self-timed schedule specifies actors assigned to each processor and specifies the order in which these actors must be executed. At run-time, each processor executes the actors assigned to it in the prescribed order. A self-timed schedule is modeled using an HSDFG, this graph known as the inter-processor communication graph (IPC) models the fact that actors assigned to the same processor execute sequentially, and it models constraints due to inter-processor communication. The IPC graph has a number of vertexes equal to the number of actors whereas the self-timed schedule specifies the actors assigned to each processor and the order in which they execute. The IPC graph has the same semantics as an HSDFG, and its execution models the execution of the corresponding self-timed schedule [78].

The iteration period for an IPC graph is given by:

$$T = \max_{cycle\ C\ in\ G} \left\{ \frac{\sum_{v\ in\ C} t(v)}{Delay(C)} \right\} \qquad \text{Equation III-3}$$

With $C$ representing a cycle of the IPC, $v$ a vertex (actor) in the cycle, and $t(v)$ the execution time of the actor. The quotient in Equation III-3 is called the **Cycle Mean** of the cycle $C$. $T$ in Equation III-3 is called the **maximum cycle mean (MCM)** of the IPC graph. If the IPC graph contains more than one cycle, then different cycles may have different asymptotic iteration periods, depending on their maximum cycle means. In such a case, the iteration period of the overall graph (and hence the self-timed schedule) is the maximum over the maximum cycle means of all the cycles, because the execution of the schedule is constrained by the slowest component in the system [78]. The throughput of the IPC is calculated as:

$$Throughput = \frac{1}{T} \qquad \text{Equation III-4}$$

From the last paragraph, it can be inferred that the IPC is necessary to model dataflow graphs where its actors are mapped to different processors in an NoC. Specifically, the concept of maximum Cycle Mean (MCM) is important for performance estimation and buffer sizing in an NoC as will be seen in the state-of-the-art review section.

*d) Latency Estimation*

According to [79], latency is the least studied performance metric for applications modeled with SDFGs compared to other performance indicators such as throughput and resource consumption (buffer sizing), with main results found mainly in research papers. In [74] the latency of an SDFG is calculated by first converting it into an equivalent HSDFG and then scheduling the HSDFG to determine the optimal latency metrics. However, as stated earlier, converting an SDFG into an HSDFG can lead to an exponential size graph and can increase the worst-case complexity of scheduling algorithms for HSDFG. Therefore, to analyze the latency of SDFGs it is more preferable to develop techniques working directly on SDFG instead of HSDFG. For example, [79] proposes a two-steps method for latency estimation. In the first step, An SDFG is transformed into a normalized graph [80], and then using periodic schedules the latency reachable by the normalized graph is computed. [81] presents a technique to compute

the minimum latency that can be achieved between firings of a designated pair of actors of an SDFG with throughput constraints. The technique is based on a new concept called throughput-constrained latency graph. The present static-order schedules for single-processor platforms, and for a multi-processor to exploit the available parallelism in an SDFG. In [82], a transformation of strongly connected HSDF graphs into timed automata is presented. These timed automata allow for the computation of exact end-to-end latencies because the correlation between the firing durations of different firings is taken into account. The transformation of HSDF graphs into an equivalent timed automata is possible because the number of tokens on edges in a strongly connected HSDF model is bounded. Therefore, buffers can be modeled using an extended timed automata which by definition have a finite number of states. This also guarantees that there is a maximum number of replicas of the same actor that can fire concurrently. This guarantees that there is a finite number of concurrent state machines, and thus states, required to model each HSDF actor.

This has been just an overview of SDFG theory and how throughput and latency in an SDFG are calculated. A more in deep treatment of the subject can be found in [74], [79], [81], [82].

*2) Review of the state of the art of SDFGs*

In [26] SDFGs are used to estimate the performance of an MJPEG application when some of its tasks are migrated from software to hardware. The communications interfaces between tasks implemented in hardware (i.e., an ASIC) and tasks implemented in software (i.e. task running in a processor) are modeled using additional actors that model the data transfer and latency in the communication interfaces (IPC actors). However, to obtain good estimations results, it is necessary to simulate the system in a cycle-accurate simulator, to obtain data about latency (the execution time of actors) and throughput (tokens produced/consumed) in the communication interfaces. The authors used the software SDF3 to simulate the SDFGs and estimate the throughput of the MJPEG application (frames/sec), obtaining close results to those of a cycle-accurate simulation. One possible drawback of this approach is the longer simulation time due to the increase of actors (IPC actors) that multiple tasks implemented in hardware would imply when an application with hundreds of tasks is modeled this way. In [83] SDFGs are used to dimension buffers of Networks Interfaces (the link between a processing element and a router) in an NoC. Similar to [26], an NI is modeled using actors that mimic the data transfer and latency of the NI. The NoC is modeled as actors whose execution time represents the latency between processing elements. The authors use a cell phone system as the case of study, and compare their results (buffer sizes in NIs) to those obtained using Network Calculus and cycle-accurate simulation. Their results show an improvement of over 40% to results obtained using Network Calculus and a 12% discrepancy to results using cycle-accurate simulation. As in [26], one possible drawback of this approach is the actor explosion problem when analyzing a system with hundreds of nodes and tasks. As stated in section c), the throughput of an SDFG is calculated using the MCM. This implies the transformation of the SDFG to an HSDFG which in general results in an exponential increase of actors, and the algorithms to calculate MCM can take a long time to execute. In [77], a new methodology for the estimation of throughput in SDFGs is presented. It consists of keeping track of the execution of the SDFG (the authors called the state-space) until a cycle in the execution of the SDFG is detected. Then, from the period of execution, the throughput of the SDFG can be calculated. At first, this approach would imply the extensive use of memory to store the state-space of the SDFG, but the authors apply various mechanisms to avoid this issue. The authors compare their methodology with the standard MCM methodology, obtaining throughput estimations of SDFGs of various orders of magnitude faster than the MCM methodology. In [84] a methodology for throughput estimation in hierarchical SDFGs is presented. Generally, when dealing with hierarchical SDFGs, flattening of the SDFGs was necessary to calculate their throughput (using the aforementioned MCM technique). For complex SDFGs, this flattening derives in the actor explosion

problem already mentioned and the execution time of algorithms for MCM is prohibitively high. The authors proposed a methodology based in a bottom-up simulation of each subgraph in the hierarchy, to obtain is state-space representation. Then, using (max,+) algebra and the state-space of each subgraph they obtain the data necessary to calculate the state-space of the subgraph at the next level of the hierarchy. This process continues until the graph at the last level of the hierarchy is reach, and the MCM of this SDFG is calculated to finally obtain the throughput of the hierarchical SDFG. The authors show that their methodology offers speed-ups of several orders of magnitude when compared with the standard technique of flattening the hierarchical SDFG. In [85], a scheduling methodology oriented to maximize the throughput of SDFGs in multi-core systems is presented. This methodology is based in Constrained Programming (CP), and differs of the standard technique for scheduling of SDFGs in that the mapping and ordering of actors is not done in separated design stages. Here, starting from an HSDFG during the search process (based in CP) with the optimization target of improving throughput, whenever mapping and ordering decision are taken, the graph is modified accordingly. During the throughput calculations, actors and edges that do not contribute data are ignored, something that improves significatively the execution time of the algorithm. The authors report that their methodology can calculate throughput results for SDFGs that using the standard methodology are time prohibitively.

### C. *Queueing Theory*

Queueing theory refers to the mathematical study of waiting lines or queues. In general, a queueing system is composed of something that needs to be served (a customer, a job, or in the case of this thesis, a data packet traveling in an NoC) and an entity that provides that service (a cashier, a server, or in the case of this thesis, a router in an NoC). To understand queueing theory, it is important to state some foundations of probability theory. In particular, the Exponential and Poisson distributions will be reviewed.

#### *1) Exponential and Poisson Distributions*

The exponential distribution with parameter with packet rate $\lambda$ is given by $\lambda e^{-\lambda t}$ for $t \geq 0$. If $T$ is a random variable that represents inter-arrival times with the exponential distribution, then the probability that the time gap between packets $T$ be less or equal than t is $P(T \leq t) = 1 - e^{-\lambda t}$. Also, the probability that the time between packets $T$ be greater than $t$ is $P(T > t) = e^{-\lambda t}$. This distribution adjusts well to modeling packet inter-arrival times or service times as explain next. First, the exponential function is a strictly decreasing function of $t$. This means that after arrival has occurred, the amount of waiting time until the next arrival is more likely to be small than large. Another important property of the exponential distribution is what is known as the no-memory property. The no-memory property suggests that the time until the next arrival will never depend on how much time has already passed. This makes intuitive sense for a model where packet arrivals are independent of one another (see Equation III-6) [86].

The Poisson distribution is used to determine the probability of a certain number of arrivals $k$ occurring in a given time period. The Poisson distribution with parameter $\lambda$ is given by:

$$P(k = n) = \frac{(\lambda t)^n e^{-\lambda t}}{n!} \qquad \text{Equation III-5}$$

Where $n$ is the number of arrivals. In this equation, it can be seen that if n = 0, the Poisson distribution becomes $e^{-\lambda t}$ which is equal to $P(T > t)$ from the exponential distribution. This relation between these

distributions connects the probability that zero arrivals will occur in a given time period, with the probability that an inter-arrival time will be of a certain length. The inter-arrival time here, as explained earlier, is the time between packet arrivals, and thus is a period of time with zero arrivals [86].

*2) The Input Process*

In queueing theory, the input process is defined as the statistical characteristics that model the traffic arriving at a server. To begin modeling an input process, define $t_i$ as the time when the $i^{th}$ packet arrives. For all $i \geq 1$, define $T_i = t_{i+1} - t_i$ to be the $i^{th}$ inter-arrival time (time elapsed between packet $i$ and packet $i+1$). It is also assumed that all $T_i$'s are independent, continuous random variables, and represented by the random variable $A$ with probability density $a(t)$. Typically, $A$ is chosen to have an exponential probability distribution with parameter $\lambda$ defined as the arrival rate, then, $a(t) = \lambda e^{-\lambda t}$. If $A$ has an exponential distribution, then for all nonnegative values of $t$ and $h$, the probability that the inter-arrival time $A$ be greater than $t + h$ given that $A$ is already greater than $t$ is:

$$P(A > t + h \mid A \geq t) = P(A > h) \qquad \text{Equation III-6}$$

This is an important result because it reflects the no-memory property of the exponential distribution (the arrival time of a packet does not depend on the arrival time of past packets). [86].

*3) The Output Process*

In queueing theory, the output process is defined as the statistical characteristics that model the traffic departing a server. Similar to the input process, the analysis of the output process begins by assuming that service times of different packets are independent random variables represented by the random variable $S$ with probability density $s(t) = \mu e^{-\mu t}$. $\mu$ is defined as the service rate, with units of packets per unit of time. Generally, the output process is modeled as an exponential random variable, as it makes calculation much simpler [86].

*4) Queueing disciplines*

Generally, queues operate like a grocery checkout line. That is to say when an arrival occurs, it is added to the end of the queue and it is not served until all of the arrivals that came before it are served in the order they arrived. Although this is a very common method for queues to be handled, it is not the only one. The method in which arrivals in a queue get processed is known as the queuing discipline. This particular example outlines a first-come-first-serve (also known as First-in-First-out) discipline or an FCFS (FIFO) discipline. Other possible disciplines include last-come-first-served (also known as Last-in-First-out) or LCFS (LIFO), and service in random order, or SIRO. While the particular discipline chosen affects waiting times for particular customers, the discipline generally doesn't affect important outcomes of the queue itself, since arrivals are constantly receiving service regardless of the discipline [86]. In general, routers in an NoC serve packets using a FIFO discipline.

*5) Kendall-Lee Notation*

Kendall-Lee notation is used to describe queueing systems. This notation consists of six abbreviations for queue characteristics separated by slashes. The first and second characteristics describe the arrival and service processes based on their respective probability distributions. For the first and second characteristics, *M* represents an exponential distribution, *E* represents an Erlang distribution, and *G* represents a general distribution. The third characteristic gives the number of servers working together at the same time, also known as the number of parallel servers. The fourth describes the queue discipline. The fifth gives the maximum number of customers allowed in the system. The sixth gives the size of the pool of customers that the system can draw from. For example, *M/M/*1*/FIFO/*inf/inf represents a queue with 1 server, exponential arrival times, exponential service times, a FIFO queue discipline, an infinite capacity, and an infinite population pool to draw from [86]. Figure III-2 is a representation of this type of queue.

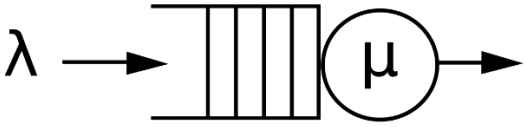


**Figure III-2.** An M/M/1/FIFO/inf/inf queue.

*6) Little's Law*

In queueing systems, it is useful to determine various waiting times and queue sizes for particular components of the system to analyze how the system will behave. Define $E[N]$ to be the average number of customers in the queue at any given moment assuming that the steady-state has been reached. $E[N]$ can be broken down into $E[N_q]$, the average number of customers waiting in the queue, and $E[N_s]$, the average number of customers in service. Since customers in the system can only be waiting in the queue or being served, $E[N] = E[N_q] + E[N_s]$. Similarly, define $E[T]$ as the average time a customer spends in the queuing system. $E[T_q]$ is the average amount of time spent in the queue itself and $E[T_s]$ is the average amount of time spent on the server. Then, $E[T] = E[T_q] + E[T_s]$. Defining λ as the arrival rate into the system, that is, the number of customers arriving the system per unit of time, it can be shown that [86]:

$$E[N] = \lambda * E[T] \qquad \text{Equation III-7}$$

$$E[N_q] = \lambda * E[T_q] \qquad \text{Equation III-8}$$

$$E[T_s] = \lambda * E[T_s] = \frac{\lambda}{\mu} = \rho \qquad \text{Equation III-9}$$

Equation III-7 to Equation III-9 are known as the Little's Law formulas.

In Equation III-9 ρ as the utilization factor of the server, or fraction of time that the server is busy. Then using ρ, Little's Law formulas can be rewritten as [87], [88]:

$$E[N] = \frac{\rho}{1-\rho} \quad \text{Equation III-10}$$

$$E\left[N_q\right] = E[N] - \rho \quad \text{Equation III-11}$$

$$E[T_q] = \frac{E[T]-\rho}{\lambda} \quad \text{Equation III-12}$$

Complex applications include networks of queues with different inter-arrival and service time distributions, and generally, analytic solutions do not exist for this type of systems. An example of these type of systems are the NoCs studied in this thesis. These kinds of systems are usually studied with the help of discrete-event simulation [89].

*7) Review of the state of the art of Queueing Theory*

[25] presents an analytical model for the study of the performance of an NoC based on the queueing theory of finite capacity and constant service time (model called M / D / 1 / B process). In contrast to similar works that focus on the estimation of latency and traffic rate, the probability of "flow-control feedback" or filling of buffers is the main contribution of this article, since it is an important factor in the latency and traffic rate behavior of an NoC. The derived model was used to calculate the probabilities of filling the routers' buffers for given traffic rates and service times, obtaining results very similar to those obtained using a cycle-accurate simulation. In [35] an analytical model based on queuing theory is derived for an NoC with routers with constant service time. Here, each router is modeled as a server that has a fixed service time *T*. In this paper, the hypothesis that the traffic on intermediate routers of the NoC can be described as Poisson traffic is rejected, and empirical equations are derived to estimate the packet delays in router buffers. [90] presents a delay and link utilization analysis methodology for wormhole-based NoCs with a different number of virtual channels per link and different link capacities. The average latency per flow is analyzed by calculating its three components: (1) The time it takes the head-flit to leave the source queue (queuing time at the source); (2) The time it takes the head-flit to reach its destination (path acquisition time) and (3) The time it takes the rest of the packet to reach its destination (transfer time). The link utilization is estimated based on the path acquisition time and transfer time. The authors show that their methodology can do estimations in agreement with those obtained using cycle-accurate simulation. In [91] a method for the analytical modeling of the performance of the *state-of-the-art single-cycle multi-hop synchronous repeated traversal (SMART) NoC* [92] is presented. The main feature of this type of NoC is that packets can bypass intermediate routers. The authors propose an analytical model for calculating the "stopping probability" of a packet at intermediate routers using the M/G/1/k and G/G/1/k queuing models. For latency prediction, the model has an average error between 2.5 and 8.4 percent. The authors report that their model is two orders of magnitude faster than the cycle-accurate GARNET network simulator [18]. [93] proposes a model for the mesh-inspired *de Bruijn* topology for NoCs. The authors designed a deadlock-free routing algorithm for the proposed topology. Additionally, this paper introduces an analytical model to predict the average latency of the *de Bruijn* topology based on an M/G/1 model. The authors show that this topology can

outperform its equivalent mesh topology in terms of network performance and energy dissipation. The proposed analytical model provides a speedup of 400% in comparison with a cycle-accurate simulation.

## D. *Network Calculus*

Network Calculus consists of the study of networks (originally Internet) based on enveloping functions that bound traffic rate and processing power of nodes in the network. The main performance metrics studied in Network Calculus are latency and buffer occupation. In Network Calculus, worst-case estimations about the performance of a network are obtained, the opposite of Queueing theory where mean performance values are obtained. Thus, Network Calculus is used when network designs need to meet Quality of Service (QoS) constraints [94], [95].

### *1) Input and Output Functions*

Data flow traffic (not related to dataflow in SDFG) is described using the cumulative function $R(t)$, defined as the number of data units (bits, flits, packets) seen in the time interval $[0, t]$. By convention, $R(0) = 0$, unless otherwise specified. Function $R$ is always increasing, that is, $R(t) \geq 0;\ t \geq 0$. In real systems, there is always a minimum granularity (bit, flit, or packet), therefore discrete time with a finite set of values for $R(t)$ could always be assumed. However, it is often computationally simpler to consider continuous time, with a function $R$ that may be continuous or not. If $R(t)$ is a continuous function, it is called a fluid model. Otherwise, the convention is that the function is either right or left-continuous [94].

To illustrate the terminology and convention previously presented, Figure III-3 shows examples of input and output functions arriving and departing from a given system (e.g., a router in an NoC). $R1$ and $R1^*$ are continuous function in continuous time (fluid model); it is assumed that packets arrive bit by bit, for a duration of one time unit per packet arrival. R2 and R2$*$ show continuous time with discontinuities at packet arrival times (times 1, 4, 8, 8.6, and 14); here it is assumed that packet arrivals are observed only when the packet has been fully received; the dots represent the value at the point of discontinuity; by convention, the function is left- or right-continuous. R3 and R3$*$ show a discrete-time model; the system is observed only at times 0, 1, 2...[94].

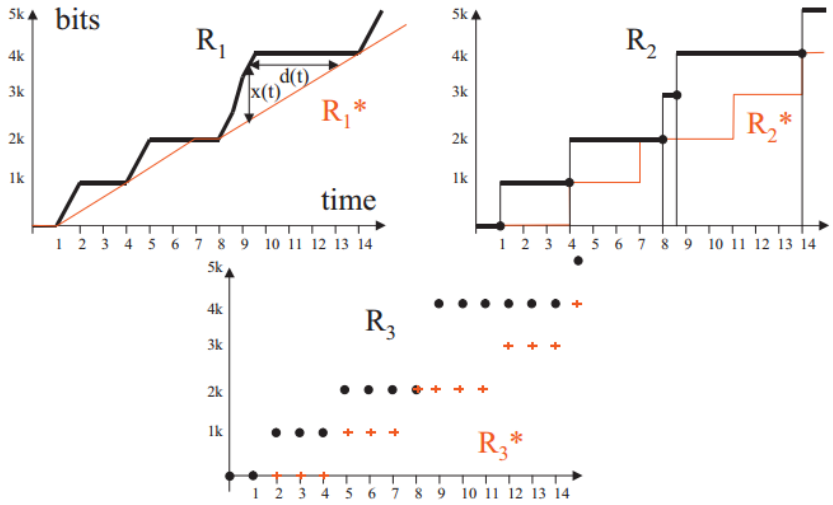


**Figure III-3.** Examples of cumulative functions. Taken from [94].

Consider a system *S*, which is viewed as a black-box; *S* receives input data, described by its cumulative function $R(t)$, and delivers the data after a variable delay. Call $R^*(t)$ the output function, namely, the cumulative function at the output of system *S*. System *S* might be, for example, a single buffer served at a constant rate, a complex communication node, or even a complete network. Figure III-3 shows input and output functions for *S*, where every packet takes exactly three time units to be served. With output function $R_1^*$ (fluid model) the assumption is that a packet can be served as soon as a first bit has arrived and that a packet departure can be observed bit by bit, at a constant rate. For example, the first packet arrives between times 1 and 2 and leaves between times 1 and 4. With output function $R_2^*$ the assumption is that a packet is served as soon as it has been fully received and is considered out of the system only when it is fully transmitted. Here, the first packet arrives immediately after time 1 and leaves immediately after time 4. With output function $R_3^*$ (discrete time model), the first packet arrives at time 2 and leaves at time 5 [94].

*2) Backlog and Virtual Delay*

From the input and output functions, the following quantities are derived:

- **The backlog** at time t is $R(t) - R^*(t)$.
- **The virtual delay** at time t is $d(t) = inf\{\tau \geq 0 : R(t) \leq R^*(t + \tau)\}$

The backlog is the amount of data units (bits, flits, packets) that are held inside the system; if the system is a single buffer, it is the queue length. In contrast, if the system is more complex, then the backlog is the number of data units "in transit", assuming that we can observe input and output simultaneously. The virtual delay at time t is the delay that would be experienced by a data unit arriving at time t if all data units received before it, are served before it [94]. In Figure III-3, the backlog, called $x(t)$, is shown as the vertical deviation between input and output functions. The virtual delay is the horizontal deviation. If the input and output function are continuous (fluid model), then $R^*(t + d(t)) = R(t)$, and $d(t)$ is the smallest value satisfying this equation.

From Figure III-3, it can be seen that the values of backlog and virtual delay differ for the three models. Thus the delay experienced by the last bit of the first packet is $d(2) = 2$ time units for the first subfigure; in contrast, it is equal to $d(1) = 3$ time units on the second subfigure. Similarly, the delay for the fourth packet on subfigure 2 is $d(8.6) = 5.4$ time units. In contrast, on the third subfigure, it is equal to $d(9) = 6$ units; the difference is the loss of accuracy resulting from discretization.

*3) Arrival Curves and Service Curves*

Generally, the exact behavior of a system is unknown at the design stage, or too complex to be handled. In Network Calculus, this is approached by abstracting the flow and server behavior by contracts, called arrival curves and service curves. For a given data flow $R(t)$, **the arrival curve** bounds the amount of data during any interval of time. $\alpha$ is an arrival curve for $R(t)$ if $R(t+s) - R(t) \leq \alpha(s)$ for all $s$ and $t$, which translates into "the amount of data that arrived between time $t$ and $t + s$ is less than $\alpha(s)$. Similarly, a guarantee on the server is to bound the minimum amount of data that can be processed during any interval of time by the server. If $\beta(s)$ is the minimum amount of data that the server can process in any time interval of length s, then β is a **service curve** [95].

the (min, plus) convolution ($*$ operator, defined in [94], [95]) relates the cumulative processes with the arrival and service curves:

$$R(t) \leq R(t) * \alpha(t) \; and \; R^*(t) \geq R(t) * \beta(t) \qquad \text{Equation III-13}$$

From this modeling, it is now possible to compute bounds from the curves. The maximum delay is bounded by the horizontal distance between $\alpha$ and $\beta$, and the maximum backlog is by the vertical distance of the curves. Moreover, $\alpha \oslash \beta$ is an arrival curve for $R^*(t)$, where $\oslash$ is the (min, plus) deconvolution which is defined in [94], [95].

*4) Review of the State of the Art of Network Calculus*

In [96], Network Calculus is used to estimate backlog (number of packets queued) and delay (latency) in an NoC. The authors derive the arriving curves and departure curves of the routers in the NoC using traffic rate information (synthetic) of the PEs and the service time of the routers. The authors compare their results with those obtained using a cycle-accurate simulator showing good accuracy. In [97] a formal model of an NoC is built using a theorem prover language called *Isabelle* [98], and Network Calculus is used to estimate the worst-case latency in the NoC. The authors present an approach that is based on the simulation of formal models of the elements of the NoC (PEs and routers). An important contribution of this article is the high-level modeling of the coherence protocol of the NoC, which permits the estimation of worst-case latency using more realistic traffic patterns, and not just synthetic traffic. In [99] a branch of the Network Calculus theory known as Stochastic Network Calculus is used to estimate the worst-case latency in soft real-time applications, that is, in applications where the loss of packets within the NoC is not critical to the operation of the application. Thus, the worst-case latency estimates are less conservative compared to the estimates obtained using standard Network Calculus, which allows the routers' buffers to be better sized. [100] introduces a Buffer-aware delay methodology for NoCs based on Network Calculus, to better estimate the worst-case end-to-end delay (latency) of the NoC. An important result of this paper was its capability to estimate the worst-case latency of an application implemented in real MPSoC hardware (TILE-Gx8036 [101]).

E. *Comparison of formal models for the Estimation of NoC Performance*

After the review of different methods for performance estimation of an NoC is time to select the one that is going to be used in this thesis.

From the discussion of SDFGs and its respective review of the state of the art, is evident that to model an application running in an MPSoC is necessary to have good estimations of the latency of packets traversing the NoC. The reason for this is that an SDFG is not a one-to-one representation of an application mapped to an MPSoC but a higher level of abstraction of the scheduling and data dependencies of the application's tasks, which is independent of the topology of the MPSoC where the application is running. For this reason, as stated in section c) is necessary to create an Inter-processor Communication graph (IPC) to model the transmission of data between actors that are mapped to different processors on the MPSoC. To construct an IPC graph, the SDFG or HSDFG (depending on the throughput estimation methodology used) is augmented by adding "communication actors" between actors mapped to different processors. The execution time of the "communication actors" serves to model the packet latency between the processors. This is illustrated in Figure III-4.

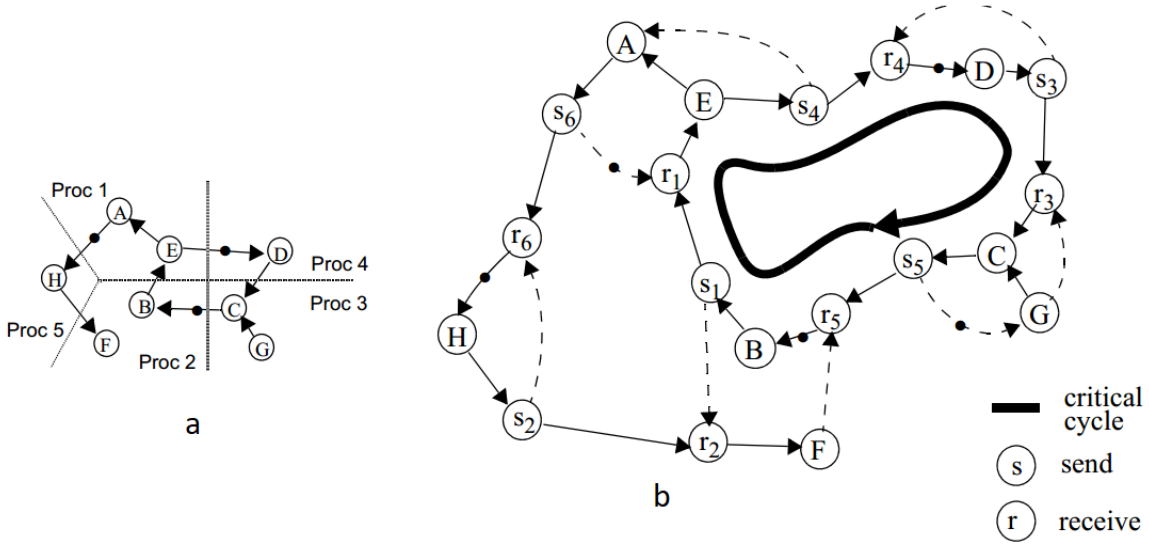


**Figure III-4.** a) An SDFG mapped to a 5-processors SoC. b) Its equivalent IPC graph. Taken from [78].

Figure III-4a shows how the SDFG is mapped to a 5-processor SoC, however, it says nothing about the communication network between the processors. It could be a bus, a mesh, a ring, etc. In Figure III-4b, its equivalent IPC graph is shown. Here, the "communication actors" are named "s" (send) and "r" (receive). The critical cycle shown in the figure is responsible for the MCM calculation in Equation III-3. As stated earlier, assigning proper execution times to these actors is necessary to model the communication network properly. Generally, this is done using a cycle-accurate simulator to obtain packet latency data between the actors mapped to different processors [26], [83], something that could take much time. Also, latency is dependent on the underlying topology, so, if several topologies must be investigated (as in DSE), multiple cycle-accurate simulations must be done, something that could be time-prohibitively. Additionally, SDFG analysis is more oriented to the performance of the application as a whole, for example, throughput is measured as the number of executions of actors per unit of time, and latency is measure as the time necessary to execute all actors of the SDFG. Then, a direct estimation of the NoC performance is not readily available (maybe a state-space technique as the one reported in [77] can be used).

Network Calculus is a design methodology oriented to offer Quality-of-Service (QoS) guarantees for network performance. As reported in the review of the state of the art (section 4)), for NoCs its principal use case is the estimation of worst-case latency of packets traveling the NoC and worst-case buffer occupancy in routers of the NoC. The estimation of these parameters can be done using closed formulas and information of traffic injection rate of the PEs (data flows) and data about the service rate of the routers. This, reduce greatly the computation time when compared with a cycle-accurate simulation as reported in [96], [100]. Worst-case performance estimations are of great interest for real-time applications, where strict performance specifications must be met for successful application execution. However, as reported in [99] these worst-case estimations are in general very conservative and can lead to an over dimensioning of the buffers of the NoC, leading to an increase of costs, area, and energy consumption of the NoC. Additionally, for applications with no real-time constraints (as those used in this thesis), the designer is interested mainly in the average-case performance of NoC (mean latency and mean throughput) because it is the most common mode of operation of the application.

Queueing theory is used to estimate mean performance metrics in queueing systems. Specifically, for NoCs, based on the many papers published in the last 20 years, it is maybe the most used methodology for formal estimation performance metrics. With Queueing Theory, important performance metrics of the NoC like mean latency, mean throughput, mean number of flits in routers' buffers, etc. are readily available. However, most of the state of the art consists in the use of synthetic traffic patterns for example [25], [35], [93] something that limits its applicability when designing an NoC for a real application. Also, most of the state of the art is focused on the mesh topology, something that limits the design space available to the designer. For basic networks, closed formulas for performance estimation exists, but for complex networks and traffic patterns, the help of discrete-event simulation is necessary [89]. However, this method is in general much faster than a cycle-accurate simulation as reported for example in [102].

Table III-1 presents a summary of the key points presented in this section. Having in mind that the objective of this thesis is the development of an NoC performance estimation tool that can handle traffics patterns based on real applications and multiple topologies, from the analysis done in this section and summarized in Table III-1, **Queueing Theory** was chosen as the formal model used in the development of this thesis to estimate NoC performance under several topologies and requirements, which were implemented in the tool proposed and described on V.

**Table III-1.** Comparison of formal models (key points).

| Formal model | Throughput | Latency | Comments |
|---|---|---|---|
| **SDFG** | Focused mainly on actor execution per unit of time. Requires simulation methods. | Focused mainly on the execution time of an SDFG. Requires simulation methods. | Limited to applications that can be represented by an SDFG. Cycle-accurate simulation is necessary to estimate the latency of inter-processor communications. The exponential actor explosion when modeling inter-processor communications could be a problem. |
| **Queueing Theory** | Requires simulations for networks of queues with complex traffic patterns. | Requires simulation for networks of queues with complex traffic patterns. | Other metrics of a queue can be estimated. For example, the mean number of flits in the buffer, utilization factor of routers, etc. |
| **Network Calculus** | Focused on QoS in netwoks, i.e., offering throughput guarantees for a given application. | Focused on QoS in netwoks, i.e., offering latency guarantees for a given application. | Used mainly for real-time applications. Specifically, for estimations of worst-case latency and buffer sizing. |

## F. *CONCLUSION*

In this chapter, a review of formal models for performance estimation of networks was done. The formal models studied were Synchronous Dataflow Graphs (SDFGs), Network Calculus and Queueing Theory. For each formal model a description of its main concepts and results along with a review of the state of the art was presented. After analyzing the advantages and disadvantages of each formal model, Queueing Theory was selected as the formal model used in the tool developed in this thesis (NoCSimulator). The main reason because, Queueing Theory is readily oriented to the estimation of NoC performance metrics like throughput and latency. From the review of the state of the art of NoC Performance estimations based on Queueing Theory it was found that it consists mainly in the use of synthetic traffic patterns (which have little relation to the traffic patterns of real applications) and the use of Mesh topologies. This, offers new

research opportunities (e.g., estimation of NoC Perfomance using traffic patterns of real applications and for other topologies, not only Mesh) that are explored in this thesis. With this chapter, specific objective *b* (see section b)) is considered fulfilled.

# IV. MCLS TRAFFIC SUITE AND SELECTION OF A CYCLE-ACCURATE SIMULATOR AS VALIDATION TOOL

**CONTENTS**



## A. *Introduction*

When studying NoC performance using a cycle-accurate simulator, generally, the benchmarks used are based on synthetic traffic patterns like uniform traffic, tornado, or hot-spot. These traffic patterns are relatively simple to generate, but offer limited insight into NoC performance, due to the lack of resemblance of these traffic patterns to the ones obtained from a real application. Another option is to use trace-based data obtained from full-system cycle-accurate simulations. These traces record every transaction at memory level (mainly, cache memory messages like loads, stores and cache coherence messages), and can be used to reproduce the traffic patterns of a real application, thus, offering better insight into NoC performance. However, the low-level nature of traced-based data makes it unsuitable to be used in a simulation tool based in Queueing Theory. Then, until recently only synthetic traffic patterns were available to be used with Queueing Theory based tools.

In this chapter, an NoC traffic suite called Multi-constraint System Level Suite (MCSL) is presented [28]. This suite was developed at the Big Data Systems Laboratory of the University of Hong Kong of Science and Technology [103], and includes stochastic representations of traffic patterns of several scientific applications running on multiple NoC topologies. The MCSL was originally conceived to speedup cycle-accurate simulations, but the stochastic representations of its applications are high-level enough to be used in a tool based in Queueing Theory like NoCSimulator. This makes NoCSimulator, to the best of our knowledge, the first Queueing Theory based tool capable of estimating the performance of a NoC with traffic patterns based on real applications.

The topics discussed in this chapter are described next. Section B presents the features of interest of the MCSL NoC traffic suite. Additionally, to validate the results obtained using NoCSimulator, it is necessary to use a cycle-accurate simulator, thus, a brief review of popular NoC cycle-accurate simulators is provided. Specifically, section C is dedicated to the Booksim simulator, section D is dedicated to the Noxim simulator, and section E is about the gem5 simulator. In section F, after an analysis of the features of each of the cycle accurate simulators, gem5 is selected as the cycle-accurate simulator to be used as validation tool. However, gem5 by default is not compatible with the MCSL traffic suite, thus, it is necessary to analyze its source code to see where changes are necessary, this is discussed in section G. Finally, in section H the modifications made to gem5 (specifically, the Garnet module) to make it compatible with the MCSL traffic suite are presented. With the work presented in this chapter, specific objectives *a* and *d* (see section b)) are fulfilled.

B. *Multi-constraint System Level Suite (MCSL)*

Realistic traffic patterns are very important for reliable NoC performance estimations. As shown in section 7) most of the traffic patterns used for NoC studies are synthetic (e.g., uniform, tornado, hot-spot, etc.). These patterns do not reflect the traffic characteristics of real applications running on the MPSoC, thus, the performance estimations done using these synthetic traffic patterns are of limited use for the system designer. In this section, the Multi-constraint System Level Suite (MCSL) is presented. This suite is a state-of-art tool capable of modeling the traffic patterns of real scientific applications, and it is used in this thesis to test the accuracy of the estimations done by the tool proposed here.

The MCSL [28] is an NoC traffic pattern suite for NoC performance estimation, which can be downloaded from [103]. MCSL includes a set of realistic traffic patterns and covers popular NoC topologies that capture both the communication behaviors and the temporal dependencies of applications running in an NoC. Each traffic pattern in MCSL has two versions, a recorded traffic pattern (RTP) and a statistical traffic pattern (STP). RTP provides detailed communication traces for comprehensive NoC studies, while STP helps to accelerate NoC design exploration at early design stages. MCSL uses Synchronous Dataflow Graphs (SDFGs) to capture both communication and computation requirements of applications. It optimizes application memory requirements, mapping, and scheduling to maximize overall system performance[5], before extracting traffic patterns through cycle-accurate simulations [28].

MCSL uses a Task Communication Graph (TCG) model as the input of the traffic modeling and generation flow to faithfully capture the computation and communication requirements of real applications. A TCG is a directed graph $G_t = (V, E)$, where $V$ is the set of vertices representing computation tasks, and $E$ is the set of edges representing communication links between tasks. A task $v$ has a normalized execution time $t$. A directed edge $e = (v_s, v_d, w)$ has a source task $v_s$, a destination task $v_d$ and the amount of data $w$ that is sent from $v_s$ to $v_d$ [28]. For example, Figure IV-1 shows part of the TCG for an H.264 decoder application.

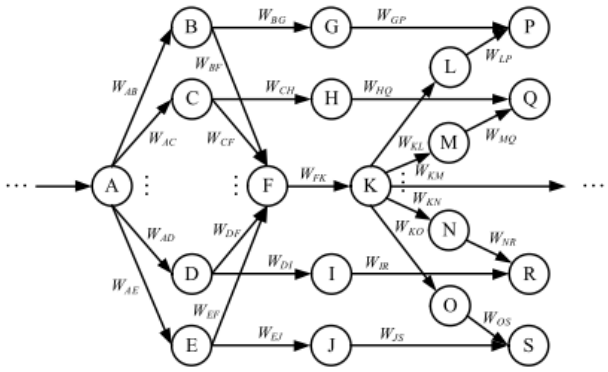


**Figure IV-1.** Example of a Task Communication Graph (TCG). Taken from [28] .

The MSCL tool includes the following applications:

[5] By using optimization techniques based in genetic algorithms [28].

| Application | Description | Number of Tasks | Number of Communication Links |
|---|---|---|---|
| RS-32_28_8_enc | Reed-Solomon code encoder with codeword format RS(32,28,8) | 262 | 348 |
| RS-32_28_8_dec | Reed-Solomon code decoder with codeword format RS(32,28,8) | 182 | 392 |
| H264-720p_dec | H.264 video decoder with a resolution of 720p | 2311 | 3461 |
| H264-1080p_dec | H.264 video decoder with a resolution of 1080p | 5191 | 7781 |
| ROBOT | Newton-Euler dynamic control calculation for the 6-degrees-of-freedom Stanford manipulator | 88 | 131 |
| FPPPP | SPEC95 Fpppp is a chemical program performing multi-electron integral derivatives | 334 | 1145 |
| FFT-1024_complex | Fast Fourier transform with 1024 inputs of complex numbers | 16384 | 25600 |
| SPARSE | Random sparse matrix solver for electronic circuit simulations | 96 | 67 |

**Figure IV-2.** Applications included in the MSCL tool. Taken from [28].

And supports the following NoC topologies:

| Topology | Code | Size (number of processors) |
|---|---|---|
| Mesh | 0 | 2x2, 2x4, 3x3, 4x4, 5x5, 4x8, 6x6, 7x7, 8x8, 9x9, 10x10, 11x11, 8x16, 12x12, 13x13, 14x14, 15x15, 16x16 |
| Torus | 1 | 2x2, 2x4, 3x3, 4x4, 5x5, 4x8, 6x6, 7x7, 8x8, 9x9, 10x10, 11x11, 8x16, 12x12, 13x13, 14x14, 15x15, 16x16 |
| Fat tree | 2 | 4, 8, 16, 32, 64, 128, 256 |

**Figure IV-3.** NoC topologies supported in the MSCL tool. Taken from [28].

*1) Statistical Traffic Patterns (STP)*

STP is a mathematical model of the traffic patterns of an application running on an MPSoC. It can be used for system-level statistical NoC performance estimation. An STP is given by $Ts = \{V_s(p) \mid p \in P\}$, where $V_s(p)$ represents the statistical behaviors of the set of tasks scheduled and executed on the Processing Element (PE) $p$. The task set $V_s = \{(s(v), D_t(v), IS(v), OS(v)) \mid v \in V\}$, where the schedule of task $v$ is given by a unique sequence number $s(v) \geq 0$, and the execution time of the task follows the Gaussian distribution with mean $\mu_t$ and standard deviation $\sigma_t$, e.g., $D_t(v) = (\mu_t(v), \sigma_t(v))$, $with\ \mu_t(v) \geq 0, \sigma_t(v) \geq 0$ [28].

The execution condition of task $v$ is given by its input set of information $IS(v) = \{(v_i(e), n_i(e), m_i(e))\ e \in E_i(v), v_i(e) \in V\}$, where $E_i(v) \subseteq E$ is the set of incoming edges of $v$, the data on every incoming edge $n_i(e)$ must be ready for $v$, and the data are obtained from the corresponding predecessor task $v_i(e)$ and read from the memory space started at $m_i(e)$. The result of the task execution is given by the output set of information $OS(v) = \{(v_o(e),\ p_o(e), m_o(e), D_d(e),\ D_i(e)) \mid e \in$

$E_o(v), v_o(e) \in V, p_o(e) \in P\}$, where $E_o(v) \subseteq E$ is the set of outgoing edges of $v$, and that is to generate some amount of data to each edge $e \in E_o(v)$, the destination is the successor task $v_o(e)$ on PE $p_o(e)$, and the data are written to the memory space started at $m_o(e)$, respectively. Each data is written to and read from the same virtual memory address regardless of the memory architecture, i.e., $m_i(e) = m_o(e)$. The data size generated on an edge can be described by the Gaussian distribution $D_d(e) = (\mu_d(e), \sigma_d(e)), with\ \mu_d(e) \geq 0,\ \sigma_d(e) \geq 0$ [28].

The file format that describes a TCG for a given application is shown in Figure IV-4.

| | | | |
|---|---|---|---|
| ***Header block (5 line)*** [1] | | | |
| trace type | | | |
| topology | number of PBs | number of rows | number of columns |
| number of tasks | number of edges | | |
| number of starting tasks | **list of** starting tasks | | |
| number of finishing tasks | **list of** finishing tasks | | |
| ***Task execution block (number of tasks lines, each of which is as follows)*** [2,3] | | | |
| task id | mapped PB id / coordinate | schedule sequence number | |
| $\mu_t$ | $\sigma_t$ | | |
| ***Task communication block (number of edges lines, each of which is as follows)*** [4] | | | |
| edge id | source task id | destination task id | |
| memory starting address | memory size | | |
| $\mu_d$ | $\sigma_d$ | $\lambda_i$ | |

**Figure IV-4.** File format of an STP TCG application. Taken from [104].

The header block section in Figure IV-4 describes general information about the TCG. For example, the topology of the NoC, the number of PEs (called PBs in the file), the number of tasks (nodes) of the TCG, the number of edges of the TCG, etc. The task execution block describes how the application's tasks must be mapped to the PEs and their execution time information $(\mu_t(v), \sigma_t(v))$. The task communication block describes how the edges connect the tasks (e.g., edge *1* connects task *b* and task *c*), how much data to send between the tasks $(\mu_d(e), \sigma_d(e))$, and the rate at which that data must be transmitted $(\lambda_i)$.

The data structures necessary to implement an STP application are shown in Figure IV-5. The *StatEdge* class models the edges of the TCG, the *StatTask* class models the tasks (nodes) of the TCG, the *StatProc* class models the PEs, and the *StatNOCTraffic* class models the NoC.

```
class StatEdge {
  int                id;                    // the id of the edge
  StatTask*          src_task;              // the source task
  StatTask*          dst_task;              // the destination task

  int                mem_start_addr;        // the starting address of the memory
  int                mem_size;              // the size of the memory
  double             mu_msg_size;           // the mean of the message size
  double             sigma_msg_size;        // the sd of the message size
  double             lambda_pkt_interval;   // the rate parameter, the inverse of
                                            // the average packet generation interval
};

class StatTask {
  int                id;                    // the id of the task
  StatProc*          proc;                  // the PB the task is assigned
  int                schedule;              // the task schedule sequence number
  double             mu_time;               // the mean of the task execution time
  double             sigma_time;            // the sd of the task execution time

  vector<StatEdge*>  incoming_edge_list;    // each entry is an incoming edge
  vector<StatEdge*>  outgoing_edge_list;    // each entry is an outgoing edge
};

class StatProc {
  int                id;                    // the id of the PB
  int                row_index;             // the row index in mesh/torus
  int                col_index;             // the column index in mesh/torus
  vector<StatTask*>  task_list;             // the list of scheduled tasks
};

class StatNOCTraffic {
  int                topology;              // the topology code
  int                num_row;               // the number of rows in mesh/torus
  int                num_col;               // the number of columns in mesh/torus
  vector<StatProc>   proc_list;             // the list of PBs
  vector<StatTask>   task_list;             // the list of tasks
  vector<StatEdge>   edge_list;             // the list of edges

  vector<StatTask*>  starting_task_list;    // the list of starting tasks
  vector<StatTask*>  finishing_task_list;   // the list of finishing tasks
};
```

**Figure IV-5.** Data Structures of an STP application. Taken from [104].

This is all the information needed to use the MCSL suite. How the MCSL suite is used by the cycle-accurate simulator selected for validation tests is explained in section H. Additionally, implementation details of the MCSL suite by the tool presented in this thesis are discussed in section D.

Given that a suitable cycle-accurate simulator should be found to validate the results obtained with the new tool developed in this thesis, the following sections will be dedicated to explore several NoC cycle-accurate simulators, to select one of them as the validation tool in this project.

## C. *BookSim*

BookSim [17] is a flexible and highly modular NoC simulator. It is composed of a hierarchy of modules that implements different functionalities of the network and simulation environment. A hierarchical view of the major simulator modules is shown in Figure IV-6. The top-level modules of the simulator are the *trafficmanager* and *network*. The *trafficmanager* is the wrapper around the network being evaluated and models the source and destination endpoints. It injects packets into the network according to the user-specified configuration, including the traffic pattern, packet size, injection rate, etc. The *trafficmanager* is also responsible for ejecting packets from the destination endpoints, collecting appropriate statistics, and terminating the simulation. The *network* top-level module (also shown in Figure IV-6) comprises a collection of routers and channels, with the topology defining how these modules are interconnected. All communication between neighboring routers occurs through explicit send and receive operations across connecting channels, rather than by updating global variables or data structures. The simulator assumes that credit-based flow control is used for buffer management between adjacent routers and uses a separate,

dedicated channel to communicate credit information. At the lowest level, BookSim simulates the network on the granularity of flits and clock cycles. A packet consists of one or more flits or flow control digits, the smallest unit of channel and buffer allocation [17].

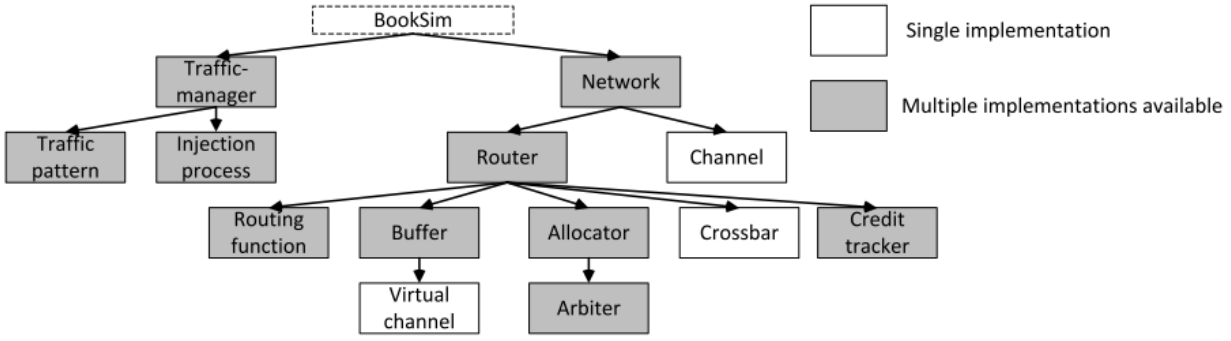


**Figure IV-6.** Module Hierarchy of BookSim. Taken from [17].

*1) Router Microarchitecture Modeling*

The router model used in BookSim is based on an Open-Source NoC Router RTL model [105], [106]. It uses a two-phase protocol to update the router's state. In the evaluation phase (phase 1), which loosely corresponds to combinational logic in a hardware implementation, access to the routers' internal state is permitted. The result of the evaluation phase is a set of state updates, each of which is tagged with the time at which it takes effect. Once the evaluation phase is completed for all pipeline stages of all routers, the simulator enters the update phase (phase 2), in which the routers' internal state is modified to reflect any such updates that are due in the current cycle. This evaluate-update protocol enforces clean clock cycle boundaries and avoids cases in which unintentional serialization is introduced where a parallel implementation was intended [17].

*2) Synthetic Traffic Simulation and Topologies supported*

Earlier versions of BookSim supported a variety of synthetic traffic patterns (uniform, tornado, permutation, etc.). In BookSim2, the flexibility of synthetic traffic simulations has been expanded by allowing arbitrary combinations of synthetic traffic patterns. A combined traffic pattern is created by specifying a list of individual synthetic traffic patterns, each with separately configurable injection rates and packet sizes. At runtime, packets are injected into the network by randomly choosing one of the specified sub-patterns for each packet based on the relative injection rates. This feature allows BookSim2 to simulate an arbitrarily diverse set of synthetic traffic patterns and can help reveal interactions between different patterns. Currently, BookSim supports mesh, torus, and fat-tree topologies [17].

*3) Support, Documentation, and modification possibilities*

BookSim can be downloaded from [107] and a user guide is available [108]. However, detailed source code documentation is unavailable. Thus, modifications of the source code to fit particular needs can be cumbersome.

D. *Noxim*

Noxim [16] allows the simulation of mesh-based NoC architectures featuring several architectural and microarchitectural parameters, including network size, routing algorithms, buffers size, traffic generators,

and so on. The Noxim simulator is developed using SystemC [109], a system description library developed using C++. As depicted in Figure IV-7, from a top-level perspective, the configuration of a specific NoC architecture instantiates two main conceptual elements: a set of nodes and a communication infrastructure. Each node represents a Processing Element (PE) that exchanges data with other nodes employing the communication infrastructure (the NoC). The actual instance of the architecture is entirely determined by the NoC configuration, allowing the customization of several parameters of an NoC [16].

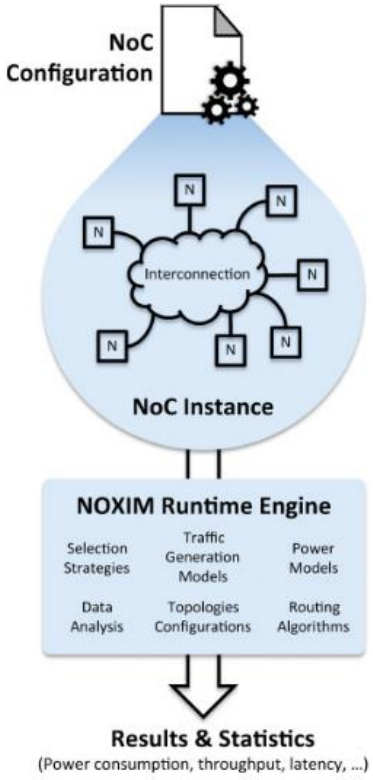


**Figure IV-7.** Simulation flow of Noxim Simulator. Taken from [16].

The NoC instance is then simulated in the Noxim Runtime Engine (RE), which contains the SystemC code for supporting the different NoC architectural elements and models. The Noxim RE allows several topology configurations, buffer and packet sizes, traffic distributions, routing algorithms, and so on. From the user's perspective, an important aspect of the Noxim simulation flow is that the NoC instance is completely determined by the NoC configuration, and there is no need to modify and recompile the source code of Noxim. In other words, the Noxim RE takes the NoC configuration file as input and generates the corresponding SystemC instance before the actual simulation is started [16].

At the end of the simulation, several execution statistics are generated, both in terms of performance (delay, throughput) and energy-related metrics. This information is delivered to the user both in terms of average and per-communication results [16].

Another important feature that further expands the capabilities of Noxim is the inclusion of wireless communication mechanisms in Hub-Hub interconnections, modeled using channels. A channel abstracts the main features of wireless communication on a given frequency, allowing Noxim to model Wireless Networks-on-Chip, which is gaining a lot of interest in recent years in the NoC research community [110], [111], [16].

*1) Router Modeling*

The router entity in Noxim consists of the components required for implementing a routing algorithm. When a header flit of a packet is received from a given input channel, the router applies the routing function and selects an appropriate output channel. If the chosen output channel has not yet been reserved by another input, the router configures the internal crossbar so that all the subsequent flits (until the tail flit) will follow the same path accordingly to the wormhole switching policy [16].

*2) Synthetic Traffic Simulation and Topologies supported*

Noxim provides several commonly used data traffic models including uniform, transpose, bit-reversal, and hot-spot, which abstract typical communication patterns. In addition, it is possible to simulate a real application by mapping its communication graph into customized table-based traffic. Such table-based traffic allows specifying the source and destination pairs along with their communication parameters, including the packet injection rate, its statistical distribution, the amount of traffic to be injected, and the time instants in which such traffic volume must be injected [16]. In Noxim, three different categories of interconnections can be instantiated:

—Tile-Tile: a wired point-to-point connection between two tile nodes.

—Tile-Hub: a wired point-to-point connection between a tile and a radio-hub element.

—Hub-Hub: a connection between two radio-hub elements. these connections can be either wired or wireless.

These three categories provide high flexibility to Noxim, allowing NoC configurations that go far beyond the traditional mesh topology. By mixing different Tile-Hub and Hub-Hub interconnections Wireless NoC topologies can be implemented.

*3) Support, documentation, and modification possibilities*

Noxim can be downloaded from [112] and the source code includes a user's guide. However, detailed source code documentation is unavailable. Thus, modifications of the source code to fit particular needs can be cumbersome.

E. *gem5*

The gem5 simulator [8], is maybe the most popular full system cycle-accurate simulator in the computer architecture community [14], [113]. gem5 provides a highly configurable simulation framework, which includes multiple ISAs; CPU models; and a flexible memory system that includes support for multiple cache coherence protocols and interconnect models. Currently, gem5 supports most commercial ISAs (ARM, ALPHA, MIPS, Power, SPARC, and x86), including booting Linux on three of them (ARM, ALPHA, and x86) [8].

All major simulation components in gem5 are modeled using *SimObjects* classes that share common behaviors for configuration, initialization, statistics, and serialization (checkpointing). *SimObjects* include models of concrete hardware components such as processor cores, caches, interconnect elements (buses and routers), as well as more abstract entities such as a workload (an executable program) and its associated process context for system-call emulation. Every *SimObject* is represented by two classes, one in Python

and one in C++ which derive from *SimObject* base classes present in each language. The Python class definition specifies the *SimObject's* parameters and is used in script-based configuration. The common Python base class provides uniform mechanisms for instantiation, naming, and setting parameter values. The C++ class encompasses the *SimObject's* state and remaining behavior, including the performance-critical simulation model [8].

*1) NoC Modeling*

In gem5, the NoC is modeled inside the Ruby memory subsystem. The Ruby module is in charge of modeling cache hierarchies, coherence protocol implementations, interconnection networks (in our case, the NoC), DMA, and memory controllers [114]. In Figure IV-8 an overview of the Ruby memory subsystem is presented.

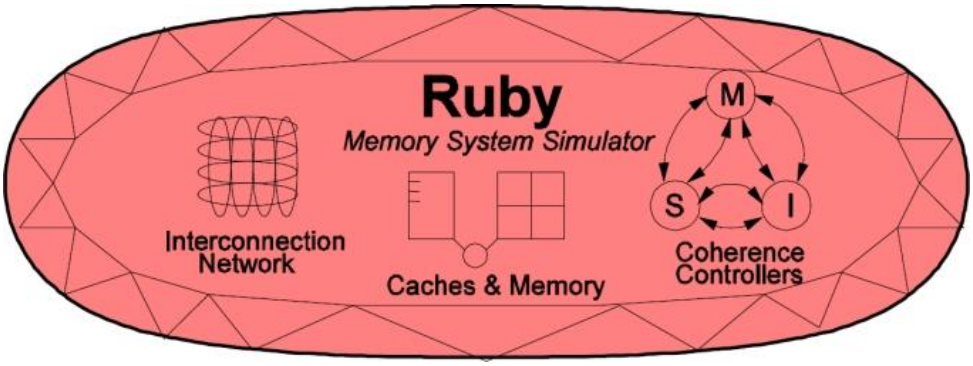


**Figure IV-8.** Overview of the Ruby memory subsystem. Taken from [114].

The interconnection network model in Ruby provides connectivity between the components of the memory system (caches, memory controllers, dma controllers). The interconnection network can be as simple as a bus system ranging to a custom topology NoC.

*2) Garnet*

To model an NoC, Ruby (and therefore gem5) uses the Garnet module [18], [115]. Garnet provides a cycle-accurate micro-architectural implementation of an on-chip network router. The default router is a state-of-the-art 1-cycle pipeline. In Garnet, any heterogeneous topology can be modeled and each router in the topology can be given an independent latency, which overrides the default. The default routing algorithm is a deterministic table-based routing algorithm with shortest paths. Additionally, Garnet can be run in a stand-alone manner and fed with synthetic traffic [116].

In a Garnet topology, the connection between the various elements (caches, memory controllers, and routers) is specified via python files. By default, garnet offers the following topologies: Crossbar, Mesh, and Point-to-Point (Pt2Pt). However, the definition of custom topologies is possible, as stated earlier. These topologies are shown in Figure IV-9, where the *Cache Controllers* (in blue) are the injectors of traffic, and the *Directory Controllers* (in green) are the sinks of traffic. Due to this design decision, to model a node where transmission and reception of traffic are possible (e.g., a CPU) is necessary that a *Cache Controller* and a *Directory Controller* be attached to a router, as shown in the Pt2Pt and Mesh topologies in Figure IV-9.

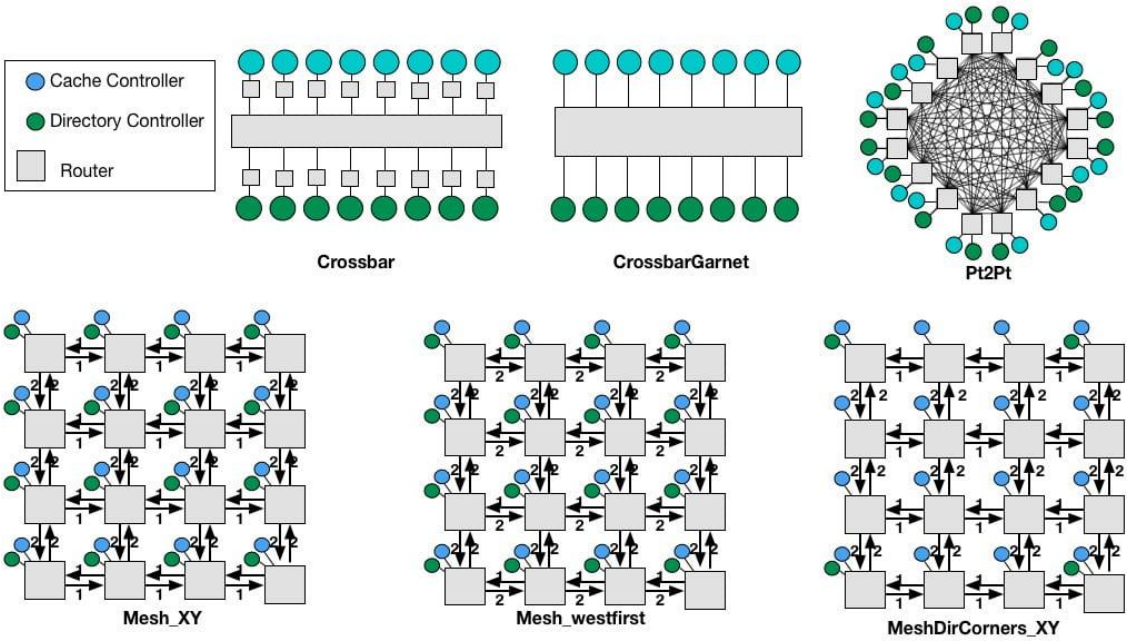


**Figure IV-9.** Topologies available in Garnet by default. Taken from [117].

Also, in Figure IV-9 all external links (between Caches/Directories and routers) are bi-directional. All internal links (between routers) are uni-directional, which allows a per-direction weight on each link (the numbers shown in links between routers) to implement multiple routing protocols [117].

Garnet offers a framework for NoC-only simulations (i.e., simulations that do not involve detailed models of other components of the system, like CPUs or memory controllers) called *Garnet_standalone*. This framework is useful for testing and debugging the NoC, and has several similarities to BookSim and Noxim.

*3) Support, documentation, and modification possibilities*

gem5 can be downloaded from [118] and extensive support is provided in [119], which includes: user's guide and tutorials, design philosophy and source code documentation, a blog, and an active mailing list. Thus, although gem5 is a more complex simulator than BookSim and Noxim, it seems that modifications of its source code are feasible.

F. *Selection of a cycle-accurate simulator*

In this thesis, the main selection criteria for a cycle-accurate simulator is its modification feasibility. This, because the MCSL tool must be implemented on top of it. From the review of cycle-accurate simulators done in the past sections is evident that gem5 is the one that offers better chances of success, and for this reason, it was selected as the validation tool in this thesis. In the following sections, a more detailed study of gem5 and its subsystems are presented, and the modifications and additions to its source code are explained.

## G. *Details of gem5's Garnet Module*

The Garnet module is in charge of controlling all communications at the NoC level in gem5, including the implementation of cache coherence protocols. Thus, to understand how the Garnet module works, it is necessary to have an understanding about cache coherence protocols. A brief introduction to them is given next.

### *1) Brief Introduction to Coherence Protocols*

Coherence protocols are used in multicore systems to ensure the integrity of the data stored in the cache memory of the processors. A coherence protocol is a messaging system that a cache use to notify other cache memories when it needs to read/write data stored in several cache memories. This, to prevent a CPU from processing stale data. Several coherence protocols exist, from simple ones like MI or MSI to more complex ones like MOESI and Token-based [120]. There are two types of coherence protocols: scooping and directory-based. In Scooping protocols, all caches in the system receive the same protocol messages. This protocol is used mainly in bus interconnections, where the number of CPUs in the system is low. Directory-based protocols are used in NoC interconnections. In this type of protocol, the messages are exchanged only between cache memories that share the same data, and also between a cache and a directory (memory) controller when data must be stored or read from main memory. Thus, in this type of protocol, the amount of traffic traversing the NoC is less than in Scooping protocols. The names of the coherence protocols come from the states a cache block (line) can have. For example, for the MI coherence protocol:

- **M** stands for *Modified*. In this state, a cache block has valid data and can be processed safely by the CPU.

- **I** stands for *Invalid*. In this state, a cache block has stale data and it can't be used by other CPUs. In this state, the cache memory that issued the Invalid notification can modify the cache block safely, because no race conditions can occur. After the cache block has been updated with new data, its state changes to *Modified*.

A cache block transitions between its different states according to the coherence protocol state machine. For example, in the MI coherence protocol, a cache block in state M transitions to state I when the cache controller receives an invalidation message for that cache block. Similarly, a cache block in the I state transitions to the M state when the cache controller receives valid data for that cache block.

A more detailed study of coherence protocols can be found in [120].

### *2) Garnet_standalone Coherence Protocol*

The Garnet_standalone coherence protocol is a basic coherence protocol used primary for NoC tests with synthetic traffic in gem5. This protocol is modified in this thesis to integrate the MCLS traffic suite with gem5, thus, a throughout review of it is given next.

The Garnet_standalone coherence protocol uses a 1-level cache hierarchy. The role of the cache is to simply send messages from the CPU to the appropriate directory (based on the address), in the appropriate virtual network (based on the message type). It does not track any state unlike other coherence protocols (MI,

MSI[6], etc.). The directory controller receives the messages from the caches but discards them immediately. The goal of this protocol is to enable simulation/testing of just the interconnection network [121], and in this regard, it enables the study of an NoC using synthetic traffic in a similar way to BookSim and Noxim. Figure IV-10 shows the hardware layout for this coherence protocol. The CPUs connect to their respective cache controller through a Ruby interface; the caches and directory controllers are connected to the NoC. Thus, when a CPU sends a packet the cache controller puts it in the NoC, and after traversing the NoC it arrives at the destination directory, where it is discarded.

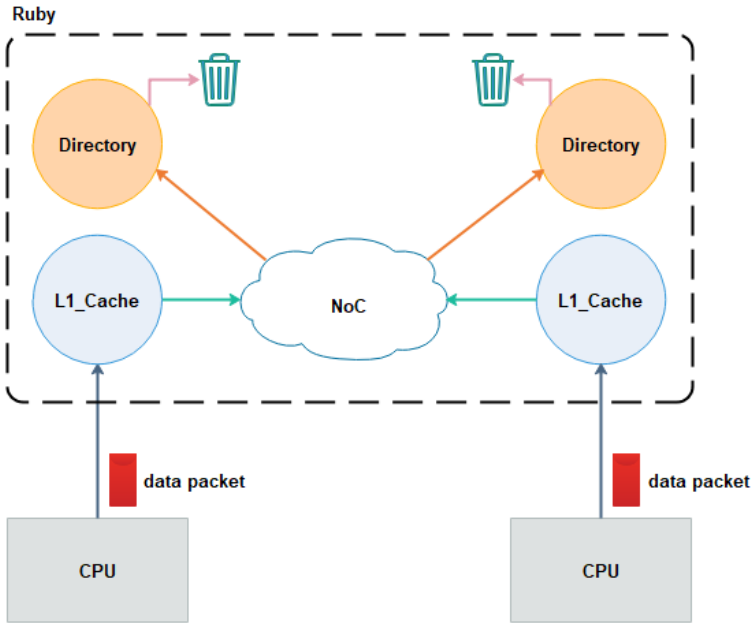


**Figure IV-10.** Hardware layout for Garnet_standalone coherence protocol.

*a) Cache Controller*

In the Garnet_standalone coherence protocol, the goal of the caches is only to act as a source node in the underlying interconnection network. It does not track any cache block states (M, I, S, etc.)[121].

- On a load (LD) request from the CPU:
  It returns a hit to the CPU, mapping the destination address to a directory controller. After that, it issues a message of type MSG in the request virtual network 0 (vnet0).

- On an instruction-fetch (IFETCH) request from the CPU:
  It returns a hit to the CPU and maps the address to a directory controller. After that, it issues a message for it of type MSG in the forward virtual network 1 (vnet1).

- On a store (ST) request from the CPU:
  It returns a hit to the CPU and maps the address to a directory controller. After that, it issues a message for it of type DATA in the response virtual network 2 (vnet2).

[6] The possible states that a cache block can take under this coherence protocol. **M** stands for *Modified*, **S** stands for *Shared* and **I** stands for *Invalid*.

*b) Directory Controller*

The goal of the directory is only to act as a destination node in the underlying interconnection network. It does not track cache block states (M, I, S, etc.). As stated earlier, the directory simply discards the incoming packet [121].

*c) Garnet_standalone Coherence State Machine*

The Garnet_standalone coherence state machine is rather simple. It only contains one state: Invalid (I) (i.e., it marks all cache blocks as Invalid). It reacts to three types of inputs (events) from the CPU: LD, I-FETCH, and ST (already presented in section a)). Figure IV-11 shows the state machine, with its only state and three events. Depending on the event, the state machine commands the Cache to send a dummy packet to the destination directory using one of the three available virtual networks, and a hit response with a dummy cache line (for LD and I-FETCH events) or an ACK (for ST event) to the CPU to end the transaction.

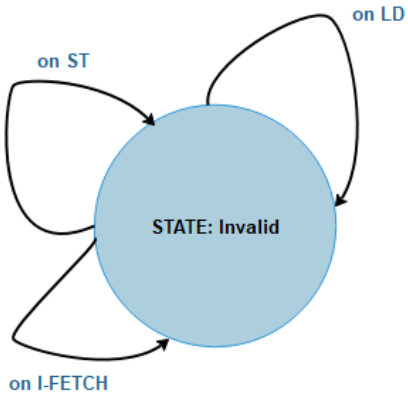


**Figure IV-11.** Garnet_standalone coherence state machine.

The Directories operate under the same state machine, the only difference is that in the event of packet arrival, the packet is discarded immediately.

*d) Specification Language for Implementing Cache Coherence (SLICC)*

To implement or modify a coherence protocol in gem5, it is necessary to use the SLICC language. A review of the SLICC language is given next.

SLICC is a domain-specific language for specifying cache coherence protocols. The SLICC compiler generates C++ code for different controllers (Caches, Directories), which can work in tandem with other parts of Ruby. SLICC is used for specifying the behavior of the state machine. Apart from a protocol specification, SLICC also combines some of the components in the memory model. The SLICC compiler takes, as input, files that specify the controllers involved in the protocol. The files necessary for specifying a protocol include the definitions of the state machines for different controllers (Caches, Directories), and of the network messages that are passed on between these controllers. The files have a syntax similar to that of C++. The compiler, written using PLY (Python Lex-Yacc), parses these files to create an Abstract Syntax Tree (AST). Finally, the compiler outputs the C++ code by traversing the AST, which represents the hierarchy of different structures present within a state machine [122]. Some of these structures are described next using as an example the MI coherence protocol [122]:

- **State Machine:** A machine is described using SLICC's *machine* datatype. Each machine has several different types of members. Machines for cache and directory controllers include cache memory and directory memory data members respectively (see Figure IV-12).

```
machine(MachineType:L1Cache, "MI Example L1 Cache")
  : Sequencer * sequencer,
    CacheMemory * cacheMemory,
    int cache_response_latency = 12,
    int issue_latency = 2 {
      // Add rest of the stuff
    }
```

**Figure IV-12.** machine declaration in SLICC. Taken from [122].

- **Message Buffers:** For the state machine to receive messages from different entities in the system, the machine has several *Message Buffers*. These act as input and output ports for the state machine (see Figure IV-13).

```
MessageBuffer requestFromCache, network="To", virtual_network="2", ordered="true";
MessageBuffer responseFromCache, network="To", virtual_network="4", ordered="true";
```

**Figure IV-13.** Message buffers declaration in SLICC. Taken from [122].

- **States:** the machine includes a declaration of the states that such machine can reach. In coherence protocols, states can be of two types – stable and transient. A cache block is said to be in a stable state if, in the absence of any activity (incoming request for the block from another controller, for example), the cache block would remain in that state forever. Transient states are required for transitioning between stable states. They are needed whenever the transition between two stable states cannot be done in an atomic fashion (see Figure IV-14).

```
state_declaration(State, desc="Cache states") {
   I, AccessPermission:Invalid, desc="Not Present/Invalid";
   II, AccessPermission:Busy, desc="Not Present/Invalid, issued PUT";
   M, AccessPermission:Read_Write, desc="Modified";
   MI, AccessPermission:Busy, desc="Modified, issued PUT";
   MII, AccessPermission:Busy, desc="Modified, issued PUTX, received nack";
   IS, AccessPermission:Busy, desc="Issued request for LOAD/IFETCH";
   IM, AccessPermission:Busy, desc="Issued request for STORE/ATOMIC";
}
```

**Figure IV-14.** State declarations in SLICC. Taken from [122].

- **Events:** The state machine needs to specify the events it can handle to transition from one state to another. SLICC provides the keyword *enumeration* which can be used for specifying the set of possible events (see Figure IV-15).

```
enumeration(Event, desc="Cache events") {
   // From processor
   Load,       desc="Load request from processor";
   Ifetch,     desc="Ifetch request from processor";
   Store,      desc="Store request from processor";
   Data,       desc="Data from network";
   Fwd_GETX,        desc="Forward from network";
   Inv,        desc="Invalidate request from dir";
   Replacement,  desc="Replace a block";
   Writeback_Ack,   desc="Ack from the directory for a writeback";
   Writeback_Nack,   desc="Nack from the directory for a writeback";
}
```

**Figure IV-15.** Events declaration in SLICC. Taken from [122].

For detailed information about how to use SLICC to implement a coherence protocol, the reader is remitted to [122], [123] and a theoretical foundation about coherence protocols can be found in [120].

Regarding the project developed in this thesis, SLICC was used to modify the Garnet_standalone coherence protocol to enable the use of the MCSL suite in Garnet. This is discussed in the next section.

## H. *Garnet_STP*

After having an understanding about the Garnet_standalone coherence protocol works, and how it can be modified using the SLICC language, in this section, the modifications done to the Garnet_standalone coherence protocol are explained.

To implement the TCGs from the MCSL suite in Garnet, modifications to its coherence protocol were necessary. To model the data dependencies of the TCGs, the CPUs must be notified when a packet arrives at the Directory Controllers (and more importantly this packet must be available to the CPUs). To accomplish this, it is necessary to implement a new coherence protocol based on Garnet_standalone. The most important feature of this new coherence protocol, which was called Garnet_STP (STP stands for *Stochastic Traffic Pattern*) is that it provides a communication mechanism between the Directory Controllers and the CPUs, to make available to the CPUs a packet when it arrives at the Directory Controllers. Figure IV-16 shows the hardware configuration for this coherence protocol. As in Garnet_standalone, the CPUs connect to their respective cache controller through a Ruby interface, the cache and directory controllers are connected to the NoC. But unlike Garnet_standalone, in Garnet_STP the Directory Controllers have a communication path to their corresponding CPU, and thus, they can pass an incoming packet to the CPUs for its processing.

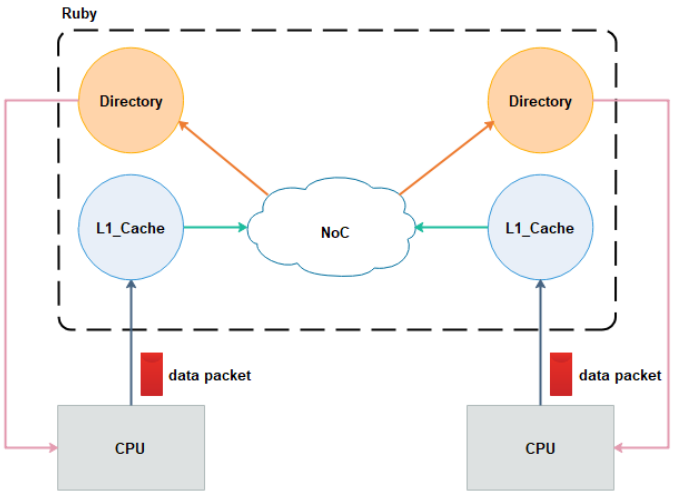


**Figure IV-16.** Hardware Configuration for Garnet_STP coherence protocol.

In Figure IV-17 the definition of the state machine of the cache controller is shown. Here, the message buffers, states (only one, the state I), and events of the state machine are depicted.

```
machine(MachineType:L1Cache, "Garnet_STP L1 Cache")
    : Sequencer * sequencer;
      Cycles issue_latency := 2;

      // NETWORK BUFFERS
      MessageBuffer * requestFromCache, network="To", virtual_network="0",
            vnet_type = "request";
      MessageBuffer * forwardFromCache, network="To", virtual_network="1",
            vnet_type = "forward";
      MessageBuffer * responseFromCache, network="To", virtual_network="2",
            vnet_type = "response";

      MessageBuffer * mandatoryQueue;
{
  // STATES
  state_declaration(State, desc="Cache states", default="L1Cache_State_I") {
    I,  AccessPermission:Invalid, desc="Not Present/Invalid";
  }

  // EVENTS
  enumeration(Event, desc="Cache events") {
    // From processor
    Request,    desc="Request from Garnet_STP";
    Forward,    desc="Forward from Garnet_STP";
    Response,   desc="Response from Garnet_STP";
  }
```

**Figure IV-17.** Description of the state machine for cache controller (using SLICC) in Garnet_STP protocol.

In Figure IV-18 the definition of the state machine for the Directory Controllers is shown. Here, the message buffers, states (only one, the state I), and events of the state machine are depicted. More importantly, a reference to the CPU (an instance of *GarnetSyntheticTraffic* class) is made available to the directory as marked by the brown rectangle. With this new coherence protocol, modeling of the MCSL traffic patterns is possible in gem5.

```
machine(MachineType:Directory, "Garnet_STP_Directory")
    : GarnetSyntheticTraffic * neighboringCPU; // Reference to the CPU

      MessageBuffer * requestToDir, network="From", virtual_network="0",
            vnet_type = "request";
      MessageBuffer * forwardToDir, network="From", virtual_network="1",
            vnet_type = "forward";
      MessageBuffer * responseToDir, network="From", virtual_network="2",
            vnet_type = "response";
{
  // STATES
  state_declaration(State, desc="Directory states", default="Directory_State_I") {
    // Base states
    I, AccessPermission:Invalid, desc="Invalid";
  }

  // Events
  enumeration(Event, desc="Directory events") {
    // processor requests
    Receive_Request, desc="Receive Message";
    Receive_Forward, desc="Receive Message";
    Receive_Response, desc="Receive Message";
  }
```

**Figure IV-18.** Description of the state machine for Directory Controller (using SLICC) in Garnet_STP protocol.

### *1) MCSL suite in Garnet_STP*

In Garnet_standalone a CPU is described by the *GarnetSyntheticTraffic* class. As the name implies this class generates patterns of synthetic traffic (uniform, shuffle, etc.). To use the MCSL suite in Garnet_STP is necessary that the *GarnetSyntheticTraffic* class be able to generate packets according to the Tasks Communication Graphs (TCGs) of the MCSL suite. This is accomplished by the addition of a new attribute to the *GarnetSyntheticTraffic* class, *TasksManager* (see Figure IV-19). *TasksManager* class is in charge of modeling the execution of the tasks assigned to a PE according to the information of the TCGs.

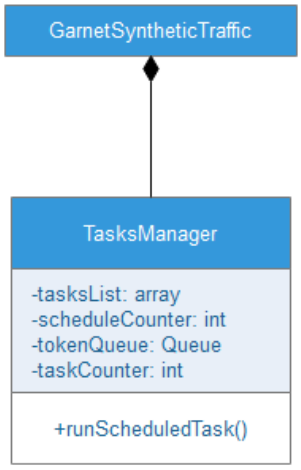


**Figure IV-19.** UML class diagram describing the addition of TasksManager class to GarnetSyntheticTraffic class.

From Figure IV-19, *tasksList* is an array containing the tasks mapped to a PE, *scheduleCounter* keeps track of the next tasks to be executed (modeled); *tokenQueue* is a Queue object (not related with queues in Queueing Theory) that stores the tokens that arrive at a PE; *taskCounter* keeps track of the tasks already executed; and the *runScheduledTask* method models the execution of the tasks. This method schedules the tasks mapped to a given PE under the condition that the task has received all the tokens necessary to execute, and it is the next task to execute (see flow diagram of Figure IV-20).

In Figure IV-20, the process block called *Execute task* models the execution time of a task using the mean execution time and standard deviation information that the TCG supplies for the task. Also, it generates the tokens that must be sent to other tasks according to the information in the TCG. On the other hand, the process block called *process tokens* analyzes the tokens that arrive from other tasks and assign them to the tasks mapped to the PE.

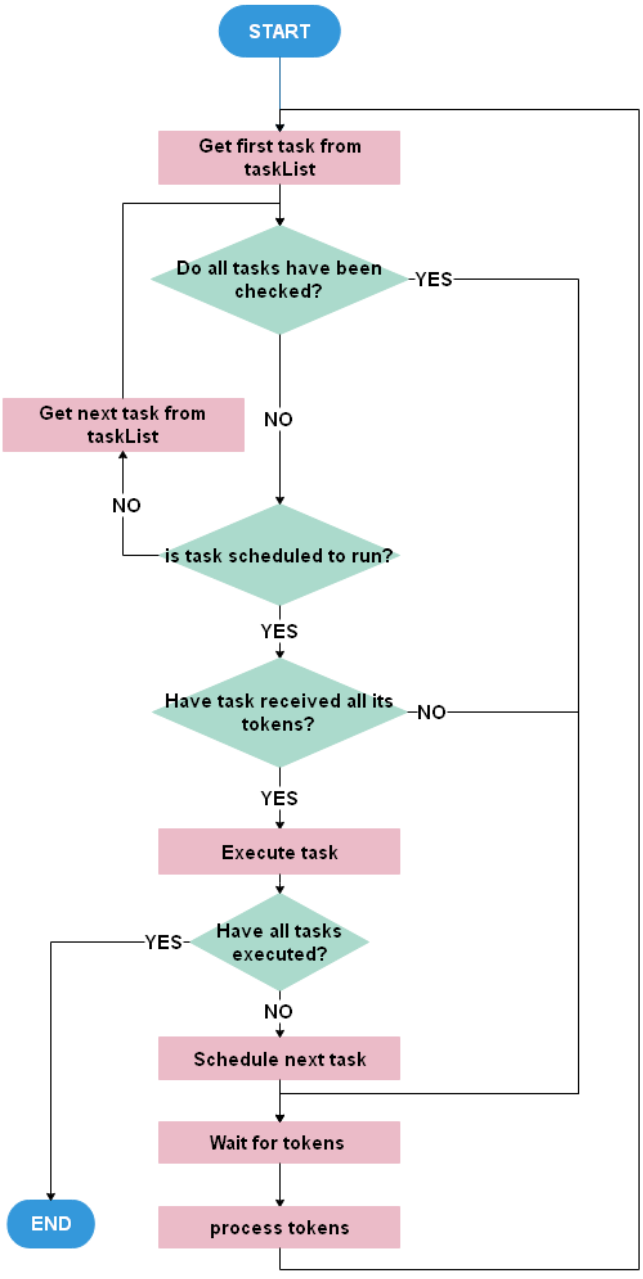


**Figure IV-20.** Flow diagram of runScheduledTasks method.

Thus, in Garnet_STP when a directory receives a packet, it passes it to its neighboring CPU (PE). The CPU processes the packet to extract token information and puts it in the token queue of the tasks manager. The tasks manager schedules and execute the tasks generating tokens according to the TCG of a given application. Tokens for tasks mapped in the same PE are appended to the token queue, and tokens for tasks

mapped in other PEs are sent to the L1_Cache, and from there they're encapsulated in packets and send into the NoC. This is depicted in Figure IV-21.

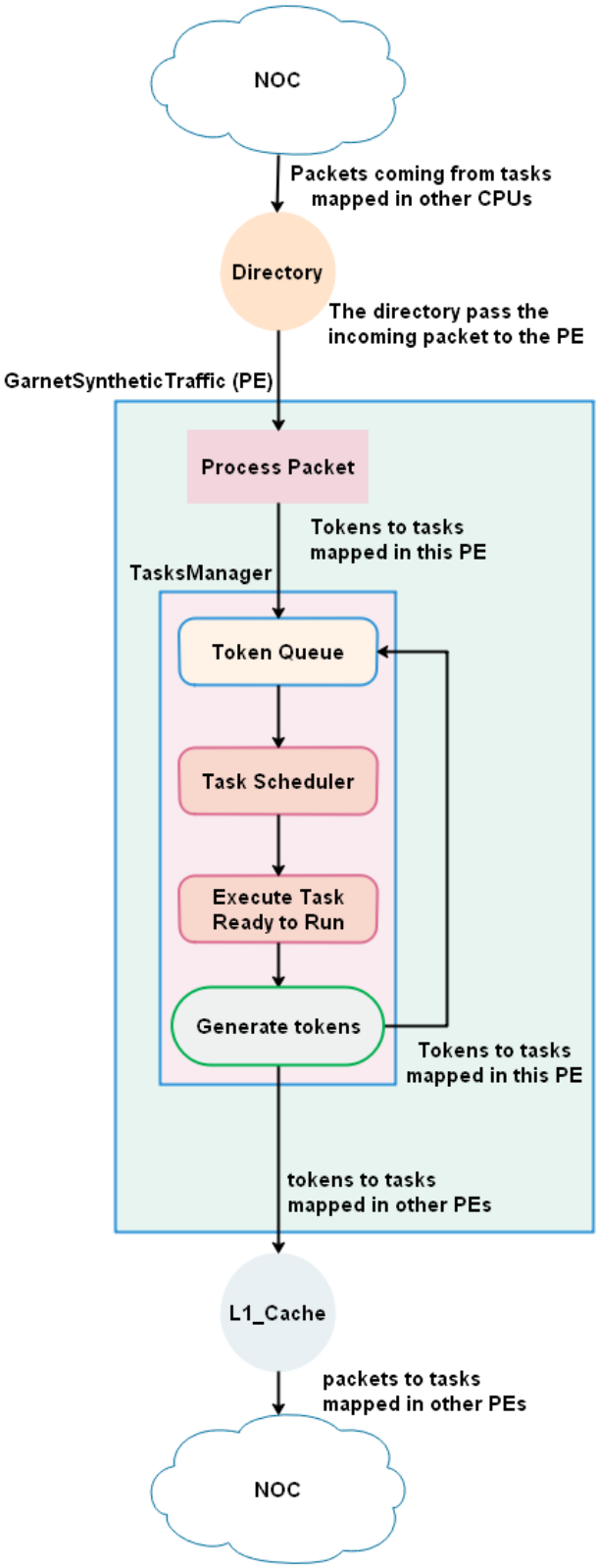


**Figure IV-21.** The interplay of GarnetSyntheticTraffic and TasksManager in Garnet_STP coherence protocol.

## I. *Implementation of Ring, Flattened Butterfly and Fat-tree topologies*

As stated in section 2), gem5 supports Mesh, Cross-bar and Point-to-point topologies by default. However, NoCSimulator can work with Ring, Flattened Butterfly and Fat-tree topologies. Thus, to validate results from NoCSimulator, these topologies must be implemented in gem5. The procedure is described next.

To make it visible to gem5, each new NoC topology must be described in a Python file located in the *gem5/configs/topologies/* folder. In this Python file, a class modeling a given topology is created. This class must implement the *SimpleTopology* interface class of gem5, specifically, the *makeTopology* method. This method describes how the PEs and routers must connect to create the given topology. This is illustrated in the UML class diagram of Figure IV-22.

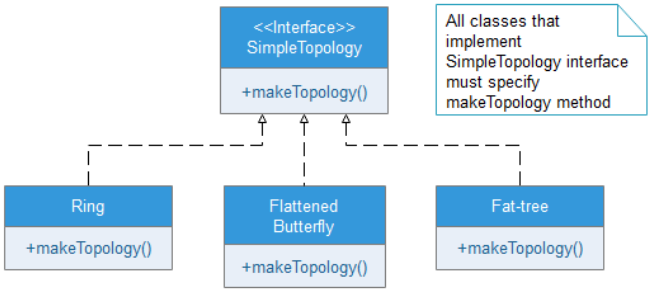


**Figure IV-22.** UML class diagram of SimpleTopology interface.

## J. *CONCLUSION*

In this chapter, the MCSL suite that models traffic patterns of real applications was presented. MCSL is very important in this thesis, because its high-level, yet, accurate representation of the traffics patterns of several real applications enables its use for NoC performance estimation with tools based in formal models like the one presented in this thesis, NoCSimulator. Also in this chapter, the gem5 cycle-accurate simulator was selected as the validation tool for NoCSimulator. This, after a review of other cycle-accurate simulators like Booksim and Noxim. The criteria for this selection was the ease of modification of the source code of these cycle-accurate simulators, because the MCSL suite must be integrated with the chosen simulator for the validations tests of NoCSimulator. Thus, gem5 was selected because it has better software documentation, and an active community of users, giving better chances for a successful integration with the MCSL traffic suite. Finally, the modifications to gem5's source code (specifically, the Garnet module) to integrate the MCSL traffic suite were explained, including modifications to the Garnet standalone coherence protocol and the implementation of the Ring, Flattened Butterfly and Fat-tree NoC topologies, not included by default in gem5. With the work presented in this chapter specific objectives *a* and *d* (see section b)) are met.

# V. NOCSIMULATOR OVERVIEW

**CONTENTS**



## A. *Introduction*

In this chapter, NoCSimulator, the NoC performance estimation tool developed in this thesis is presented. NoCSimulator sets apart from the other Queuing Theory based tools discussed in section 7), in that, to the best of our knowledge, it is the only one capable of processing traffic patterns of real applications. This is achieved thanks to the use of the MCSL NoC traffic suite [28] presented in section B. Thus, more accurate insights about NoC performance can be obtained from the early design stages of a system. Additionally, the tools reported in section 7), only support one topology (most commonly Mesh), while NoCSimulator supports by default four common NoC topologies: Mesh, Ring, Flattened Butterfly and Fat-tree. This, widens the design space available to the system architect, giving better opportunities to find a NoC configuration that meets the design specification of a system.

In the remainder of this chapter NoCSimulator is discussed thoroughly. Section B gives a general overview of NoCSimulator. Next, section C covers implementation details of NoCSimulator, including topics such as how NoC topologies are constructed, how Processing Elements (PEs) and Routers are modeled, and how Queueing Theory is used in NoCSimulator. Later, section D explains how the MCSL NoC traffic suite (see section B) is used in NoCSimulator. Finally, section E explains the runtime behavior of NoCSimulator. This chapter is intended to fulfill specific objective *c* (see section b)).

B. *NoCSimulator Description*

NoCSimulator was developed using the Python programming language [124] (version 3.7) and its discrete-event-simulation framework Simpy (version 3.0.1) [125]. NoCSimulator models the routers of an NoC as M/M/1/FIFO/c queues (see section 5)) that are interconnected to form an NoC topology. Figure V-1 is a representation of a network of queues in NoCSimulator for a Mesh NoC.

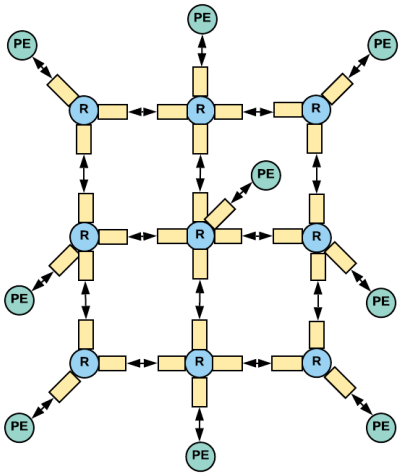


**Figure V-1.** Network of queues representing a Mesh NoC.

NoCSimulator allows the modeling of the main elements that make up an NoC, such as the Processing Elements (PEs) that are the source and receivers of NoC traffic, and the routers that are responsible for moving traffic from its origin to its destination. This simulation tool is flexible enough to implement several topologies like Mesh, Ring, Flattened Butterfly, and Fat-tree.

In addition, several performance metrics can be estimated using NoCSimulator, e.g., the performance of the routers (average number of flits in buffers, utilization factor of routers, latency of flits in routers, and flit throughput). Besides, overall NoC performance metrics like average latency and throughput can also be estimated. Additionally, NoCSimulator can handle synthetic traffic and traffic based on real applications (the last one, using the MCSL suite discussed in section B). The latter differentiates NoCSimulator from others tools currently used, like BookSim [17] and Noxim [16] which, by default, can only handle synthetic traffic patterns. The possibility to work with traffic patterns based on real applications enables the user to do design space exploration, as will be shown in Chapter 7.

C. *NoCSimulator implementation details*

In this section, an in-depth description of how NoCSimulator was developed is presented. Topics such as how an NoC is modeled and how performance estimation is calculated are discussed. But first, a brief explication of why it was necessary to use event-driven simulation is given next.

*1) Why use Simpy?*

As stated in section C, discrete-event simulation is necessary when studying complex networks of queues due to the lack of closed analytical solutions. Simpy [125] provides mechanisms to model the passing of time and the access to shared resources by different actors of a system. For example, planes (actors) arriving

at a busy airport (shared resource of limited capacity) or packet deliveries (actors) done by a mailman (shared resource). Specifically, in NoCSimulator, Simpy is used to model the passing of time and packet contention at the routers of an NoC. Thus, Simpy is used to provide the input data to the Queueing Theory formulas presented in 6) to estimate the performance of an NoC.

*2) Topology modeling*

In NoCSimulator, the general entity that models an NoC is called a *Topology*. In turn, a Topology is composed of Routers and PEs. Figure V-2 presents a UML class diagram describing this relationship. From this figure, a topology can have one or more routers associated with it (as indicated by the multiplicity value of "1…*"); the same applies to the PEs. Additionally, any given router or PE can be associated with only one topology (as indicated by the multiplicity value of "1").

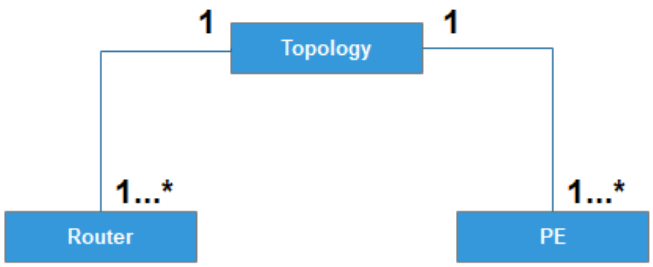


**Figure V-2.** Class Diagram of a Topology in NoCSimulator.

Topology is an interface, and classes that model specific topologies must describe how they are constructed implementing the *setupTopology* method, as shown in the UML class diagram of Figure V-3. Thus, the Topology interface contains common data to all topologies e.g., it contains a reference to the Simpy simulation environment, the PEs, and routers. Also, the Topology interface models common behavior to all topologies like the start of a simulation run (*start* method), uniform traffic initialization (*uniformTraffic* method), and calculation and presentation of results (*calculateStatistics* and *printStatistics* methods).

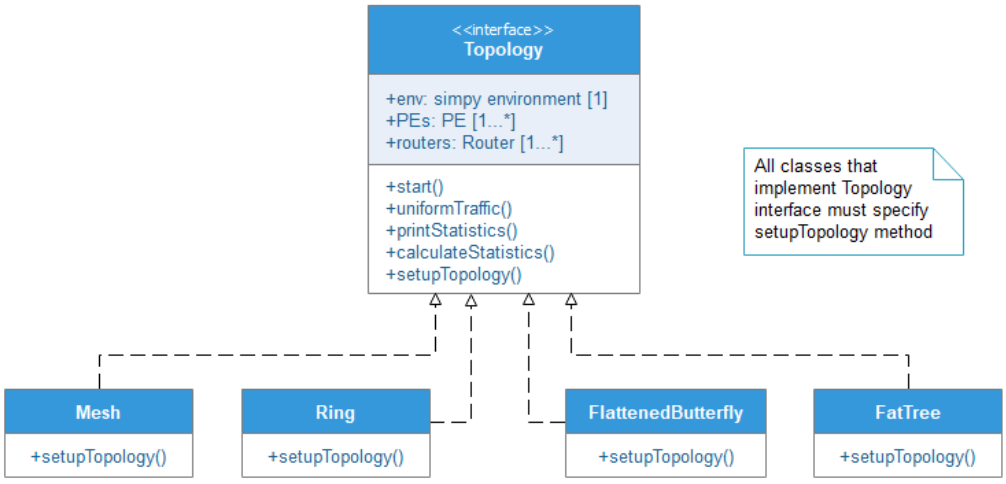


**Figure V-3.** UML class diagram of Topology interface.

*3) uniformTraffic method (synthetic traffic in NoCSimulator)*

In this method, uniform traffic for the NoC is implemented. In uniform traffic, each PE in the NoC has the same probability of sending a packet to every other PE in the NoC. This traffic pattern is a common synthetic pattern found in many NoC simulation tools like BookSim [17], Noxim [16], and Garnet [18]. It was implemented in NoCSimulator to compare results with BookSim (as shown in Chapter 6). The probability of sending a packet to a PE is calculated as shown in Equation III-1:

$$P_s = \frac{1}{N - 1} \qquad \text{Equation V-1}$$

Where $P_s$ is the probability of a PE of sending a packet to any other PE in the NoC, and $N$ is the number of PEs in the NoC. $N - 1$ is used in the formula because a PE doesn't send packets to itself.

*a) setupTopology method*

When modeling a specific topology e.g., a Mesh, the *setupTopology* method must be defined. In this method, the topology class must describe how the PEs and routers must be connected to realize the topology. This design choice permits the generalization of common features to all topologies and the description of unique features of a given topology.

*b) calculateStatistics method*

In this method, the average latency and throughput of the NoC are calculated. It takes as input data the latency and throughput calculated at each PE of the NoC, averaging these values to calculate the overall performance of the NoC. These formulas are shown in Equation V-2 and Equation V-3.

$$L_{avg} = \frac{\sum_{i=1}^{N} L_{PE_i}}{N} \qquad \text{Equation V-2}$$

$$T_{avg} = \frac{\sum_{i=1}^{N} T_{PE_i}}{N} \qquad \text{Equation V-3}$$

Where $L_{avg}$ and $T_{avg}$ are the average latency and throughput of the NoC, $L_{PE_i}$ and $T_{PE_i}$ are the latency and throughput of $PE_i$, and $N$ is the number of PEs in the NoC.

*4) Processing Element (PE) modeling*

In NoCSimulator, a PE is modeled as a source and sink of packets. Each PE is composed of a Source and a Sink object, as shown in the UML class diagram of Figure V-4.

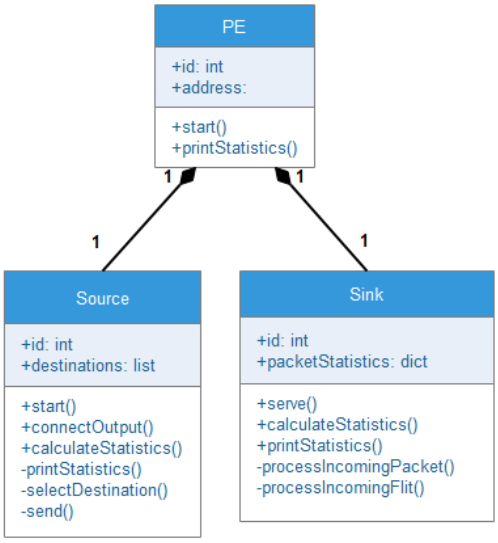


**Figure V-4.** UML class diagram for PE class.

A given PE can have only one source and one sink. Additionally, a source and a sink must be associated with only one PE. These restrictions are indicated with the multiplicity values of "1" in Figure V-4. The Source class is defined to implement the transmission of data packets. For this task, it specifies behaviors such as initialization (*start* method), connect to a router (*connectOutput* method), select the destination of a packet (*selectDestination* method), send a packet (*send* method), and performance estimation (*calculateStatistics* method). The Sink class is defined to implement the reception of data packets. For this task, it implements behaviors such as initialization (*start* method), servicing an incoming packet (*serve*, *processIncomingPacket* and *processIncomingFlit* methods), and performance estimation *calculateStatistics* method).

*a) Source object's selectDestination method*

This method is called when the simulation is running. In this method, one of the PEs in the NoC is selected randomly to be the destination of the next data packet. The next PE to receive a packet is chosen using the exponential distribution (this distribution calculates time between packets, see section 1)) and the routing probabilities calculated with the *uniformTraffic* method (section 3)). The PE with the minimum time between packets is selected as the destination of the next packet. This is shown in Equation V-4.

$$\text{next\_PE} = \min(\mathrm{F_{exp}}(\mathrm{P_{si}})) \quad i = 1, 2, \dots, N \qquad \text{Equation V-4}$$

Where $next\_PE$ is the PE to send the next packet, $F_{exp}$ is the exponential distribution, $P_{si}$ is the probability to send a packet to $PE_i$ (calculated in Equation III-1) and $N$ is the number of PEs in the NoC.

*b) Source object's calculateStatistics method*

This method is called at the end of a simulation. In this method, the packet generation rate (throughput) of the PEs is calculated. Generally, under no saturation conditions of NoC traffic, this throughput equals the nominal throughput selected for the simulation ($\lambda$), but under saturation conditions, the throughput at the

PEs is lower than nominal due to packet contention in the NoC. This parameter is calculated by counting the number of packets that were generated during the simulation, dividing this value by the simulation time. This is shown in Equation V-5, where $\lambda_{real}$ is the actual packet generation rate of the PEs:

$$\lambda_{\text{real}} = \frac{\text{number\_of\_packets}}{\text{simulation\_time}} \qquad \text{Equation V-5}$$

*c) Sink object's serve method*

This method is called while the simulation is running. In this method, partial statistics are kept about the packets that arrive at a given PE. This method uses two helper methods (private): *processIncomingPacket* and *processIncomingFlit.* Some of the statistics that are measured are packet latency (i.e., the time necessary to reach a destination), the number of hops (i.e., how many routers were traversed to reach a destination), and the route taken (i.e., which routers were traversed to reach a destination). These measurements are gathered during a simulation run and are used by the Sink object's *calculateStatistics* method at the end of a simulation.

*d) Sink object's calculateStatistics method*

This method is called at the end of a simulation. In this method, the average latency and hop count of the packets that arrived at a given PE are calculated. This is shown in Equation V-6 and Equation V-7.

$$\mathrm{L_{PE}} = \frac{\sum_{i=1}^{k} \mathrm{L_{P_i}}}{\mathrm{k}} \qquad \text{Equation V-6}$$

$$\mathrm{H_{avg}} = \frac{\sum_{i=1}^{k} \mathrm{H_{P_i}}}{\mathrm{k}} \qquad \text{Equation V-7}$$

For Equation V-6, $L_{PE}$ is the average packet latency for a PE, $L_{P_i}$ is the latency of packet $i$ and $k$ is the number of packets that arrived at the PE. For Equation V-7, $H_{avg}$ is the average hop count of the packets for that arrived at the PE, $H_{P_i}$ is the hop count of packet $i$ and $k$ in the number of packets that arrived at the PE.

*5) Router modeling*

A router is responsible for moving packets from their source to their destination. For each topology to be modeled, the routers must implement the Router interface. This interface contains all common data and behavior to all routers (see UML class diagram of Figure V-5). e.g., initialization (*start* method), connection setup (*addBuffer* and *connect* methods), and packet processing (*serve* method). Additionally, the Router interface enables the description of behavior that is unique to a router used in a specific topology: the routing

protocol. For example, the routing protocol used in a Mesh topology (XY routing) differs from the one used in a Ring topology (shortest path routing). The routers describe their routing protocol when they specify the *calculateRoute* method. As in the case of the Topology interface, this design decision enables the generalization of common behavior to all routers and the specification of unique behavior to a router used in a given topology (the routing protocol).

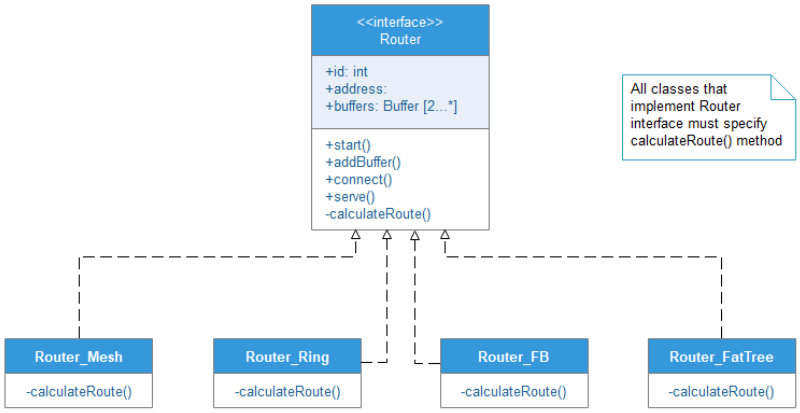


**Figure V-5.** UML class diagram for Router interface.

*a) Router object's serve method*

This method is called during a simulation. In this method the router processes the incoming packets, keeping records of variables such as waiting time at the buffers ($W_q$), service time ($T$), and packet latency ($W$). Also, in this method, the next hop to the destination of the packet is determined using the routing protocol of the given topology.

- **Service time**

The service time at the routers is determined randomly using the exponential distribution (see Chapter 3, section 3.2.1), using as parameter the processing rate ($\mu$) assigned to the router as shown in Equation V-8.

$$T = F_{exp}(\mu) \qquad \text{Equation V-8}$$

Where T is the service time, $F_{exp}$ is the exponential distribution and $\mu$ is the processing rate of the router.

- **Waiting time in the buffers**

Waiting time at the buffers is calculated as the elapsed time from the moment at which the packet arrives at the buffer until the packet is processed by the router, as shown in Equation V-9.

$$W_q = t_s - t_a \qquad \text{Equation V-9}$$

Where $W_q$ is the waiting time in the buffer, $t_s$ is the time at which the router starts to process the packet, and $t_a$ is the time of arrival of the packet at the buffer.

- **Packet latency**

The latency of packets is calculated as the service time plus the waiting time in the buffers along the path of the packet, from source to destination, as shown in Equation V-10.

$$L = \sum_{i=1}^{H} (T_i + W_{qi}) \qquad \text{Equation V-10}$$

Where $L$ is the latency of the packet, $T_i$ is the service time at router $i$ in the path of the packet, $W_{qi}$ is the waiting time at the buffer of router $i$, and H is the number of routers (hops) in the path of the packet.

*b) Router object's calculateStatistics method*

This method is called at the end of the simulation. In this method, statistics about the performance of the router are calculated. These statistics include: average service time ($E[T]$), the average number of packets in the router $E[N]$, and the utilization factor of the router ($\rho$).

- **Average service time of a router**

The average service time of a router is a measure of the time a given packet spends at a router and is calculated using Equation V-11.

$$E[T] = \sum_{j=1}^{b} \frac{\sum_{i=1}^{p_j} T_i}{p_j} \qquad \text{Equation V-11}$$

Where $E[T]$ is the average (or expected) time of a packet in the router, $T_i$ is the service time of packet $i$ in buffer $j$, and $p_j$ is the number of packets that were received in buffer $j$.

- **Average waiting time in a buffer**

The average waiting time in a buffer is a measure of how long a packet must wait before being serviced by the router and is calculated using Equation V-12.

$$E[W_q] = \frac{\sum_{i=1}^{p} W_{qi}}{p} \qquad \text{Equation V-12}$$

Where $E[W_q]$ is the average (or expected) waiting time of a packet at a buffer, $W_{qi}$ is the waiting time of packet *i*, and *p* is the number of packets that were received at the buffer.

- **Average number of packets in a router**

The average number of packets in a router is calculated using Little's Law (chapter 3, section 3.2.6) as shown in Equation V-13.

$$E[N] = \lambda * E[T] \qquad \text{Equation V-13}$$

Where $E[N]$ is the average (or expected) number of packets in a router, $\lambda$ is the rate at which packets arrive at a router, and $E[T]$ is the average service time.

- **Utilization factor of a router**

The utilization factor of a router measures how busy a given router is under a given packet rate condition. It is calculated using a formula derived from Little's law (chapter 3, section 3.2.6) as shown in Equation V-14.

$$\rho = \frac{\lambda}{\mu} \qquad \text{Equation V-14}$$

Where $\rho$ is the utilization factor of a router, $\lambda$ is the packet arrival rate, and $\mu$ is the service rate of the router.

- **Average number of packets in a buffer**

The average number of packets in a buffer is calculated using one of the forms of Little's Law (chapter 3, section 3.2.6) as shown in Equation V-15.

$$\mathrm{E}[\mathrm{N_q}] = \lambda * \mathrm{E}[\mathrm{W_q}] \qquad \text{Equation V-15}$$

Where $E[N_q]$ is the average (or expected) number of packets in a buffer, $\lambda$ is the packet arrival rate, and $E[W_q]$ is the average waiting time in the buffer.

*c) Router object's calculateRoute method*

This method is called during a simulation. Depending on the topology of the NoC, the routers must run a routing protocol suitable to that topology. This method is defined to implement details of the routing protocols used in NoCSimulator (Mesh, Ring, Flattened Butterfly, and Fat Tree).

- **Mesh topology**

For Mesh topologies the XY routing protocol was implemented. In this routing protocol packets, are first routed in the x direction (horizontal) and then in the y direction (vertical). This is shown in Figure V-6, where a packet from node $(0,0)$ must first travel to node $(2,0)$ in the horizontal direction, and then the packet must travel in the vertical direction to reach its destination at node $(2,2)$.

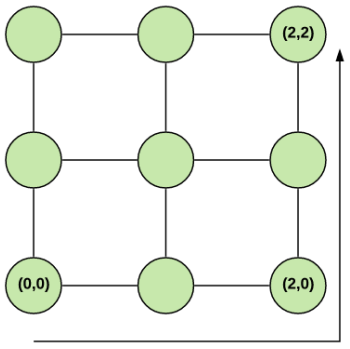


**Figure V-6.** XY routing in Mesh topology.

Figure V-7 shows the flow diagram of the XY routing protocol as it was implemented in the *Router_Mesh* class. Here, *x address* and *y address* correspond to the *(x,y)* coordinates assigned to the router. Additionally, *packet x address* and *packet y address* correspond to the *(x, y)* coordinates assigned to the destination PE.

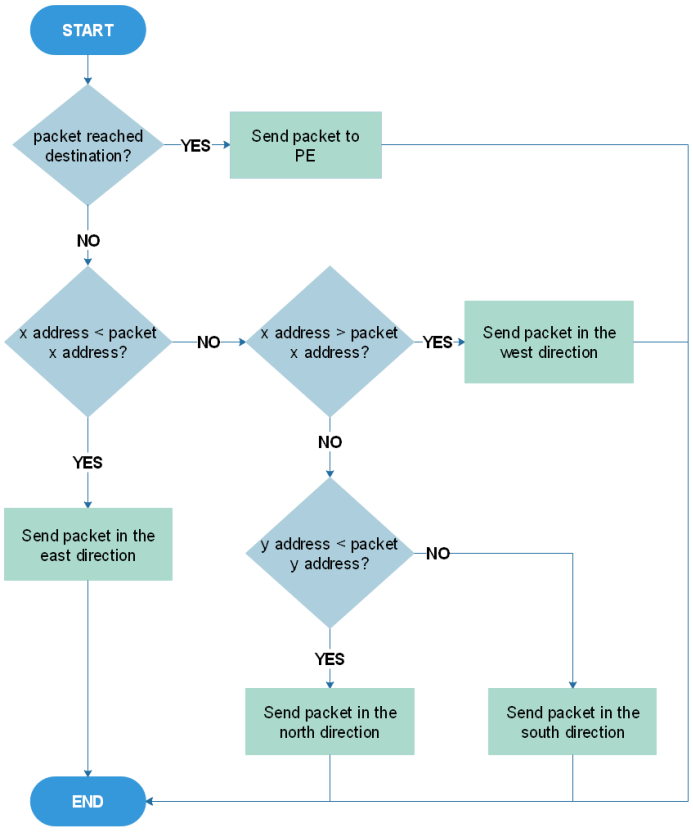


**Figure V-7.** Flow diagram of XY routing algorithm.

- **Ring topology, Flattened Butterfly and Fat-tree topologies**

For Ring, Flattened Butterfly, and Fat-tree topologies, the shortest path between each PE is identified using Dijkstra's shortest-path algorithm [126]. During topology construction, a one-to-one graph representation of the topology is constructed alongside. Dijkstra's algorithm is run over this graph, and the shortest path between each node is stored in a routing table to be used by the routers when directing traffic. A brief description of Dijkstra's shortest-path algorithm is given next.

- **Dijkstra's shortest-path algorithm**

Shortest-path identification is a common application in graph theory, an example is a navigation system where the graph vertices represent cities in a map and the edges correspond to roads that communicate those cities. Edges have an assigned weight that represents the cost of traversing that edge (it can represent time, distance, fuel consumption, wear level, etc.). The objective is to find the path with the lowest-cost (shortest-path) from a source vertex to a destination vertex. Dijkstra's shortest-path algorithm traverses the graph from a source vertex keeping track of the shortest path (lowest sum of edge weights) between the source vertex and the vertices it finds in the graph. When a vertex is not reachable from the source vertex by convention the path has an infinite cost. Also, when multiple shortest paths between a source vertex and a

destination vertex exist (i.e., paths with the same cost), the algorithm only selects one of them. A flow diagram representation of Dijkstra's algorithm is shown in Figure V-8.

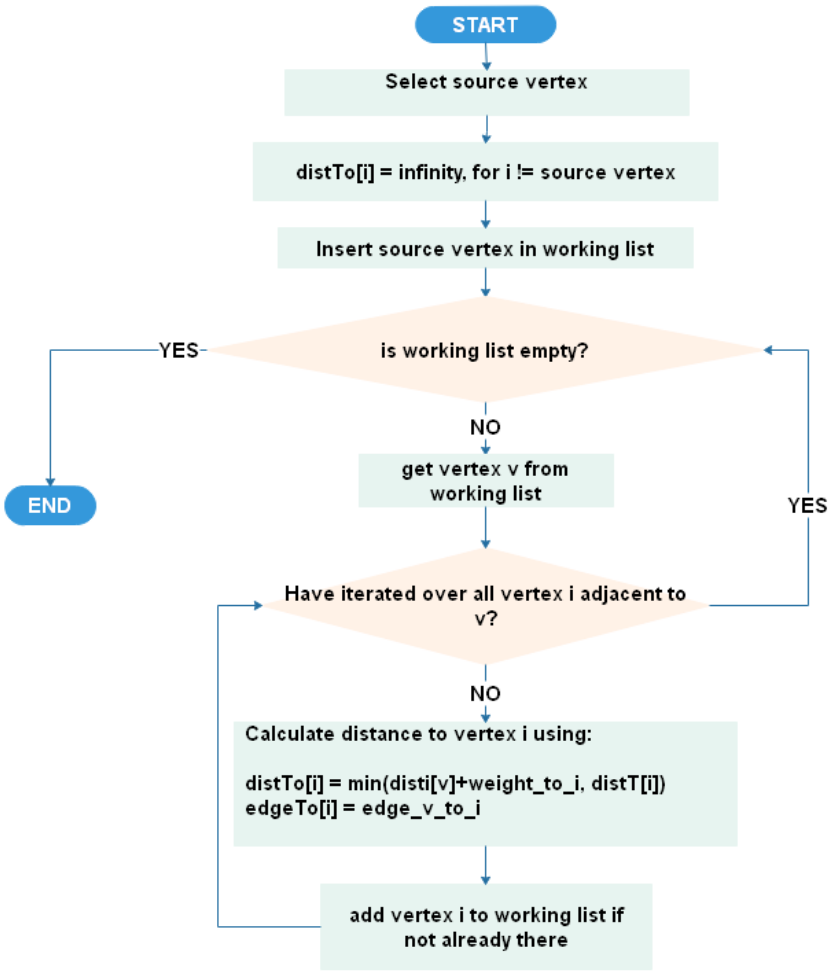


**Figure V-8.** Flow diagram of Dijkstra's shortest-path algorithm.

Where *working list* is an array that stores the vertices not processed yet by the algorithm. *distTo* is an array that stores the cost of the path to vertex *i*, and it is calculated by selecting the minimum value between the sum of the cost to reach vertex *v* and the weight of the edge from *v* to *i* ($distTo[v] + weight\_to\_i$), and the current cost to reach vertex *i* ($distTo[i]$). Thus,$distTo[i] = \min(distTo[v] + weight\_to\_i, distTo[i])$ . *edgeTo[i]* is an array that keeps track of the edges that must be traversed to reach vertex *i*. A more in-depth treatment of Dijkstra's shortest-path algorithm and graph theory can be found in [126].

For the Ring, Flattened Butterfly, and Fat-tree graph representations, the weight assigned to their edges was 1 (one), since the objective of the routing protocol is to direct traffic minimizing the number of hops (routers) that a packet must traverse to reach its destination. this is shown in Figure V-9 for the Ring topology. This figure shows the NoC topology (on the left) and its graph representation (on the right). weight 1 (one) is assigned to all edges in the graph and Dijkstra's algorithm is run over the graph. The information about the shortest paths in the graph is made available to the routers of the NoC, thus, they can direct traffic between the PEs correctly. For example, in Figure V-9 R4 needs to send a packet to R2, after consulting the routing table, R4 finds out that the shortest path to reach R2 is through R3.

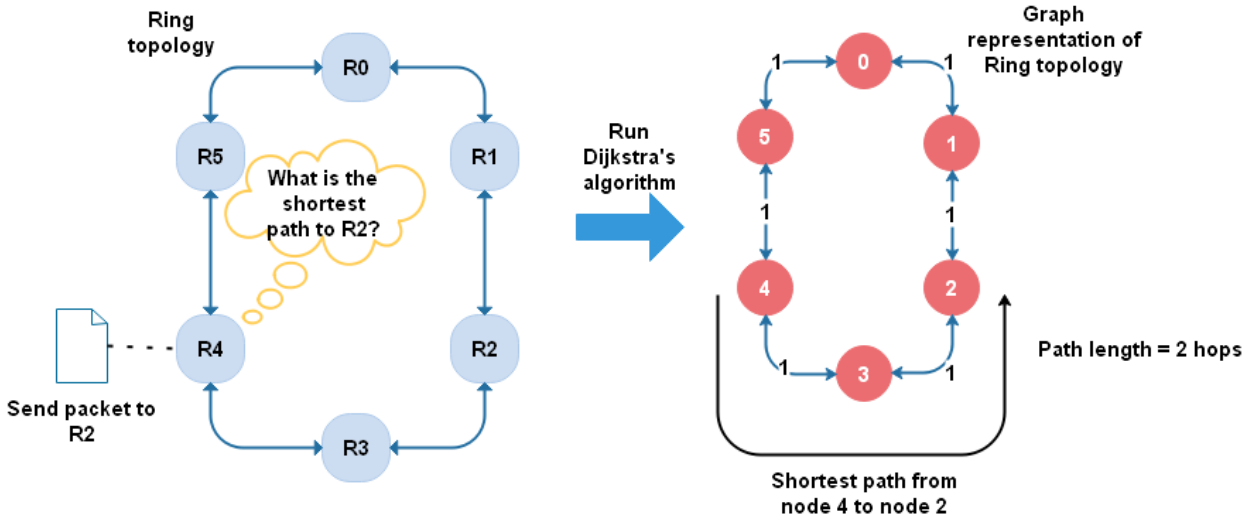


**Figure V-9.** Ring topology and its graph representation.

D. *MCSL suite in NoCSimulator*

NoCSimulator uses the MCSL tool discussed in section B to model traffic patterns of scientific applications. To do so, new classes were added to NoCSimulator and are discussed next.

*1) Mesh_STP, Ring_STP, FlattenedButterfly_STP and FatTree_STP Classes*

To use the Tasks Communication Graphs (TCGs) of the MCSL tool in NoCSimulator, new topology classes must be defined. These classes are *Mesh_STP*, *Ring_STP*, *FlattenedButterFly_STP*, and *FatTree_STP* classes, where STP stands for *Statistical Traffic Patterns*. These classes are subclasses of the Mesh, Ring, FlattenedButterfly, and FatTree classes discussed in section 2) and are shown in the UML class diagram of Figure V-10.

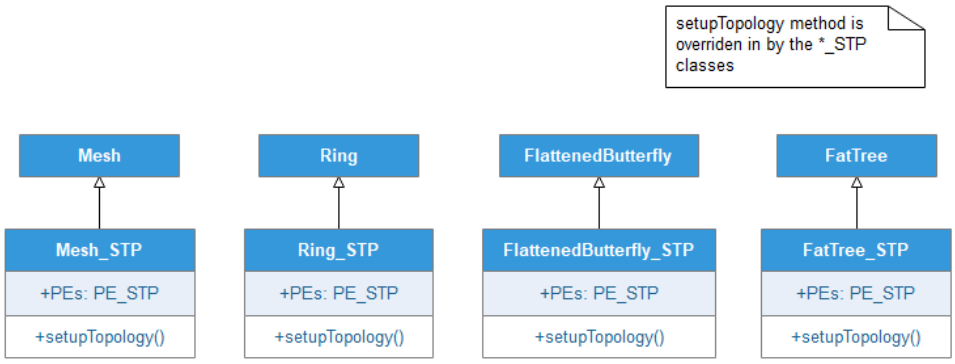


**Figure V-10.** STP classes UML class diagram.

These classes differ from their parent classes in that they use a new kind of PE called PE_STP, and override the *setupTopology* method (section a)). The new functionality added to the *setupTopology* method is that during topology construction tasks are mapped to the PEs following the TCG of a given application.

*2) PE_STP class*

The PE_STP class is a subclass of the PE class discussed in section 4) and differs from its superclass in that it uses new Source_STP, Sink_STP, and TaskManager objects as shown in the UML class diagram of Figure V-11.

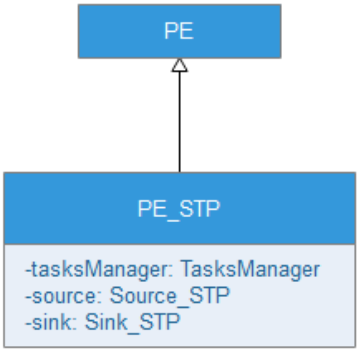


**Figure V-11.** PE_STP class diagram.

*a) tasksManager Class*

This class is responsible for modeling the tasks that are mapped to a PE according to the Task Communication Graphs (TCGs) of the MCSL suite (see section B). This class uses the information contained in the TCGs (for example, the mean execution time of the tasks, mean number of packets generated, schedule number, etc.) to model the execution of the tasks. A UML class diagram of this class is shown in Figure V-12.

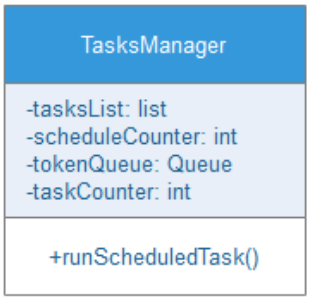


**Figure V-12.** UML class diagram of TasksManager class.

In Figure V-12, the *tasksList* attribute contains the tasks mapped to a given PE, *scheduleCounter* keeps track of the next tasks to be executed (modeled), *tokenQueue* is a Queue object (not related to queues in Queueing Theory) to store the tokens received, and *taskCounter* keeps track of the number of tasks executed (modeled). The *runScheduledTask* method is discussed next.

- ***runScheduledTask* method**

This method schedules the tasks mapped to a given PE under the condition that the task has received all the tokens necessary to execute, and it is the next task to execute. This is shown in the flow diagram of Figure V-13.

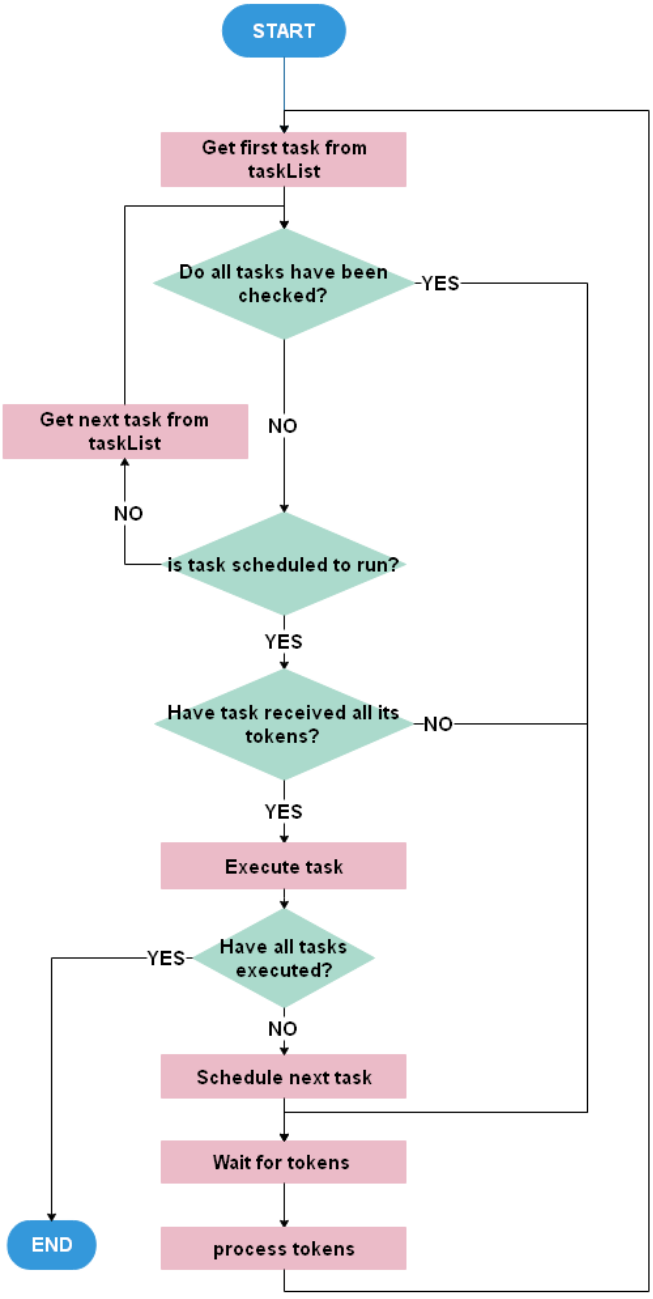


**Figure V-13.** Flow diagram of *runScheduledTask* method.

The process block called *Execute task* in Figure V-13 is defined to model the execution time of a task using the mean execution time and standard deviation information that the TCG supplies for the task. Also, it generates the tokens that must be sent to other tasks according to the information in the TCG. On the other hand, the process block called *process tokens* analyzes the tokens that arrive from other tasks and assign them to the tasks mapped to the PE.

*b) Source_STP class*

This is a subclass of the *Source* class discussed in section 4). It is responsible for converting tokens generated in *TasksManager* class to packets that can travel in the NoC. Also, *Source_STP* puts tokens

generated by *TasksManager* directed to tasks mapped in the same PE, in the tokenQueue of *TasksManager* class (see Figure V-16). To accomplish this, Source_STP overrides the send method of the Source class as shown in the UML class diagram of Figure V-14.

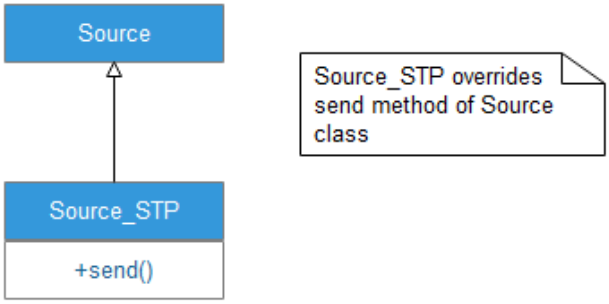


**Figure V-14.** *Source_STP* UML class diagram.

*c) Sink_STP class*

Sink_STP is a subclass of the Sink class discussed in section 4). It is responsible for converting the packets that arrive from other PEs to tokens for the tasks mapped in a PE, putting those tokens in the tokenQueue of *TasksManager* class (see Figure V-16). To accomplish this, Sink_STP overrides the serve method of the Sink class. This is shown in the UML class diagram of Figure V-15.

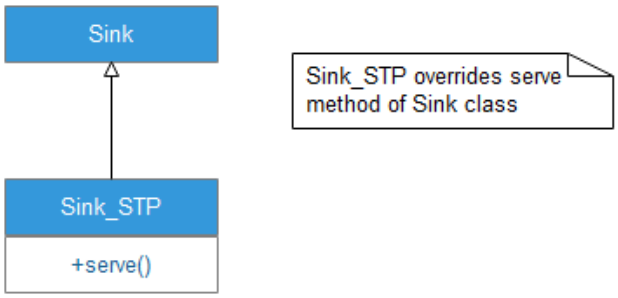


**Figure V-15.** *Sink_STP* UML class diagram.

Figure V-16 summarizes how the Source_STP, Sink_STP, and TasksManager classes work together to model the traffic patterns of a given application. When a Sink_STP object receives a packet, it is processed and tokens are appended to the token queue of the tasks manager. The tasks manager schedule and execute the tasks according to the TCG of a given application, passing information to the Source_STP object which is in charge of generating tokens for tasks mapped in the same PE, and packets that are sent to tasks that are mapped in other PEs.

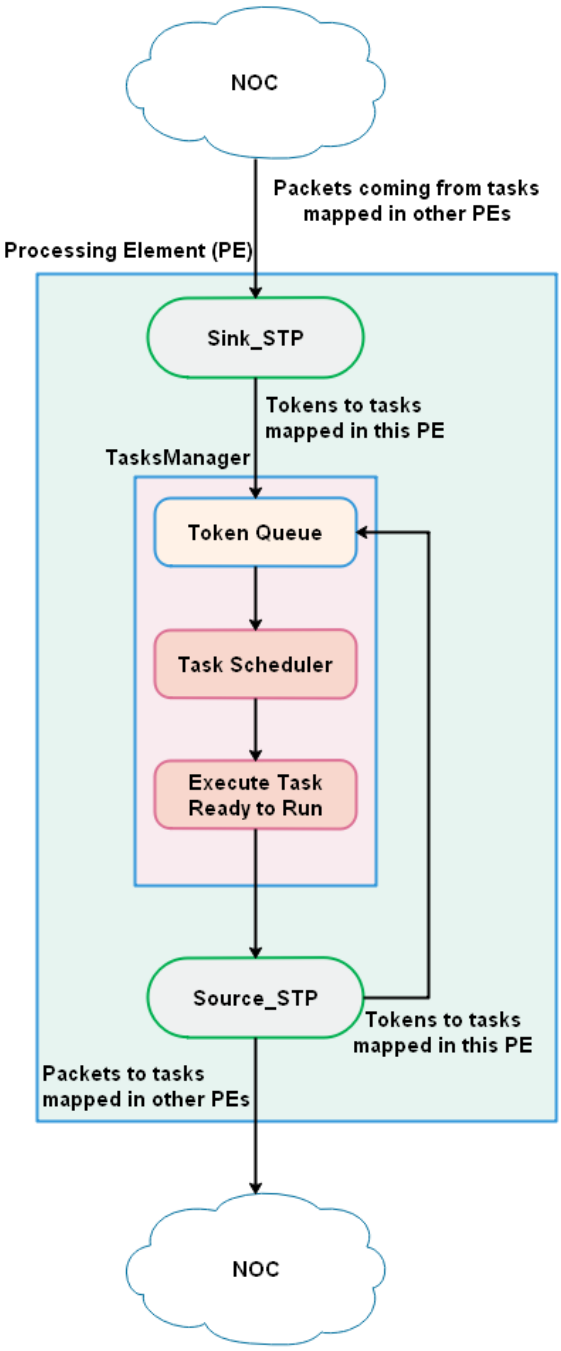


**Figure V-16.** The interplay of Source_STP, Sink_STP, and TasksManager classes.

E. *NoCSimulator Runtime overview*

In Figure V-17, a UML sequence diagram shows the runtime behavior of NoCSimulator. NoCSimulator runtime is divided into three sections: *simulation startup*, *simulation running*, and *simulation end*.

- **Simulation startup:** In this phase, the NoC topology is built up using the *setupTopology* method, and uniform traffic or real traffic pattern is selected as the traffic type used in the simulation. The topology object also sends messages to its routers and PEs to get ready for simulation (*start* method). At the same

time, each router sends messages to its buffers to get ready (*start* simulation) and the PEs do the same with each of their sources and PEs (see Figure V-17).

- **Simulation running:** In this section of the simulation, the PEs are sending packets to other PEs in the NoC using the *send* method (which in turn uses the *selectedDestination* method) of the source object. Also, each sink object on the PEs is receiving packets from the NoC and processing them using the *serve* method. Additionally, the routers get packets from other routers and the PEs through their buffer objects. When a packet is received at a buffer (*get* method) it signals the router to process it (*serve* method). When the router is notified by the buffer, it processes the incoming packet by determining the next hop to its destination using the routing table constructed by Dijkstra's shortest-path algorithm (*calculateRoute* method) and sends it to the next hop using the *put* method (see Figure V-17).

- **Simulation end:** When the simulation ends, the topology object sends messages to its routers and PEs to calculate their performance metrics (*calculateStatistics* method). In turn, each PE sends the same message to its source and sink objects. The performance metrics of each PE are returned to the topology object to calculate the overall performance metrics of the NoC (see Figure V-17). A detailed description of the *calculateStatistics* methods is given in sections b), b), d), and b).

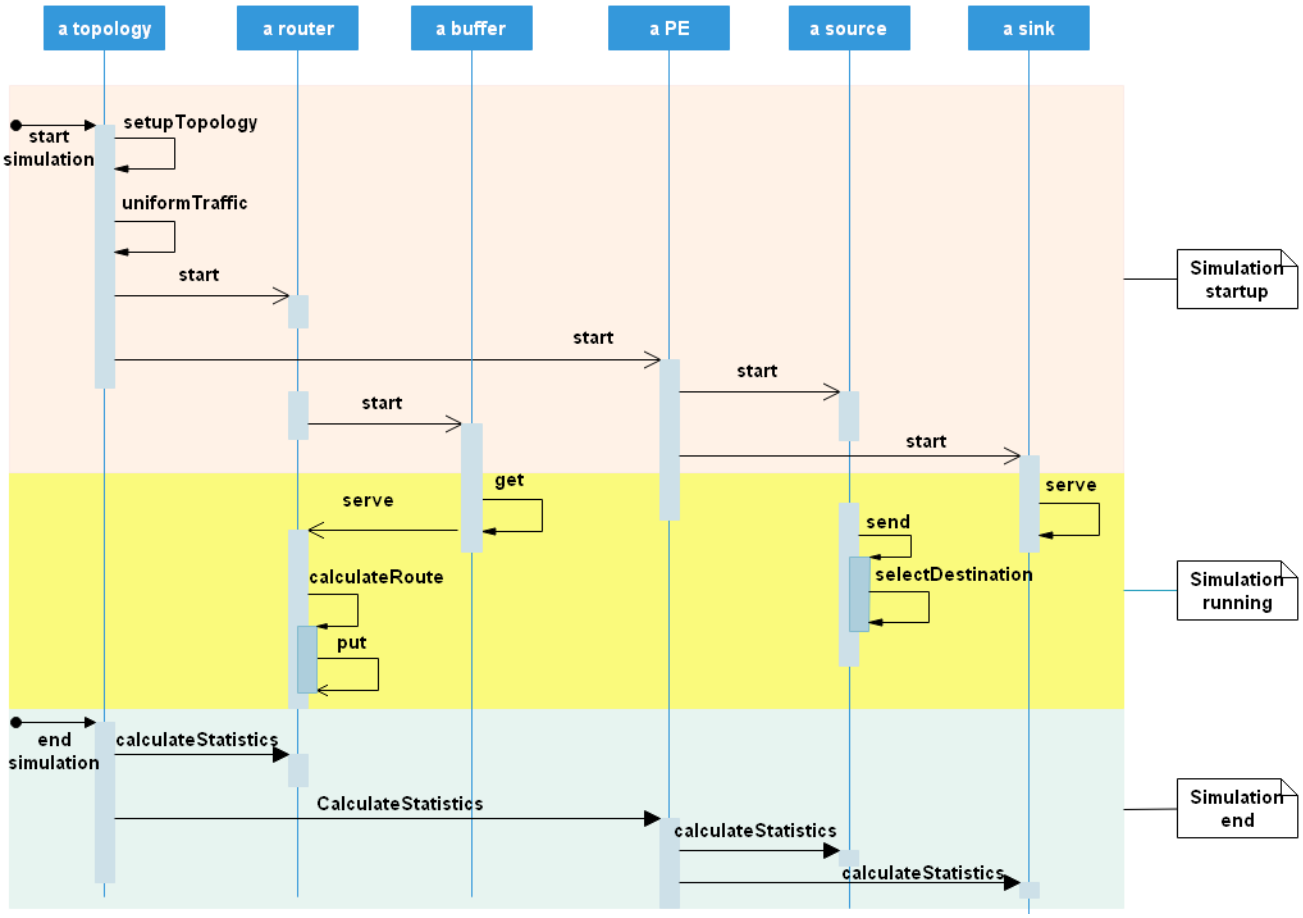


**Figure V-17.** NoCSimulator runtime sequence diagram.

## F. *CONCLUSION*

In this chapter, a detailed description of NoCSimulator was given. It included how topologies, Processing Elements (PEs) and Routers are modeled. Also, how Queueing Theory is used by NoCSimulator to estimate the performance metrics of an NoC is discussed. Then, the integration of the MCSL traffic suite into NoCSimulator was explained. This integration, enables the main feature of NoCSimulator, its ability to process traffic patterns based in real applications. Also, NoCSimulator is capable of modeling Mesh, Ring, Fat-tree and Flattlened Butterfly topologies, these features, differentiate NoCSimulator from similar tools reviewed in the state of the art (section 7)) and give the possibility of obtaining accurate NoC performance estimates of real applications using a tool based in Queueing Theory. In the next chapter, validation test for NoCSimulator are performed. With this Chapter, objective *c* (see section b)) is met.

# VI. VALIDATION TESTS FOR NOCSIMULATOR

**CONTENTS**



## A. *Introduction*

As stated in I, cycle-accurate simulation is the primary tool used by systems architects to evaluate the performance of a NoC. In this thesis, NoCSimulator is proposed as an alternative to cycle-accurate simulation, especially for early stages in the design process. Thus in this chapter, NoCSimulator is put to test. For each topology supported (Mesh, Ring, Flattened Butterfly, and Fat-tree) tests using synthetic traffic patterns and traffic patterns of real applications are performed. The objective of these tests is to evaluate how well can NoCSimulator estimate the performance metrics of a NoC (latency, throughput, etc.) when compared with the cycle-accurate simulator gem5. For the tests using synthetic traffic, an unmodified version of the gem5's garnet module is used (see section G). For the tests using traffic patterns of real applications Garnet_STP is used (see section H). Also, from I and section 7) it is expected that a Queueing Theory based tool as NoCSimulator be faster that a cycle-accurate simulator. Thus, the execution times of NoCSimulator and gem5 are compared to evaluate how fast NoCSimulator is with respect to gem5.

This chapter is divided as follows: Section B explains how the test are to be performed. Section C presents tests results for Mesh topology, section D is for Ring topology, section E is for Flattened Butterfly topology , and section 2) is for Fat-tree topology. Finally, in section F the execution times of NoCSimulator and gem5 are compared. With this chapter, specific objective *e* (see section b)) is fulfilled.

B. *Experimental setup*

For each topology (Mesh, Ring, Flattened Butterfly and Fat Tree), three network sizes are studied (16 PEs, 32 PEs and 64 PEs). The performance metrics of interest are: average throughput of the NoC, average throughput per PE, average latency, and average execution cycles. TCGs of the Fpppp, Robot, and Sparse applications from the MCSL traffic suite are used as benchmarks, and their details are presented in Table VI-1. The tasks that describe these applications are mapped to every topology in both Garnet_STP and NoCSimulator according to their respective TCGs. Each test is repeated 100 times to account for stochastic variations and the results are presented in plots and summary tables. The intention of these experiments is to compare how closely can NoCSimulator reproduce the NoC performance results obtained when using a cycle-accurate simulator, and thus, the differences in performance estimation between NoCSimulator and Garnet_STP are calculated. The statistical significance of these differences (errors) is evaluated using hypothesis testing under the following null and alternate hypotheses:

$$H_0: \mu_x - \mu_y = 0 \quad versus \quad H_a: \mu_x - \mu_y \neq 0$$

Where $\mu_x$ represents the mean value of one of the performance metrics obtained using Garnet_STP and $\mu_y$ represents the same performance metric but obtained using NoCSimulator. Thus, the null hypothesis ($H_0$) proposes that the mean value of the performance metrics obtained using NoCSimulator and Garnet_STP are equal, and the alternate hypothesis ($H_a$) proposes that these performance metrics are not equal. Finally, the null hypothesis is rejected when p-value < 0.05 (rule of thumb used in statistics).

**Table VI-1.** Applications Used as Benchmarks[7]

| Application | Description | No. of tasks | No. of edges |
|---|---|---|---|
| Fpppp | SPEC95 Fpppp: a chemical program performing multi-electron integral derivatives. | 334 | 1145 |
| Robot | Newton-Euler dynamic control calculation for the 6-degrees-of-freedom stanford manipulator. | 88 | 131 |
| Sparse | Random sparse matrix solver for electronic circuit simulation. | 96 | 67 |

[7] Descriptions taken from [28].

C. *Mesh topology*

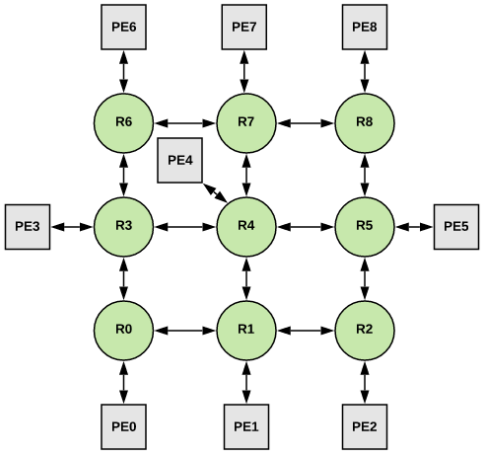


**Figure VI-1.** Mesh Topology - 3x3 (9 PEs).

A mesh topology (see Figure VI-1) is a 2-dimensional grid with *k* nodes in each dimension and links between neighboring nodes. This topology is found in many NoC designs because its 2-D characteristics are easy to map in the metal substrate of the chip [127]. In this topology, each of the routers serves simultaneously as an input/output terminal for the PEs, and a switching node of the network. However, the lack of edge symmetry (routers at the edges of the mesh are not connected), can cause load imbalance for many traffic patterns, as the demand for the central channels can be significantly higher than for the edge channels [128].

*1) Synthetic traffic simulation*

As a first approach to NoCSimulator validation, tests using uniform traffic are performed. Figure VI-2 shows a plot of average latency vs throughput using NoCSimulator, Booksim [17], and Garnet [18] (for a remainder of the latency and throughput concepts see section D). The simulation parameters are shown in Table VI-2. In this test, all PEs inject packets at a constant rate, and the latency of the network is recorded after the network has reached steady-state behavior. For each point in the chart, the test is repeated several times to account for stochastic variations, and the latency results are then averaged.

**Table VI-2.** Simulation Parameters

| Item | Value |
|---|---|
| Topology | Mesh 4x4 |
| Routing protocol | XY |
| Traffic pattern | Uniform[8] |
| Service time at routers (cycles) | 3 |
| Buffer size (flits) | 10 |
| Virtual channels | 1 |
| Packet size (flits) | 1 |

[8] Each PE has the same probability to send a packet to the other PEs in the NoC.

In Figure VI-2, NoCSimulator has a discrepancy between 6 and 8 cycles for estimating the average latency of the NoC at low traffic rates when compared with both BookSim and Garnet, which is reduced to around 1 cycle for traffic rates higher than the saturation rate of the network when compared with BookSim. This discrepancy between results was expected since NoCSimulator is based on Queuing Theory and does not account for the hardware details of the routers that Booksim and Garnet do consider. On the other hand, the latency behavior of Garnet is quite different, growing unbounded for traffic rates higher than the saturation rate. This behavior is attributed to the way Garnet models traffic contention at the PEs, which appends packets at the cache buffers no matter if there's packet contention in the NoC. Thus, the latency of the packets at the caches keeps growing. This behavior doesn't model the real behavior of a PE, because a PE stops sending messages to the cache when the cache buffer is full, thus limiting the maximum latency of packets, as is modeled by both BookSim and NoCSimulator. This issue was consulted with the designer of Garnet Professor. T. Krishna at Georgia Tech [129], and the messages exchanged with him are available here [130]. However, from Figure VI-2 it can be seen that NoCSimulator, in general, captures the latency-throughput relation of the NoC, especially the saturation throughput of the NoC (0.07 flit/cycle/PE) and the average latency of the NoC after reaching saturation (35.7 cycles).

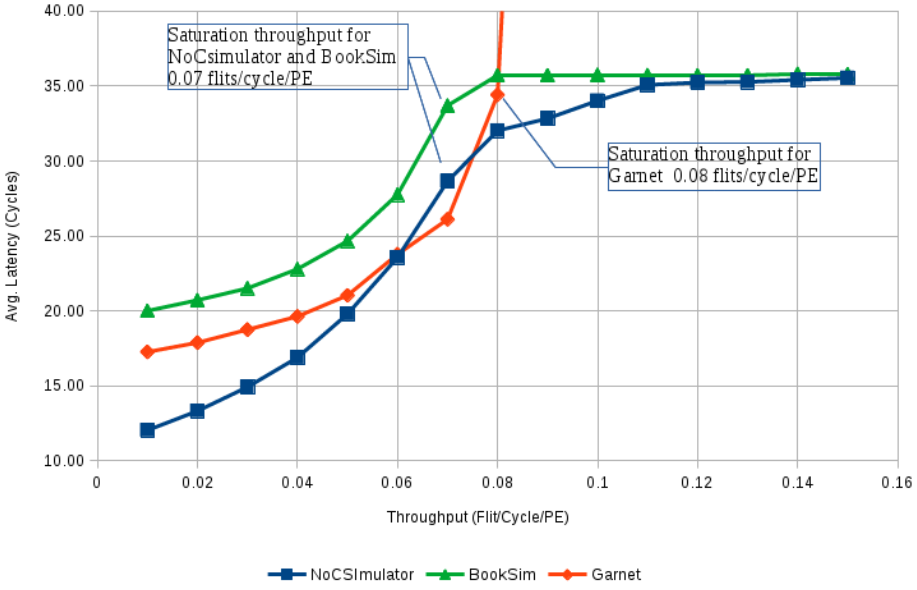


**Figure VI-2.** Average Latency vs Throughput - NoCSimulator, Garnet, and BookSim for Mesh topology (16-PEs 4x4).

*2) Estimation of NoC performance using statistical traffic patterns of real applications*

In this section, the suitability of NoCSimulator to estimate the performance of a Mesh topology with statistical traffic patterns (STP) of real applications is presented. To do this, benchmarks from the MCSL traffic suite [28] are used. For validation purposes, the same benchmarks were tested in the modified version of Garnet (Garnet_STP, see section 1)). In Table VI-1, the traffic patterns used as benchmarks are described. The packets are composed of eight flits.

The performance metrics considered for these tests are throughput, throughput per PE, latency, and execution cycles. Figure VI-3 shows the results for network throughput and throughput per PE. Figure VI-4 shows the results for latency and execution cycles.

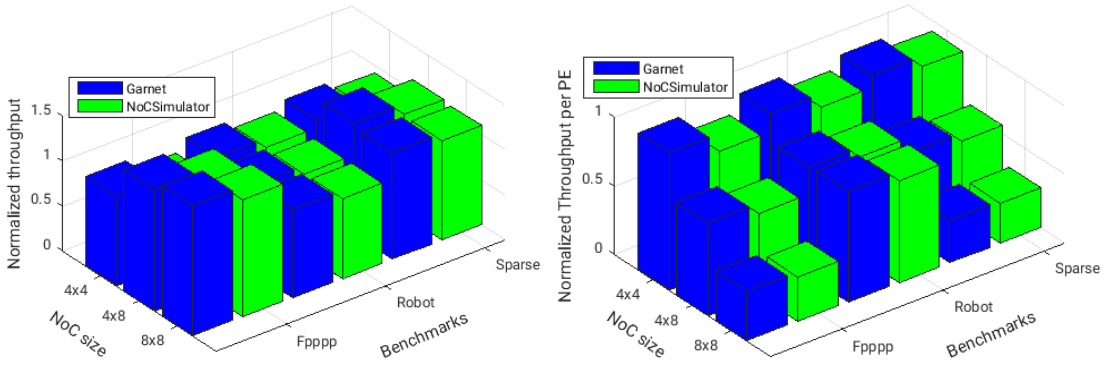


**Figure VI-3.** Performance of benchmarks running in different Mesh sizes. Throughput and throughput per PE.

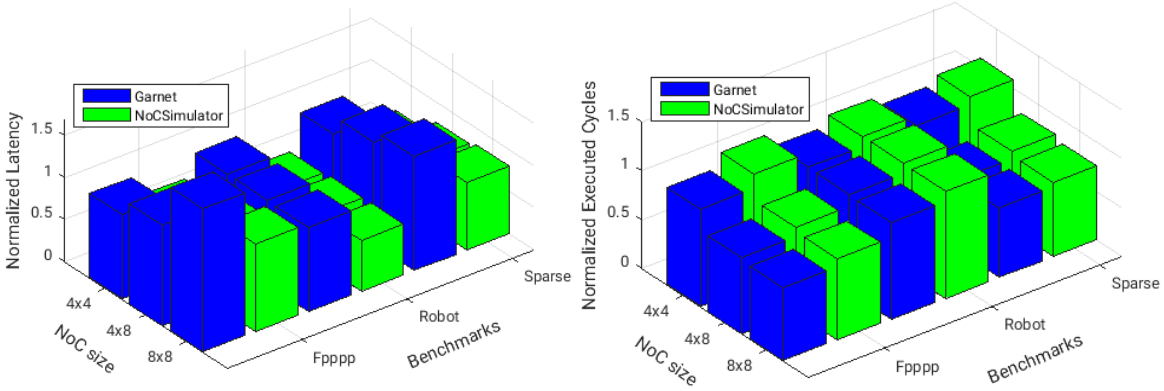


**Figure VI-4.** Performance of benchmarks running in different Mesh sizes. Latency and execution cycles.

The results of these metrics for Garnet_STP simulations on the 16-PE Mesh are used to normalize the results obtained for the other NoC sizes. This, to compare visually the differences of NoCSimulator and Garnet_STP for each application and network size.

Figures 6-3,4 show that the three applications used as benchmarks have no strong similarities with each other, i.e., the results on NoC performance are determined not only by network parameters (topology size, routing protocol, etc.) but also by the characteristics of the traffic patterns used as benchmarks. From Figures 6-3,4 is evident that NoCSimulator and Garnet_STP are close in their estimations of NoC performance for all metrics except for latency. This sub-estimation of latency is in part because this tool is based on Queueing Theory and thus it does not model any hardware details found in Garnet_STP (routing logic, allocators, crossbar switch, router pipeline, etc.). But for the most part, the latency sub-estimation can be attributed to the unbound growth of the latency of packets when Garnet_STP is working under saturation conditions, as discussed in section 1). Nevertheless, from Figure VI-4 NoCSimulator latency results have the same tendencies of Garnet_STP latency results, and thus they provide the same insights about the latency behavior of the different benchmarks in different Mesh sizes. In Table VI-6, the results of NoCSimulator and Garnet_STP are summarized.

*a) Hypothesis tests for Mesh topology*

Table VI-3 presents the simulation results for the Fpppp application on the three mesh topologies studied. For each metric, the error (i.e., the difference between the result obtained using Garnet_STP and NoC simulator) is shown, including its value in percentage. Also, to evaluate the statistical significance of these errors, the p-value is calculated.

**Table VI-3.** Performance metrics for Fpppp application – Mesh topology

| Topology | | Metric | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **Mesh 4x4** | Garnet_STP | 0.204 | 0.013 | 18.585 | 262260 |
| | NoCSimulator | 0.199 | 0.012 | 13.911 | 268041 |
| | diff | 0.005 | 2.8e-4 | 4.674 | -1.3e4 |
| | diff (%) | 2.3 | 2.3 | 25.1 | -2.2 |
| | p-value | 0.021 | 0.021 | 1e-169 | 0.028 |
| **Mesh 4x8** | Garnet_STP | 0.289 | 0.009 | 22.232 | 194860 |
| | NoCSimulator | 0.280 | 0.009 | 17.506 | 201249 |
| | diff | 0.010 | 3.0e-4 | 4.726 | -6389 |
| | diff (%) | 3.4 | 3.4 | 21.3 | -3.3 |
| | p-value | 0.004 | 0.004 | 9e-161 | 0.007 |
| **Mesh 8x8** | Garnet_STP | 0.310 | 0.005 | 28.895 | 185331 |
| | NoCSimulator | 0.302 | 0.005 | 19.513 | 189984 |
| | diff | 0.008 | 1.2e-4 | 9.383 | -4654 |
| | diff (%) | 2.5 | 2.5 | 32.5 | -2.5 |
| | p-value | 0.034 | 0.034 | 6.0e-66 | 0.036 |

From Table VI-3, the differences in estimation between NoCSimulator and Garnet are small for all metrics except latency. For example, for the Mesh_4x4 topology, the error in estimation for average throughput (AT) and average throughput per PE (AVP) is 2.3%; for average latency (AL) the error is of 25.1%, and for average execution cycles (AEC) the error is -2.2% (a negative percentage can be interpreted as that NoCSimulator overestimates the performance metric when compared to Garnet_STP).

The p-value is used to evaluate the statistical significance of the errors observed during the tests under the null and alternate hypotheses shown in section B. The p-values for all metrics are less than 0.05, thus, the errors observed in the tests, have statistical significance (i.e., the null hypothesis can be rejected, and the magnitude of the errors observed can be expected in subsequent experiments). However, the errors observed in AT, ATPE and AEC metrics are relatively small, thus it can be said that this metrics are estimated accurately by NoCSimulator. The notable inaccuracy in the estimation of AL was explained above.

Table VI-4 and Table VI-5 present the same information for both the Robot application and the Sparse application.

**Table VI-4.** Performance metrics for Robot application – Mesh topology

| Topology | | Metric | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **Mesh 4x4** | Garnet_STP | 0.022 | 1.350e-3 | 16.870 | 216725 |
| | NoCSimulator | 0.021 | 1.342e-3 | 9.851 | 218077 |
| | diff | 1.359e-4 | 8e-6 | 7.018 | -1352 |
| | diff (%) | 0.6 | 0.6 | 41.6 | -0.6 |
| | p-value | 0.404 | 0.404 | 1e-196 | 0.418 |
| **Mesh 4x8** | Garnet_STP | 0.021 | 6.614e-4 | 16.901 | 216385 |
| | NoCSimulator | 0.021 | 6.521e-4 | 10.266 | 219520 |
| | diff | 2.995e-4 | 9e-6 | 6.635 | -3135 |
| | diff (%) | 1.4 | 1.4 | 39.3 | -1.4 |
| | p-value | 0.078 | 0.078 | 1e-189 | 0.081 |
| **Mesh 8x8** | Garnet_STP | 0.021 | 3.305e-4 | 16.908 | 216583 |
| | NoCSimulator | 0.021 | 3.255e-4 | 10.284 | 219793 |
| | diff | 3.179e-4 | 4e-6 | 6.624 | -3210 |
| | diff (%) | 1.5 | 1.5 | 39.2 | -1.5 |
| | p-value | 0.047 | 0.047 | 3e-194 | 0.051 |

**Table VI-5.** Performance metrics for Sparse application – Mesh topology

| Topology | | Metric | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **Mesh 4x4** | Garnet_STP | 0.148 | 0.009 | 16.585 | 70391 |
| | NoCSimulator | 0.138 | 0.009 | 9.759 | 75242 |
| | diff | 0.01 | 6.212e-4 | 6.825 | -4851 |
| | diff (%) | 6.6 | 6.6 | 41.2 | -6.9 |
| | p-value | 8e-19 | 8e-19 | 3e-211 | 3e-17 |
| **Mesh 4x8** | Garnet_STP | 0.183 | 0.006 | 20.119 | 51208 |
| | NoCSimulator | 0.172 | 0.005 | 12.632 | 54201 |
| | diff | 0.011 | 3.381e-4 | 7.487 | -2993 |
| | diff (%) | 5.9 | 5.9 | 37.2 | -5.8 |
| | p-value | 1e-13 | 1e-13 | 5e-274 | 3e-12 |
| **Mesh 8x8** | Garnet_STP | 0.175 | 0.003 | 22.534 | 49866 |
| | NoCSimulator | 0.168 | 0.003 | 13.468 | 51928 |
| | diff | 0.007 | 1.106e-4 | 9.066 | -2062 |
| | diff (%) | 4.0 | 4.0 | 40.2 | -4.1 |

| | | | | | |
|---|---|---|---|---|---|
| | p-value | 4e-6 | 4e-6 | 2e-283 | 6e-6 |

**Table VI-6.** Performance summary - Mesh Topology

| Attribute | NoCSimulator vs Garnet_STP | |
|---|---|---|
| | **Avg. diff.** | **Max. diff** |
| Throughput | 3.1 % | 6.6% |
| Throughput per PE | 3.1 % | 6.6% |
| Latency | 35.3% | 41.6% |
| Execution cycles | -3.1% | -6.9% |

D. *Ring Topology*

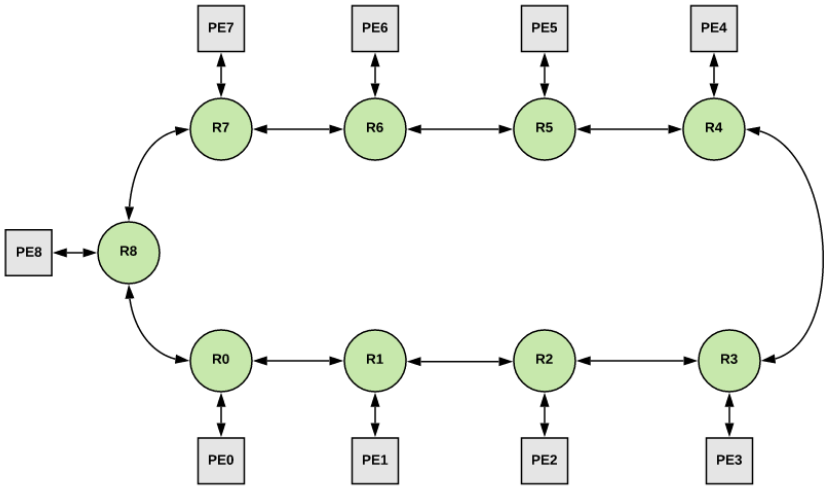


**Figure VI-5.** Ring Topology (9 PEs).

In a ring topology (Figure VI-5) every router in the network is connected to two other routers, such that the first and the last routers are connected (closing the ring). The packets are transmitted over the network through each of the routers in the ring until they reach the destination node. The Ring topology is primarily used in small multi-core systems because of its better throughput performance in comparison with other topologies, like a mesh of the same size. However, as the number of nodes increases, the performance of this topology degrades in comparison with other topologies due to the increase in hop count [45].

In Figure VI-6, a plot of average latency vs. throughput for a Ring of sixteen PEs using NoCSimulator and Garnet is presented. Simulation parameters are shown in Table VI-7. In this test, all PEs inject packets at a constant rate, and the latency of the network is recorded after the network has reached steady-state behavior. For each point in the chart, the test is repeated several times to account for stochastic variations, and the latency results are then averaged.

**Table VI-7.** Simulation Parameters - Ring Topology.

| Item | Value |
|---|---|
| Topology | Ring – 16 PEs |
| Routing protocol | Shortest-path[9] |
| Traffic pattern | Uniform[10] |

[9] The shortest path between each PE is used to route packets.
[10] Each PE has the same probability to send a packet to the other PEs in the NoC.

| | |
|---|---|
| Service time at routers (cycles) | 3 |
| Buffer size (flits) | 10 |
| Virtual channels | 1 |
| Packet size (flits) | 1 |

From Figure VI-6, it can be seen that NoCSimulator and Garnet have a discrepancy of about 6 cycles when estimating the latency at traffic rates below saturation throughput. As stated in section 1), this discrepancy was expected due to the different nature of both NoCSimulator and Garnet. Also, for rates higher than the saturation rate the latency grows unbound in Garnet. This behavior is the same as for the Mesh topology and it was discussed in section 1). The saturation rate of this topology is 0.06 flit/cycle/PE, which is lower than the saturation rate of a Mesh of the same size (Figure VI-2). This result is consistent with the fact that in a Ring topology as the number of PEs increases the available throughput decreases due to the increase in packet contention in the limited paths offered by this topology [131].

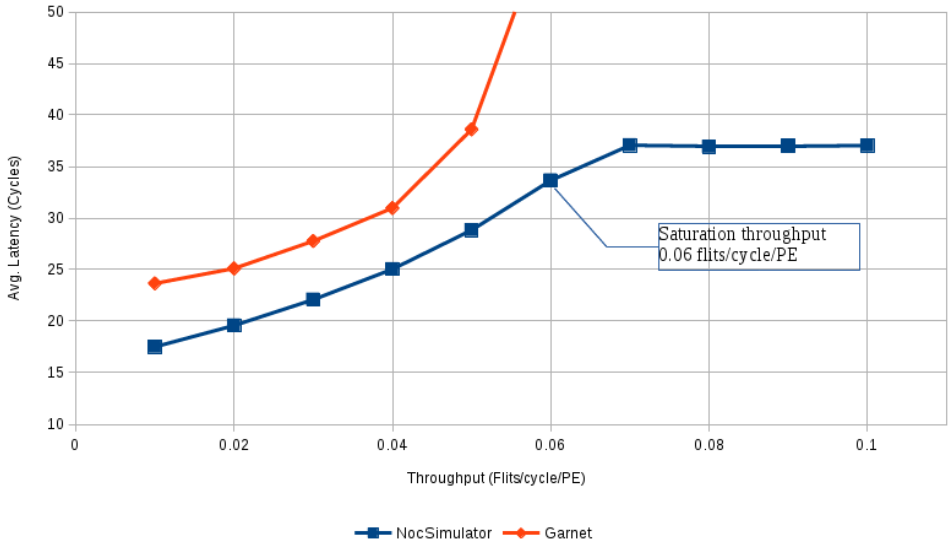


**Figure VI-6.** Average Latency vs Throughput - Ring Topology (16-PEs).

*1) Estimation of NoC performance using statistical traffic patterns of real applications*

In this section, the suitability of NoCSimulator to estimate the performance of an NoC Ring topology with STP traffic patterns is shown. The same benchmarks used for the Mesh topology were used and they are described in Table VI-1. Also, these benchmarks were used in Garnet_STP for validation purposes. Three topology sizes are tested: 16-PE Ring, 32-PE Ring, and 64-PE Ring. The packets are composed of eight flits. The results of these metrics for Garnet_STP simulations on the 16-PE Ring are used to normalize the results obtained for the other NoC sizes. Figure VI-7 shows the performance results for the metrics network throughput and network throughput per PE. Figure VI-8 shows the performance results for the metrics average latency and simulated cycles.

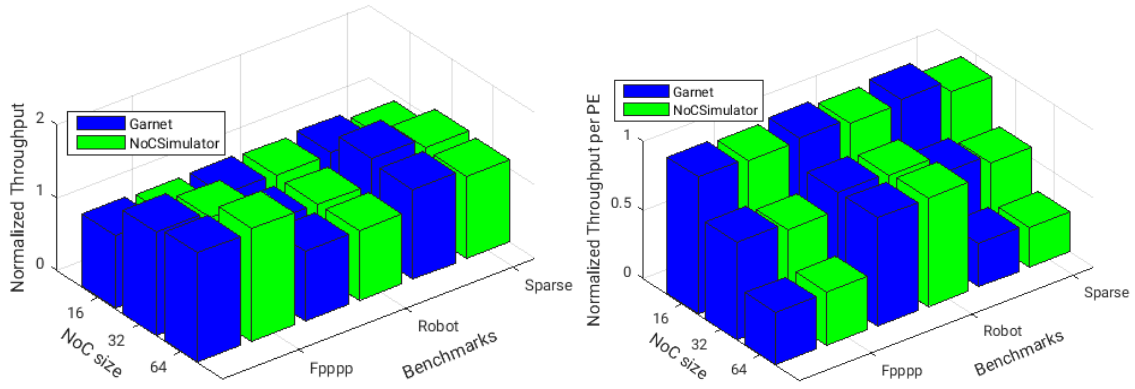


**Figure VI-7.** Performance of applications running in different NoC sizes. Throughput and throughput per PE – Ring topology.

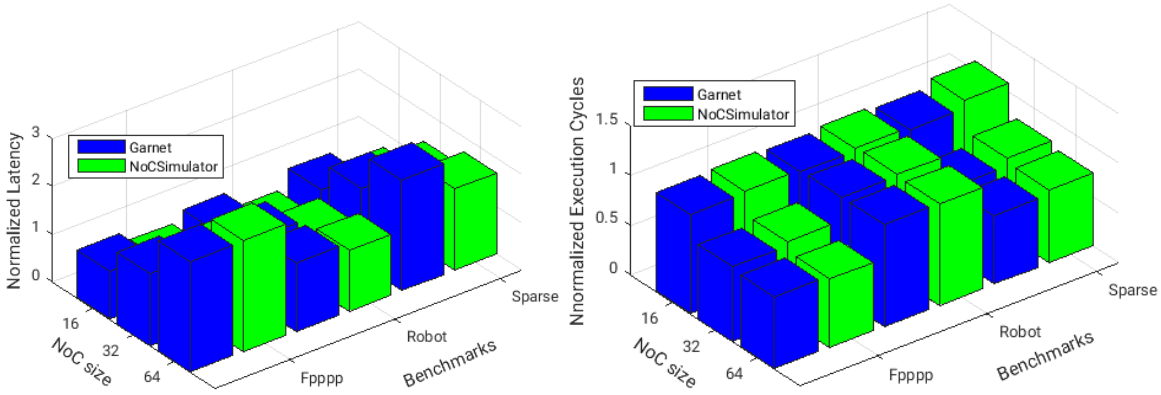


**Figure VI-8**. Performance of applications running on different NoC sizes. Average latency and execution cycles - Ring topology.

From Figures 6-7,8 it can be seen that the applications used as benchmarks have no strong similarities with each other. As in the case of the Mesh topology, the results on performance are determined not only by network parameters (topology size, routing protocol, etc.) but also by the traffic characteristics of the benchmarks used.

In Table VI-11, a comparison of results between NoCSimulator and Garnet_STP is summarized. These results show that NoCSimulator and Garnet_STP are close on all the metrics for all the cases analyzed, except in the latency results (Figure VI-8). This discrepancy in the estimation of latency can be mainly attributed to the over-estimation of latency in Garnet_STP as discussed in section 1). Regardless of the discrepancy in the latency estimations by both NoCSimulator and Garnet_STP, in Figure VI-8 it is seen that both results have the same tendencies, and thus the same insights about the latency behavior across the different NoC sizes and benchmarks can be reached.

*a) Hypothesis tests for Ring topology*

**Table VI-8.** Performance metrics for Fpppp application – Ring topology

| Topology | | Metric | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **Ring16** | Garnet_STP | 0.209 | 0.013 | 25.910 | 255805 |
| | NoCSimulator | 0.199 | 0.012 | 20.501 | 268099 |
| | diff | 0.010 | 5.9e-4 | 5.409 | -12295 |
| | diff (%) | 4.6 | 4.6 | 20.9 | -4.8 |
| | p-value | 7e-7 | 7e-7 | 2e-150 | 7e-7 |
| **Ring32** | Garnet_STP | 0.285 | 0.009 | 38.999 | 198102 |
| | NoCSimulator | 0.279 | 0.009 | 36.697 | 201850 |
| | diff | 0.005 | 1.7e-4 | 2.303 | -3748 |
| | diff (%) | 1.9 | 1.9 | 5.9 | -1.9 |
| | p-value | 0.110 | 0.110 | 3e-73 | 0.126 |
| **Ring64** | Garnet_STP | 0.308 | 0.005 | 59.796 | 186850 |
| | NoCSimulator | 0.300 | 0.005 | 60.638 | 192362 |
| | diff | 0.008 | 1.2e-4 | -0.841 | -5513 |
| | diff (%) | 2.5 | 2.5 | -1.4 | -3.0 |
| | p-value | 0.055 | 0.055 | 3.4e-4 | 0.029 |

**Table VI-9.** Performance metrics for Robot application – Ring topology

| Topology | | Metric | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **Ring16** | Garnet_STP | 0.022 | 0.001 | 22.555 | 215699 |
| | NoCSimulator | 0.021 | 0.001 | 13.876 | 219795 |
| | diff | 4.5e-4 | 2.8e-5 | 8.680 | -4095 |
| | diff (%) | 2.1 | 2.1 | 38.5 | -1.9 |
| | p-value | 0.008 | 0.008 | 1e-244 | 0.021 |
| **Ring32** | Garnet_STP | 0.021 | 0.001 | 31.022 | 216925 |
| | NoCSimulator | 0.021 | 0.001 | 23.295 | 220697 |
| | diff | 4.1e-4 | 1.3e-5 | 7.728 | -4043 |
| | diff (%) | 2.0 | 2.0 | 24.9 | -1.9 |
| | p-value | 0.014 | 0.014 | 2e-239 | 0.019 |
| **Ring64** | Garnet_STP | 0.021 | 3.2e-4 | 32.443 | 220343 |
| | NoCSimulator | 0.021 | 3.2e-4 | 28.604 | 218826 |
| | diff | 4.7e-5 | 7.4e-7 | 3.839 | 1517 |
| | diff (%) | 0.2 | 0.2 | 11.8 | 0.7 |

| | | | | | |
|---|---|---|---|---|---|
| | p-value | 0.785 | 0.785 | 9e-166 | 0.704 |

**Table VI-10.** Performance metrics for Sparse application – Ring topology

| Topology | | Metric | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **Ring16** | Garnet_STP | 0.147 | 0.009 | 24.317 | 70713 |
| | NoCSimulator | 0..136 | 0.009 | 16.963 | 76114 |
| | diff | 0.011 | 6.8e-4 | 7.354 | -5410 |
| | diff (%) | 7.5 | 7.5 | 30.2 | -7.6 |
| | p-value | 1e-22 | 1e-22 | 2e-218 | 5.1e-20 |
| **Ring32** | Garnet_STP | 0.184 | 0.006 | 38.388 | 50700 |
| | NoCSimulator | 0.169 | 0.005 | 33.026 | 55366 |
| | diff | 0.015 | 4.7e-4 | 5.362 | -4667 |
| | diff (%) | 8.3 | 8.3 | 14.0 | -9.2 |
| | p-value | 7e-23 | 7e-23 | 5e-132 | 3e-21 |
| **Ring64** | Garnet_STP | 0.175 | 0.003 | 56.339 | 49985 |
| | NoCSimulator | 0.168 | 0.003 | 41.825 | 52144 |
| | diff | 0.007 | 1.1e-4 | 14.514 | -2158 |
| | diff (%) | 4.3 | 4.3 | 25.8 | -4.3 |
| | p-value | 6e-7 | 6e-7 | 1e-153 | 2e-6 |

**Table VI-11.** Performance summary - Ring topology

| Attribute | NoCSimulator vs Garnet_STP | |
|---|---|---|
| | **Avg. diff.** | **Max. diff** |
| Throughput | 3.7% | 8.3% |
| Throughput per PE | 3.7% | 8.3% |
| Latency | 19.0% | 38.5% |
| Execution cycles | -3.8% | -9.2% |

## E. *Flattened Butterfly Topology*

The flattened butterfly topology is used with high-radix routers (i.e., routers with many I/O channels). The flattened butterfly is derived by combining (or flattening) the routers in each row of a conventional butterfly topology while preserving the inter-router connections [47], [132], [133]. When minimal routing (shortest-path) is used, PEs in this network are separated by only 2 hops, which is a significant improvement over the hop count of a mesh topology. The 3x3 version of this topology is shown in Figure VI-9.

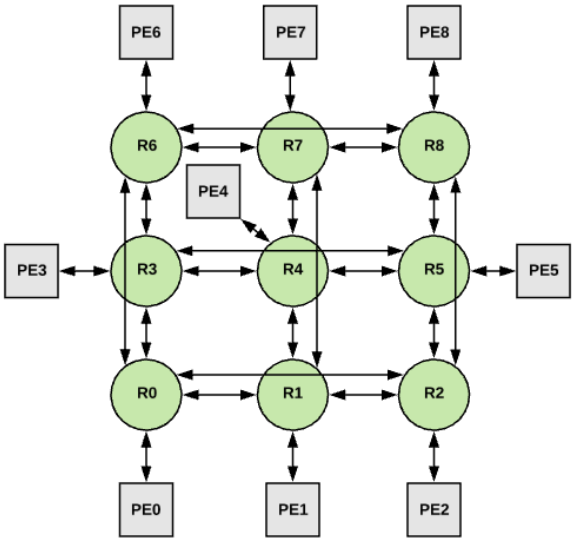


**Figure VI-9.** Flattened Butterfly topology (9-PEs 3x3).

In Figure VI-10 a plot of average latency vs. throughput for a 4x4 flattened butterfly using NoCSimulator and Garnet is presented. Simulation parameters are shown in Table VI-12. In this test, all PEs inject packets at a constant rate, and the latency of the network is recorded after the network has reached steady-state behavior. For each point in the chart, the test is repeated several times to account for stochastic variations, and the latency results are then averaged.

**Table VI-12.** Simulation Parameters - Flattened Butterfly Topology.

| Item | Value |
|---|---|
| Topology | Flattened Butterfly 4x4 |
| Routing protocol | XY-Shortest-path |
| Traffic pattern | Uniform[11] |
| Service time at routers (cycles) | 3 |
| Buffer size (flits) | 10 |
| Virtual channels | 1 |
| Packet size (flits) | 1 |

In Figure VI-10 there is a difference of about 7 cycles between latency estimations of NoCSimulator and Garnet at traffic rates below saturation. As stated earlier, this difference can be attributed to NoCSimulator being a tool based on Queueing theory, it doesn't model the hardware low-level details that Garnet does. Also, in Figure VI-10 it can be seen that latency estimations from Garnet grow rapidly for traffic rates higher than saturation rate. The reason for this is related to the way Garnet models packet contentions at the PEs and was discussed in section 1). In NoCSimulator saturation throughput of this topology is 0.13 flit/cycle/PE, and the maximum latency is about 26 cycles, both values much better than those of a Mesh of equivalent size (see Figure VI-2). This can be attributed to the lower average hop count of the flattened butterfly topology. These results are consistent with the features described for this topology in the literature [47].

[11] Each PE has the same probability to send a packet to the other PEs in the NoC.

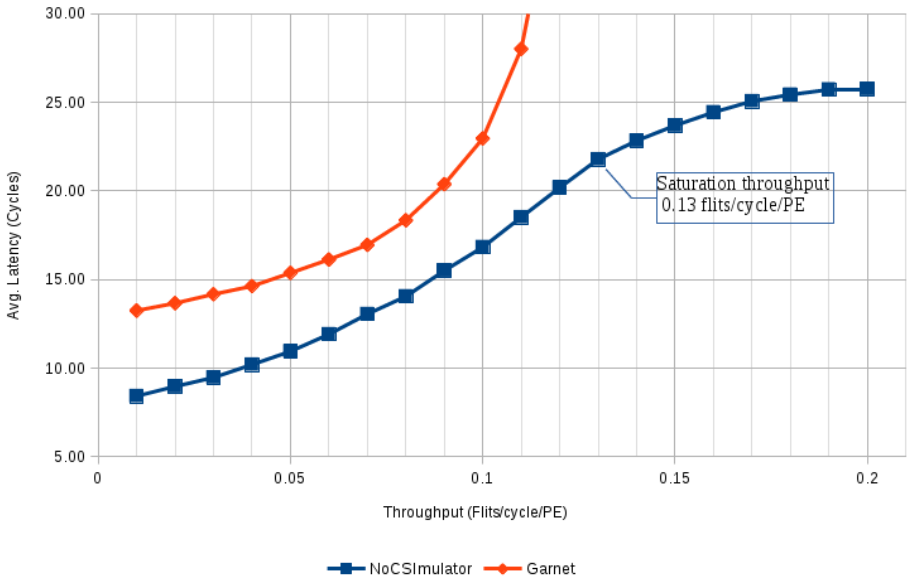


**Figure VI-10.** Average Latency vs Throughput - Flattened Butterfly Topology (16-PEs 4x4).

*1) Estimation of NoC performance using statistical traffic patterns of real applications*

In this section, the suitability of NoCSimulator to estimate the performance of an NoC Flattened Butterfly topology with real traffic patterns is shown. As with Mesh topology, benchmarks from the MCSL traffic suite are used and they are described in Table VI-1. Three NoC sizes are tested: 4x4 (16 PEs), 8x4 (32 PEs), and 8x8 (64 PEs). The packets are composed of eight flits. Figure VI-11 shows the performance results for the metrics network throughput and network throughput per PE. Figure VI-12 shows the performance results for the metrics average latency and simulated cycles.

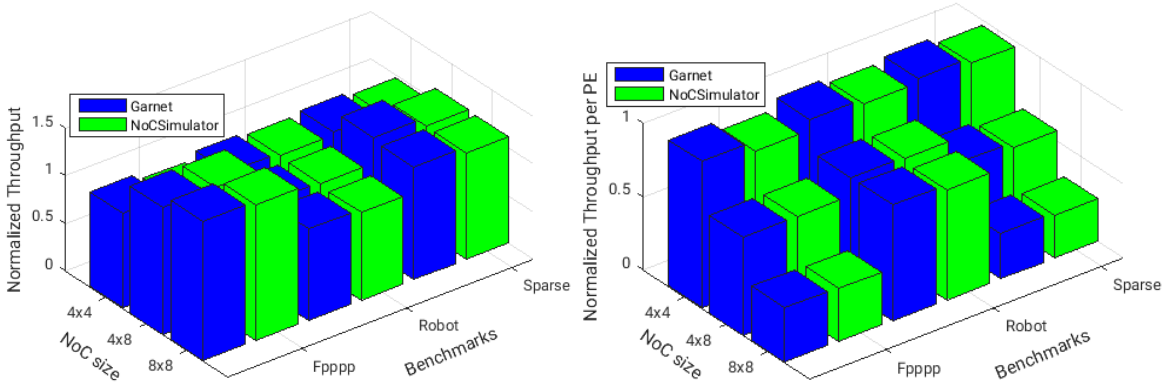


**Figure VI-11.** Performance of applications running in different NoC sizes. Throughput and throughput per PE – Flattened Butterfly topology.

The results of these metrics for Garnet_STP simulations on the 16-PEs Flattened Butterfly are used to normalize the results obtained for the other NoC sizes. The Figures show the relative results for each application on different network sizes. Figures 6-11,12 show that the three applications used as benchmarks have no strong similarities with each other, which indicates that the results on NoC performance are determined not only by network parameters (topology size, routing protocol, etc.) but also by the different traffic characteristics of the benchmarks used.

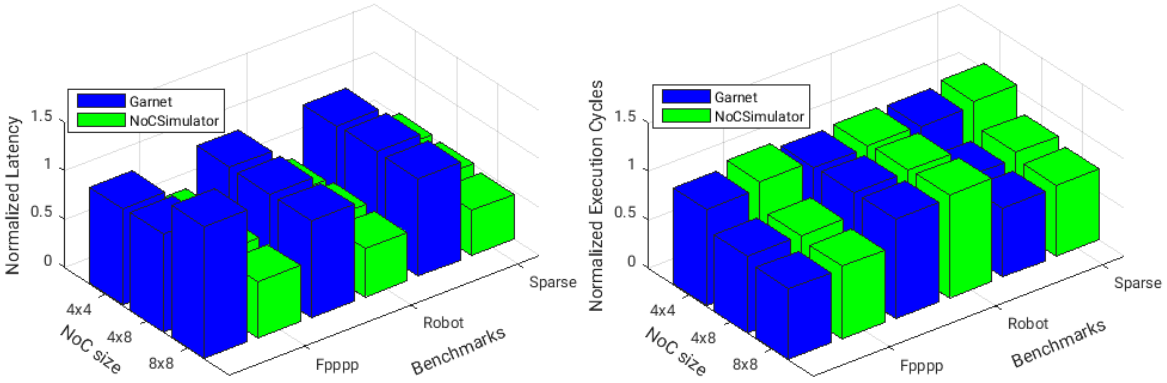


**Figure VI-12.** Performance of applications running on different NoC sizes. Average latency and execution cycles - Flattened Butterfly topology.

In Table VI-16, a comparison of results between NoCSimulator and Garnet_STP is summarized. These results show that NoCSimulator and Garnet_STP are close on all the metrics for all the cases analyzed, except in the latency results (Figure VI-12). As discussed in section 1), this sub-estimation in latency results can be in part due that NoCSimulator doesn't model any low-level hardware details of the NoC unlike Garnet does. However, this discrepancy can be mainly attributed to the unbound latency that Garnet estimates at high traffic rates.

*a) Hypothesis tests for Flattened Butterfly topology*

**Table VI-13.** Performance metrics for Fpppp application – Flatenned Butterfly topology

| **Topology** | | **Metric** | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **FB_4x4** | Garnet_STP | 0.208 | 0.013 | 15.786 | 257016 |
| | NoCSimulator | 0.199 | 0.012 | 9.485 | 267861 |
| | diff | 0.009 | 5.3e-4 | 6.301 | -10854 |
| | diff (%) | 4.1 | 4.1 | 39.9 | -4.2 |
| | p-value | 8e-5 | 8e-5 | 3e-202 | 1e-4 |
| **FB_4x8** | Garnet_STP | 0.288 | 0.009 | 16.209 | 195938 |
| | NoCSimulator | 0.284 | 0.009 | 9.351 | 198493 |
| | diff | 0.004 | 1.2e-4 | 6.858 | -2556 |
| | diff (%) | 1.4 | 1.4 | 42.3 | -1.3 |
| | p-value | 0.219 | 0.219 | 3e-282 | 0.257 |
| **FB_8x8** | Garnet_STP | 0.310 | 0.005 | 21.960 | 185670 |
| | NoCSimulator | 0.303 | 0.005 | 9.180 | 190565 |
| | diff | 0.007 | 1.1e-4 | 12.780 | -4895 |
| | diff (%) | 2.3 | 2.3 | 58.2 | -2.6 |
| | p-value | 0.080 | 0.080 | 6e-79 | 0.058 |

**Table VI-14.** Performance metrics for Robot application – Flattened Butterfly topology

| Topology | | Metric | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **FB_4x4** | Garnet_STP | 0.022 | 0.001 | 14.762 | 216346 |
| | NoCSimulator | 0.021 | 0.001 | 7.295 | 220454 |
| | diff | 4.2e-4 | 2e-5 | 7.466 | -4107 |
| | diff (%) | 1.9 | 1.9 | 50.5 | -1.8 |
| | p-value | 0.016 | 0.016 | 1e-191 | 0.025 |
| **FB_4x8** | Garnet_STP | 0.021 | 0.001 | 14.738 | 217894 |
| | NoCSimulator | 0.021 | 0.001 | 7.398 | 220930 |
| | diff | 3.6e-4 | 1e-5 | 7.340 | -3037 |
| | diff (%) | 1.7 | 1.7 | 49.8 | -1.4 |
| | p-value | 0.035 | 0.035 | 6e-193 | 0.106 |
| **FB_8x8** | Garnet_STP | 0.021 | 3.2e-4 | 14.742 | 218316 |
| | NoCSimulator | 0.021 | 3.1e-4 | 7.387 | 218810 |
| | diff | 2e-5 | 3e-7 | 7.355 | -494 |
| | diff (%) | 0.1 | 0.1 | 49.9 | -0.2 |
| | p-value | 0.887 | 0.887 | 2e-198 | 0.780 |

**Table VI-15.** Performance metrics for Sparse application – Flattened Butterfly topology

| Topology | | Metric | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **FB_4x4** | Garnet_STP | 0.149 | 0.009 | 14.913 | 69846 |
| | NoCSimulator | 0.137 | 0.009 | 7.881 | 75747 |
| | diff | 0.012 | 7.5e-4 | 7.032 | -5901 |
| | diff (%) | 8.1 | 8.1 | 47.2 | -8.4 |
| | p-value | 2e-28 | 2e-28 | 9e-235 | 1e-25 |
| **FB_4x8** | Garnet_STP | 0.183 | 0.006 | 15.131 | 51085 |
| | NoCSimulator | 0.171 | 0.005 | 7.630 | 54745 |
| | diff | 0.012 | 3.8e-4 | 7.502 | -3660 |
| | diff (%) | 6.7 | 6.7 | 49.6 | -7.2 |
| | p-value | 1e-14 | 1e-14 | 6e-250 | 3.9e-15 |
| | Garnet_STP | 0.175 | 0.003 | 15.045 | 49693 |
| | NoCSimulator | 0.169 | 0.003 | 7.073 | 51897 |

| | | | | | |
|---|---|---|---|---|---|
| **FB_8x8** | diff | 0.007 | 1e-4 | 7.972 | -2204 |
| | diff (%) | 3.8 | 3.8 | 53.0 | -4.4 |
| | p-value | 2e-6 | 2e-6 | 2e-278 | 6e-7 |

**Table VI-16.** Performance summary - Flattened Butterfly topology

| Attribute | NoCSimulator vs Garnet_STP | |
|---|---|---|
| | **Avg. diff.** | **Max. diff** |
| Throughput | 3.3% | 8.1% |
| Throughput per PE | 3.3% | 8.1% |
| Latency | 48.9% | 58.2% |
| Execution cycles | -3.5% | -8.4% |

*2) Fat-tree Topology*

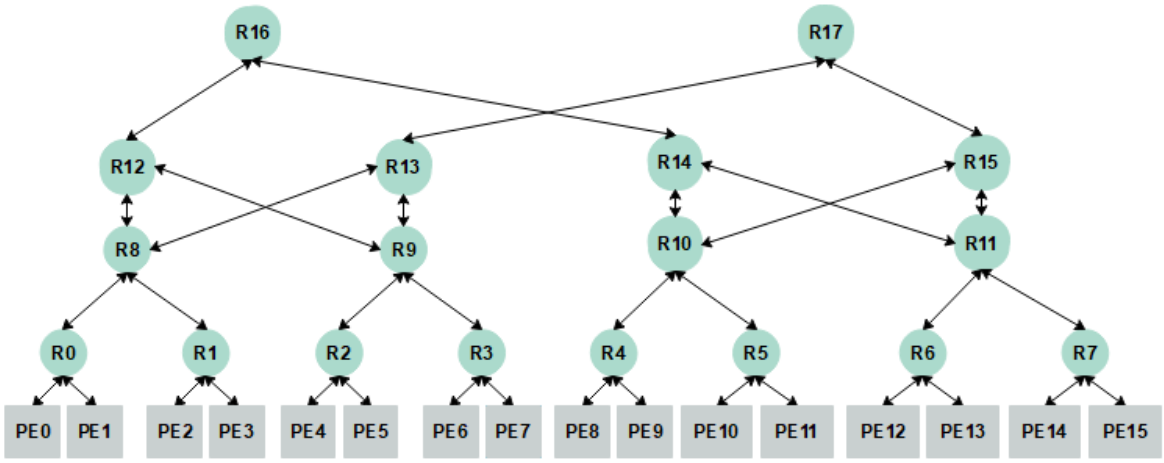


**Figure VI-13.** Fat-tree topology (16-PEs).

In a Fat-tree the PEs are at the leaves of the tree, and the interior nodes are routers that switch traffic between the leaves. An advantage of the Fat-tree topology is that distances are short for local communication patterns. In this topology messages are routed up the tree until a common ancestor is reached and then routed down to the destination; this allows the fat-tree to take advantage of locality between communicating nodes [127], [128]. Figure VI-13 shows a Fat-tree for 16 PEs. R0-R7 are concentrator routers with two PEs connected to each of them.

In Figure VI-14, a plot of average latency vs. throughput for a Fat-tree with sixteen PEs (Figure VI-13) using NoCSimulator and Garnet is presented. Simulation parameters are shown in Table VI-17. In this test, all PEs inject packets at a constant rate, and the latency of the network is recorded after the network has reached steady-state behavior. For each point in the chart, the test is repeated several times to account for stochastic variations, and the latency results are then averaged.

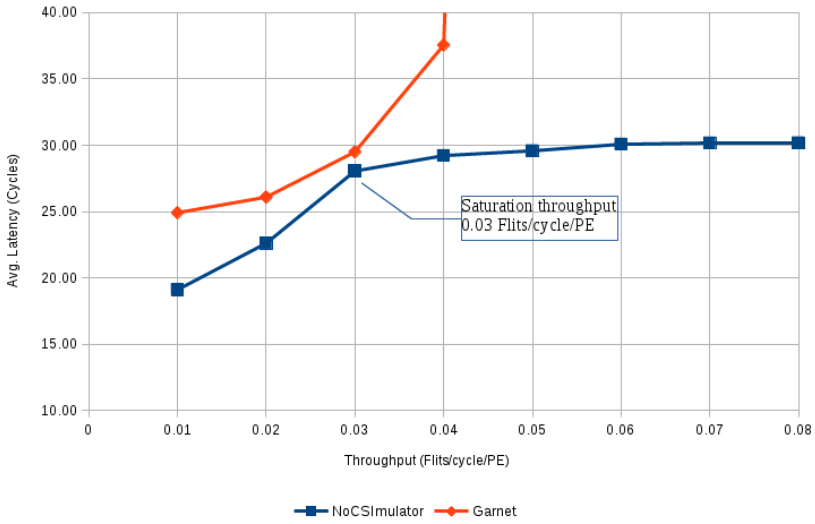


**Figure VI-14.** Average Latency vs Throughput - Fat-tree Topology (16 PEs).

Figure VI-14 shows the low saturation throughput of this topology (0.03 flit/cycle/PE). This is an indication that this topology is not suitable for applications with high data traffic between distant PEs (i.e., many hops between PEs). This result is in agreement with what is reported in the literature about that this topology favors communication between close PEs (for example, PEs connected to the same concentrator) [46].

**Table VI-17.** Simulation Parameters - Fat-tree topology.

| Item | Value |
|---|---|
| Topology | Fat-tree – 16 PEs |
| Routing protocol | Shortest-path[12] |
| Traffic pattern | Uniform[13] |
| Service time at routers (cycles) | 3 |
| Buffer size (flits) | 10 |
| Virtual channels | 1 |
| Packet size (flits) | 1 |

*3) Estimation of performance using statistical traffic patterns of real applications*

In this section, the suitability of NoCSimulator to estimate the performance of an NoC Fat-tree topology with real traffic patterns is shown. As stated earlier, benchmarks from the MCSL traffic suite [28] were used and they are described in Table VI-1. Three NoC sizes are tested: 16-PEs Fat-tree, 32-PEs Fat-tree, and 64-PEs Fat-tree. The packets are composed of eight flits. Figure VI-15 shows the performance results for the metrics network throughput and network throughput per PE. Figure VI-16 shows the performance results for the metrics average latency and execution cycles.

[12] The shortest path between each PE is used to route packets.
[13] Each PE has the same probability to send a packet to the other PEs in the NoC.

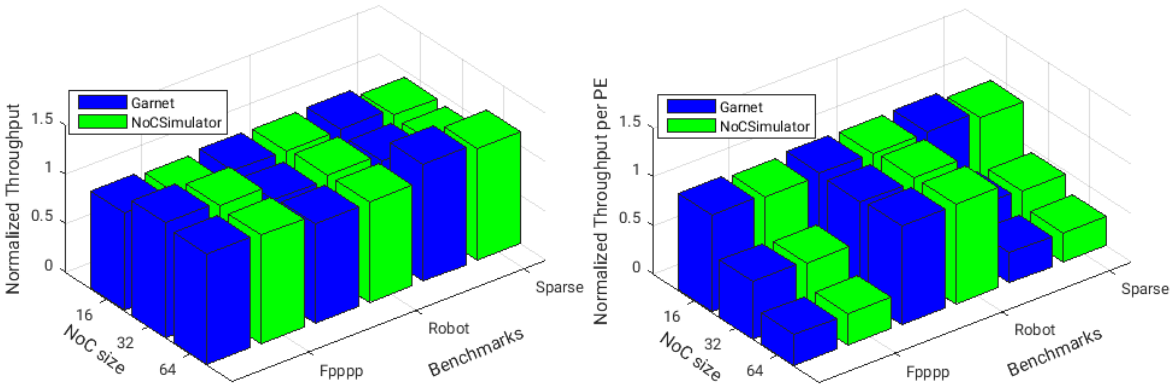


**Figure VI-15.** Performance of applications running in different NoC sizes. Throughput and throughput per PE – Fat-tree topology.

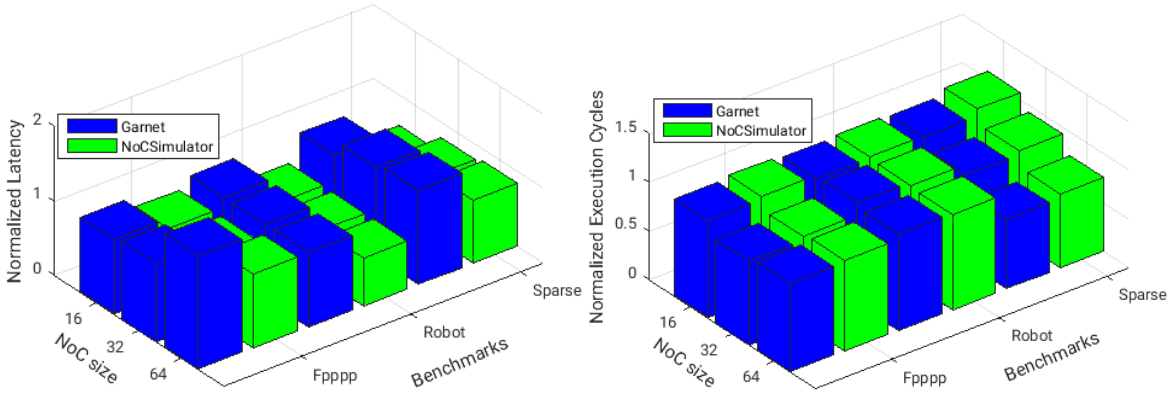


**Figure VI-16.** Performance of applications running on different NoC sizes. Average latency and execution cycles - Fat-tree topology.

The performance metrics for each application simulated in Garnet_STP on the 16-PEs Fat-tree are used to normalize all other performance metrics. Figures 6-15,16 show that the four performance metrics have no strong similarities between the benchmarks, and as stated previously, this indicates that the results on performance are determined not only by network parameters (topology size, routing protocol, etc.) but also by the traffic patterns of the benchmarks used.

In Table VI-21, a comparison of results between NoCSimulator and Garnet_STP is summarized. These results show that NoCSimulator and Garnet_STP are close on all the metrics for all the cases analyzed, except in the latency results (Figure VI-16). This sub-estimation in latency can be attributed mainly to the unbounded latency estimations that Garnet does, as discussed in section 1).

*a) Hypothesis tests for Fat-tree topology*

**Table VI-18.** Performance metrics for Fppp application - Fat Tree topology

| **Topology** | **Metric** | | | |
|---|---|---|---|---|
| | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| Garnet_STP | 0.202 | 0.013 | 26.236 | 250813 |

| | | | | | |
|---|---|---|---|---|---|
| **FatTree16** | NoCSimulator | 0.197 | 0.012 | 22.559 | 256765 |
| | diff | 0.005 | 3e-4 | 3.677 | -5952 |
| | diff (%) | 2.4 | 2.4 | 14.0 | -2.3 |
| | p-value | 0.020 | 0.020 | 4e-136 | 0.034 |
| **FatTree32** | Garnet_STP | 0.244 | 0.008 | 27.555 | 211061 |
| | NoCSimulator | 0.242 | 0.008 | 23.278 | 212756 |
| | diff | 0.002 | 6e-5 | 4.277 | -1695 |
| | diff (%) | 0.8 | 0.8 | 15.5 | -0.8 |
| | p-value | 0.448 | 0.448 | 1e-121 | 0.463 |
| **FatTree64** | Garnet_STP | 0.240 | 0.004 | 38.272 | 217721 |
| | NoCSimulator | 0.234 | 0.004 | 25.853 | 222326 |
| | diff | 0.005 | 7e-5 | 12.419 | -4614 |
| | diff (%) | 2.1 | 2.1 | 32.4 | -2.1 |
| | p-value | 0.037 | 0.037 | 1e-44 | 0.047 |

**Table VI-19.** Performance metrics for Robot application - Fat Tree topology

| **Topology** | | **Metric** | | | |
|---|---|---|---|---|---|
| | | Avg. Throughput (flits/cycle) | Avg. throughput per PE (flits/cycle/PE) | Avg. latency (cycles) | Avg. execution cycles (cycles) |
| **FatTree16** | Garnet_STP | 0.018 | 0.001 | 22.582 | 220210 |
| | NoCSimulator | 0.017 | 0.001 | 14.339 | 224807 |
| | diff | 4e-4 | 2e-5 | 8.243 | -4597 |
| | diff (%) | 2.3 | 2.3 | 36.5 | -2.1 |
| | p-value | 0.005 | 0.005 | 1e-270 | 0.011 |
| **FatTree32** | Garnet_STP | 0.018 | 0.001 | 22.581 | 220156 |
| | NoCSimulator | 0.017 | 0.001 | 14.341 | 224644 |
| | diff | 3e-4 | 1e-5 | 8.240 | -4488 |
| | diff (%) | 2.2 | 2.2 | 36.5 | -2.0 |
| | p-value | 0.002 | 0.002 | 9e-271 | 0.006 |
| **FatTree64** | Garnet_STP | 0.018 | 2.7e-4 | 22.580 | 220120 |
| | NoCSimulator | 0.017 | 2.7e-4 | 14.369 | 223693 |
| | diff | 2e-4 | 4e-6 | 8.211 | -3573 |
| | diff (%) | 1.7 | 1.7 | 36.4 | -1.6 |
| | p-value | 0.021 | 0.021 | 5e-277 | 0.027 |

**Table VI-20.** Performance metrics for Sparse application - Fat-Tree topology

| **Topology** | **Metric** | | | |
|---|---|---|---|---|
| | Avg. Throughput (flits/cycle) | Avg. throughput per PE | Avg. latency | Avg. execution cycles |

| | | | (flits/cycle/PE) | (cycles) | (cycles) |
|---|---|---|---|---|---|
| **FatTree16** | Garnet_STP | 0.113 | 0.007 | 22.564 | 69935 |
| | NoCSimulator | 0.104 | 0.006 | 13.327 | 76274 |
| | diff | 0.009 | 5e-4 | 9.237 | -6339 |
| | diff (%) | 7.9 | 7.9 | 40.9 | -9.1 |
| | p-value | 1e-25 | 1e-25 | 4e-270 | 2e-27 |
| **FatTree32** | Garnet_STP | 0.107 | 0.003 | 26.934 | 62241 |
| | NoCSimulator | 0.101 | 0.003 | 17.980 | 65567 |
| | diff | 0.006 | 1e-4 | 8.955 | -3326 |
| | diff (%) | 5.3 | 5.3 | 33.2 | -5.3 |
| | p-value | 8e-12 | 8e-12 | 2e-247 | 8e-11 |
| **FatTree64** | Garnet_STP | 0.133 | 0.002 | 28.436 | 50175 |
| | NoCSimulator | 0.124 | 0.002 | 19.066 | 53331 |
| | diff | 0.009 | 1e-4 | 9.419 | -3157 |
| | diff (%) | 6.5 | 6.5 | 33.1 | -6.3 |
| | p-value | 1e-13 | 1e-13 | 3e-257 | 9e-11 |

**Table VI-21.** Performance summary - Fat-tree topology

| **Attribute** | **NoCSimulator vs Garnet_STP** | |
|---|---|---|
| | **Avg. diff.** | **Max. diff** |
| Throughput | 3.5% | 7.9% |
| Throughput per PE | 3.5% | 7.9% |
| Latency | 30.9% | 40.9% |
| Execution cycles | -3.5% | -9.1% |

## F. *NoCSimulator simulation times*

From the study of the state of the art made in section 7), it is expected that NoC performance estimation tools based on formal models be faster than cycle-accurate ones. This is attributed mainly to the use of mathematical models to estimate the performance of an NoC. These models, being high-level representations of the NoC are faster to execute than detailed models of the hardware of an NoC present in cycle-accurate simulators. In this section, the simulation times of NoCSimulator when running the benchmarks used in sections 6.1 to 6.4 are compared to the ones obtained when using Garnet_STP.

### *1) Mesh topology*

For each topology size and benchmark, the average simulation time of NoCSimulator is normalized using the average simulation time obtained with Garnet_STP. Figure VI-17 shows the results.

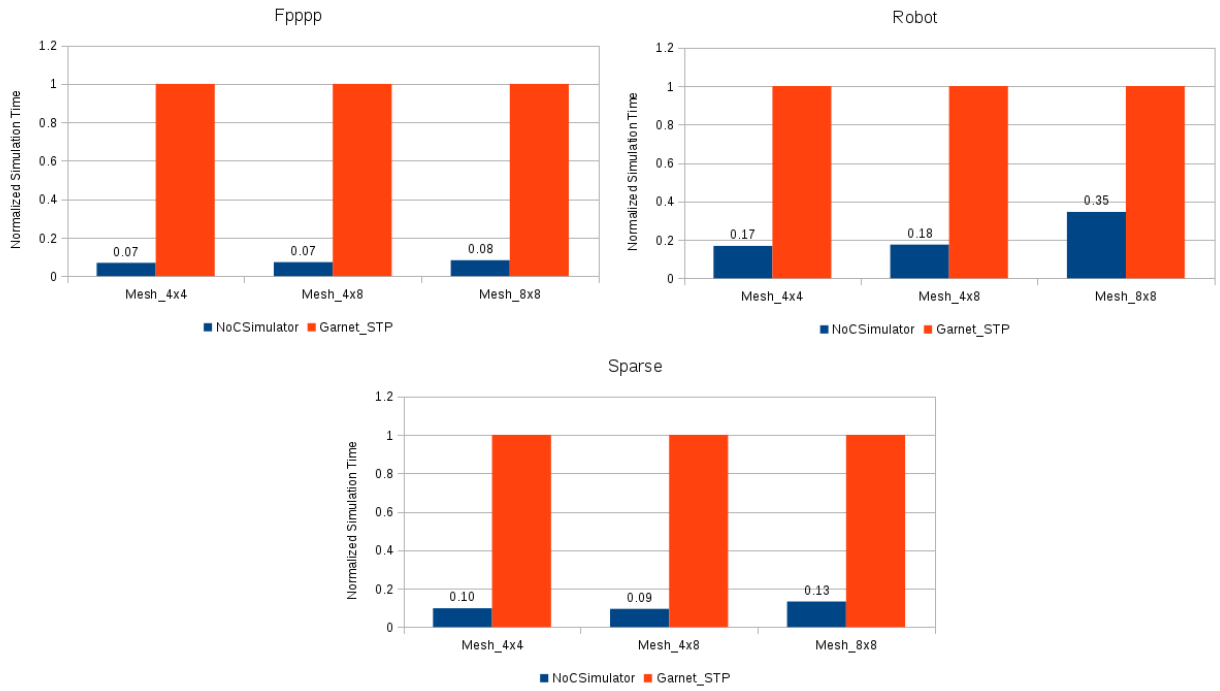


**Figure VI-17.** Normalized simulation time for each benchmark running in a Mesh topology.

From Figure VI-17 it can be seen that NoCSimulator is faster than Garnet_STP for all benchmarks and Mesh sizes, with simulation times that are on average about 13% of the ones obtained when using Garnet_STP. The different simulation times obtained across Mesh sizes and benchmarks is an indication of the effect that the TCGs data (mapping of tasks, data packets, tasks execution times, etc.) has on the simulation time of both tools. The diminished gains in simulation time for the Robot application can be an indication that this application is simpler to model by both NoCSimulator and Garnet_STP, and thus, benefits less for being modeled in NoCSimulator.

*2) Ring topology*

For Ring topology, the average simulation time of NoCSimulator is normalized using the average simulation time of Garnet_STP for each Ring size and benchmark. Figure VI-18 presents the results. From Figure VI-18, it can be seen that NoCSimulator is faster than Garnet_STP for all benchmarks and Ring Sizes. On average, the simulation time of NoCSimulator is 7% of the one obtained when using Garnet_STP. As in section 1), it seems that the TCGs data influences the simulation time of the tools. i.e., for more complex TCGs (Fpppp and Sparse) the differences in simulation time between NoCSimulator and Garnet are more evident.

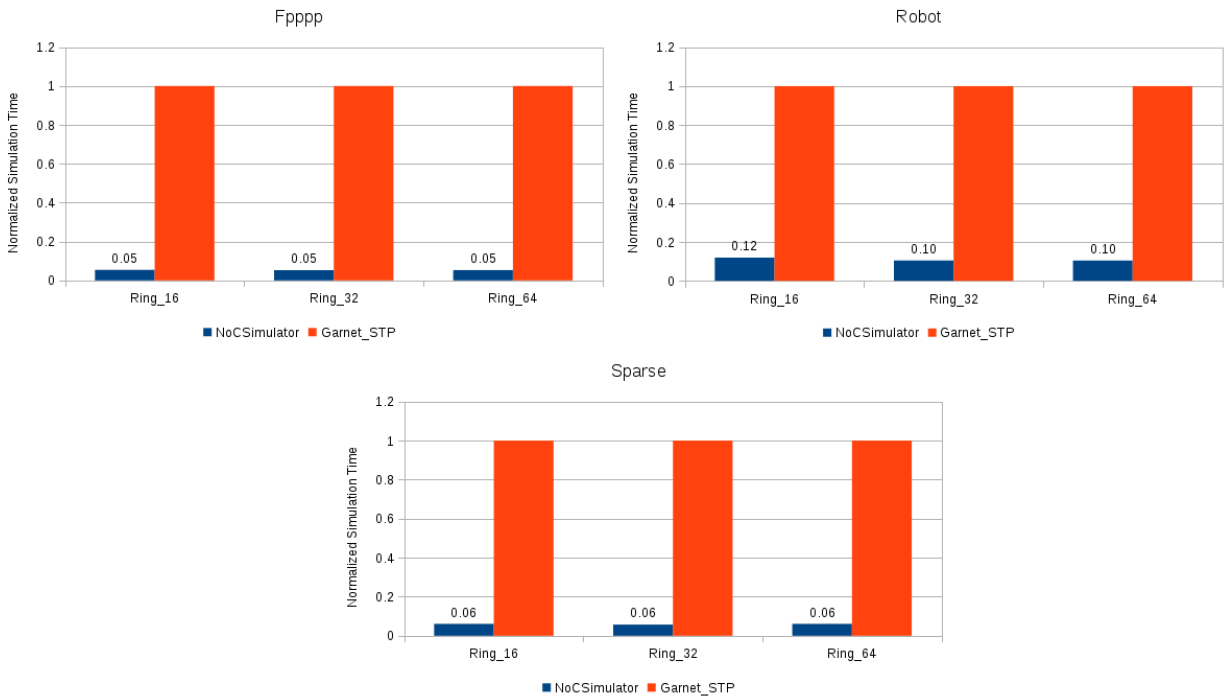


**Figure VI-18.** Normalized simulation time for each benchmark running in Ring topology.

### *3) Flattened Butterfly topology*

The average simulation time of NoCSimulator for each NoC size and benchmark is normalized using the simulation time of Garnet_STP. Results are shown in Figure VI-19, for all benchmarks and NoC sizes NoCSimulator is faster than Garnet_STP. On average, the simulation time for NoCSimulator is about 16% of the simulation time of Garnet_STP. As indicated in the above sections, it is evident that for the Fpppp and Sparse benchmarks the gap between the simulation times of NoCSimulator and Garnet_STP is wider than for the Robot benchmark. This suggests that these benchmarks (Fpppp and Sparse) are more complex to handle by Garnet_STP and benefit for its modeling using a high-level tool as NoCSimulator.

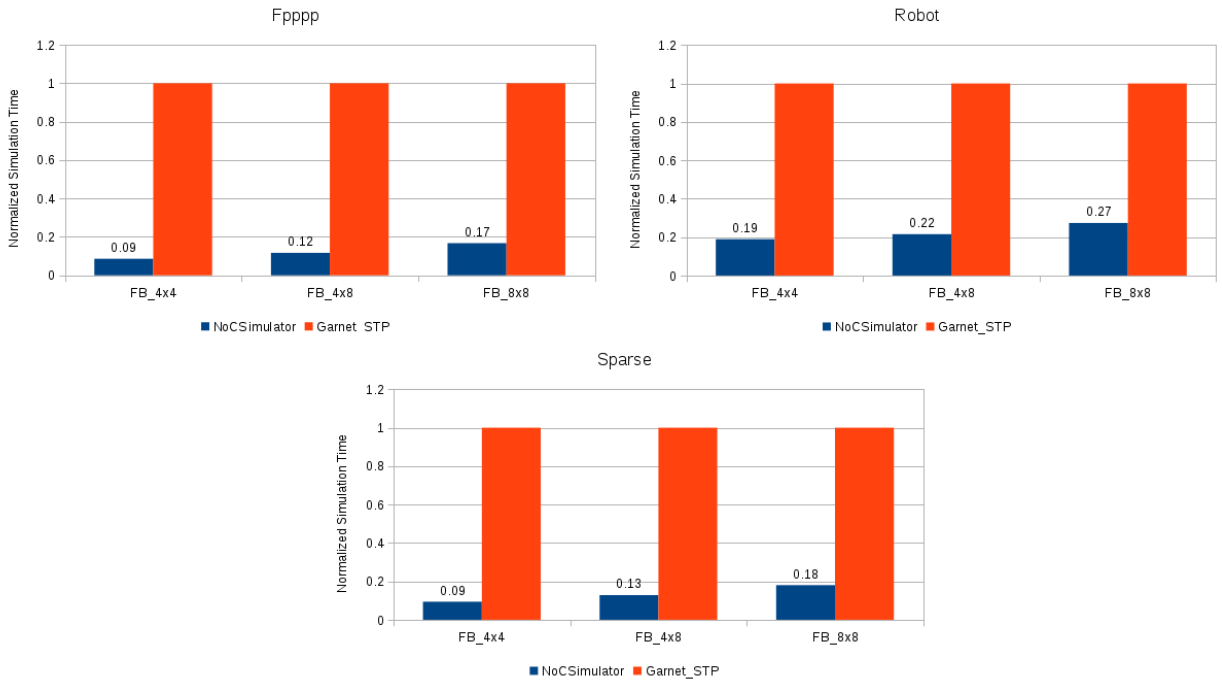


**Figure VI-19.** Normalized simulation time for each benchmark running in a Flattened Butterfly (FB) topology.

### *4) Fat-tree topology*

For this topology, the average simulation time of NoCSimulator is normalized by the one obtained using Garnet_STP for all NoC sizes and benchmarks. The results are shown in Figure VI-20. For all tests, NoCSimulator outperforms Garnet_STP, with an average simulation time of about 10% of the simulation time of Garnet_STP. As described in the previous sections, the gap between NoCSimulator and Garnet_STP is bigger for the Fpppp and Sparse benchmarks than for the Robot benchmark, this can be an indication that for complex TCGs the simulation time benefits from modeling these TCGs in a high-level tool as NoCSimulator.

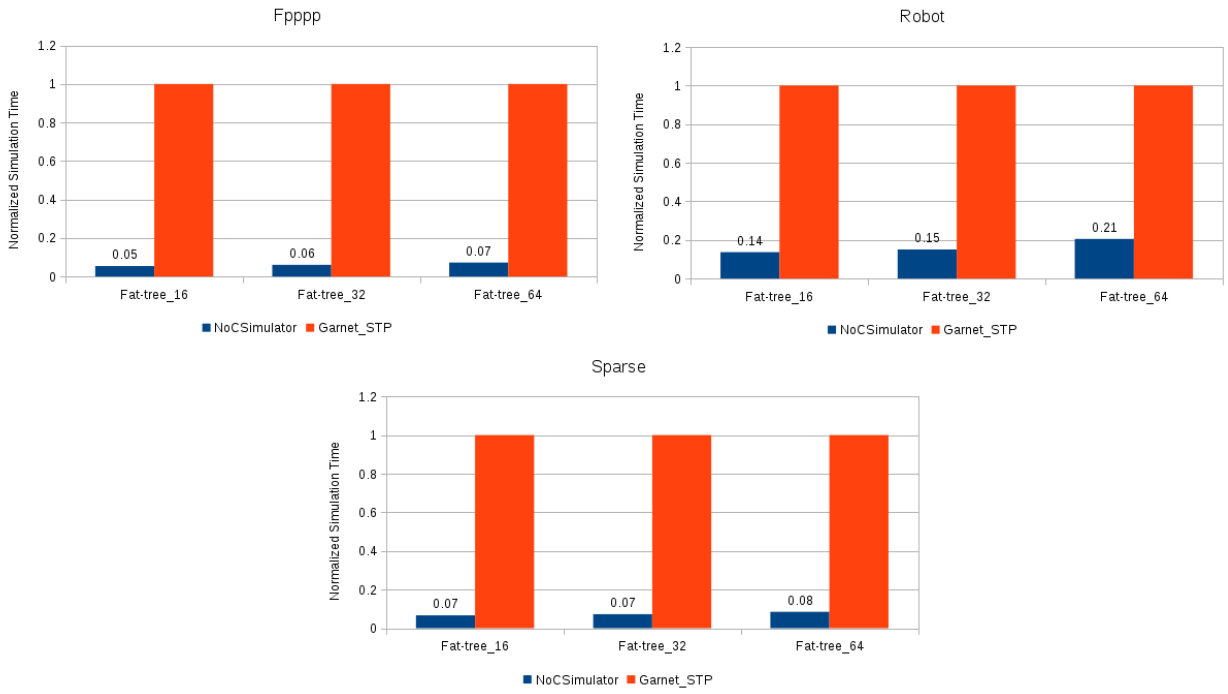


**Figure VI-20.** Normalized simulation time for each benchmark running in a Fat-tree topology.

## G. *CONCLUSION*

In this chapter, validation tests for NoCSimulator were performed. The tests included the use of synthetic traffic and traffic patterns based on real applications, the latter thanks to the integration of the MCSL traffic suite into NoCSimulator. The gem5 cycle-accurate simulator was used as validation tool. After analyzing the results of NoCSimulator for different applications from the MCSL traffic suite, mapped in several topologies of varied sizes, it was found that NoCSimulator can accurately estimate the performance metrics of the different NoCs, except for latency. The statistical significance of the differences in performance estimation between NocSimulator and gem5 (Garnet_STP), were tested using the hypothesis testing method, finding that for many tests the differences observed had statistical significance, yet they were small enough to be usable. For other tests it was observed that these differences had no statistical significance due to the closeness of the estimations. As discussed in sections 6.1 to 6.4 the sub-estimation of latency can be mainly attributed to the unbound growth of latency that Garnet_STP estimates when the throughput approaches saturation rates. This, due to how Garnet handles packet contention at cache level. However, at early stages of the design process where several topologies and application mappings must be tested, this sub-estimation of latency does not prohibit the obtention of insights about NoC performance, because the tendencies in the estimation of latency between Garnet_STP and NoCSimulator are preserved. Then, after prospect system configurations are selected, cycle-accurate simulations can be performed to obtained more refined results. Thus, helping in lowering the time spent in the Design Space Exploration (DSE) process. Also, as shown in section F, the simulation time of NoCSimulator is lower than the Garnet_STP as expected from the review of the state of the art in section 7). All of above demonstrates the features of NoCSimulator and its suitability for NoC performance estimation. In the next chapter, NoCSimulator is used to do a simple DSE experiment, showing how NoCSimulator can be embedded in the DSE workflow. With this chapter, specific objective *e* (see section b)) is met.

# VII. DESIGN SPACE EXPLORATION EXERCISE

**CONTENTS**



## A. *Introduction*

In VI it was found that NoCSimulator was capable of estimate NoC Performance with high accuracy when compared to the gem5 cycle-accurate simulator (except avg. latency, but the reasons for this are explained in VI). Now, having confidence in the capabilities of NoCSimulator, in this chapter, it is used to do a simple Design Space Exploration (DSE) exercise (see section E for a discussion about DSE). The goal is to use results from NoCSimulator to identify a suitable design point that maximizes NoC performance for each of the applications studied in VI. This DSE exercise is man supervised (i.e., no automated methods as the ones discussed in section F are used). Also, in section F, it was shown that NoCSimulator is considerable faster than the cycle-accurate simulator gem5. This makes it ideal to be used during the DSE process, especially at early design stages.

This chapter is divided as follows: section B explains how the DSE exercise is going to be carried out. Later, in section C the Fpppp application is studied, section D is for the Robot application, and finally, section E is devoted to the Sparse application. With this chapter, specific objective *f* (see section b)) is fulfilled.

B. *Experimental setup*

A DSE exercise consist of finding a design point in the vast space of design options (for example, NoC topologies, NoC size, application mapping, etc.) that meets specific design constraints (like power consumption, execution time, occupied area, etc.). For the experiments performed in this chapter, the goal is to find a NoC topology that offers maximum throughput, while minimizing latency and execution cycles. Additional criteria for the selection process is that an increasing number of PEs (i.e., bigger NoC sizes) generally imply an increase in cost, area occupied, and power consumption of the system. Thus, each application (Fpppp, Robot and Sparse) is mapped in each topology (Mesh, Ring, Flattened Butterfly and Fat-tree) and each NoC size (16, 32 and 64 PEs). 100 samples of each configuration are taken to account for statistical variations. Later, comparing the performance of each application for each topology and NoC size, the configuration that meets the design constrains is selected as the best configuration for the given application.

C. *Fpppp application*

Fpppp application is mapped to Mesh, Ring, Flattened Butterfly (FB), and Fat-tree topologies of 16, 32, and 64 PEs. In Figure VII-1 performance metrics of throughput and throughput per PE are presented. These metrics are normalized to those obtained for a Mesh of 16 PEs. In Figure VII-1-A the tendency is that throughput increases with network size except for Fat-tree topology. The maximum increase of throughput for each topology is shown in Figure VII-1-A. Thus, the topology that offers maximum throughput for the Fpppp traffic pattern is a Ring of 64 PEs, followed closely by a Flattened Butterfly Topology of 64 PEs. In Figure VII-1-B the tendency is that throughput per PE decreases with network size, with the maximum decrease in Fat-tree topology of 64 PEs.

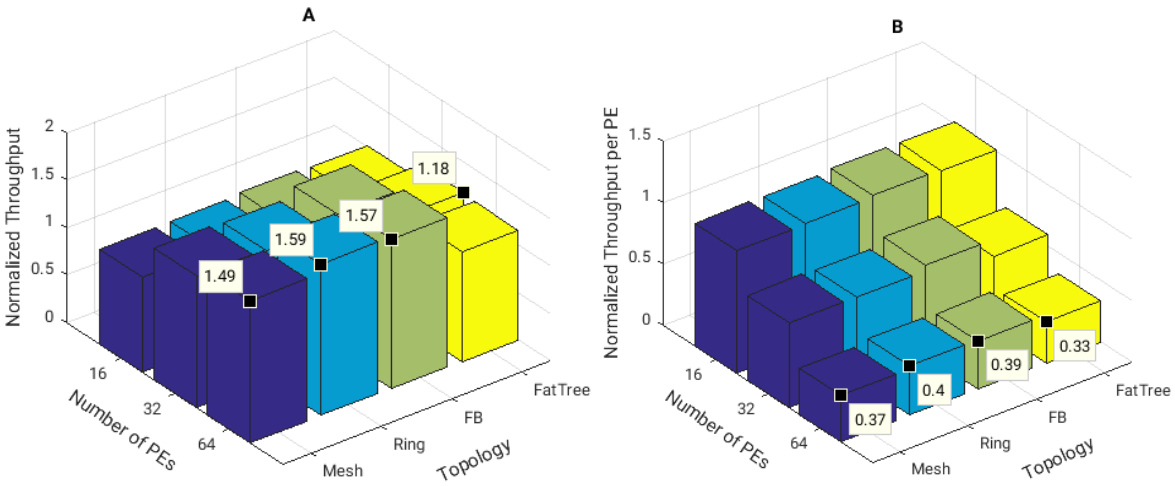


**Figure VII-1.** Performance of Fpppp application mapped in different topologies. **A)** Normalized throughput **B)** Normalized throughput per PE.

In Figure VII-2 latency and execution cycles are presented. These metrics are normalized to those of Mesh of 16 PEs. In Figure VII-2-A, latency increases with network size except for Flattened Butterfly topology. As expected, Ring topology has the maximum latency of all topologies, with the Ring of 64 PEs having more than four times the latency of a Mesh of 16 PEs. Also, Fat-tree topology has almost 2 times the latency of a Mesh of 16 PEs, independently of the network size. Thus, the best topology in terms of latency is

Flattened Butterfly. In Figure VII-2-B, execution cycles decrease with network size, except for Fat-tree topology. The maximum decrease in execution cycles is for Ring topology of 64 PEs, followed closely by Flattened Butterfly of 64 PEs.

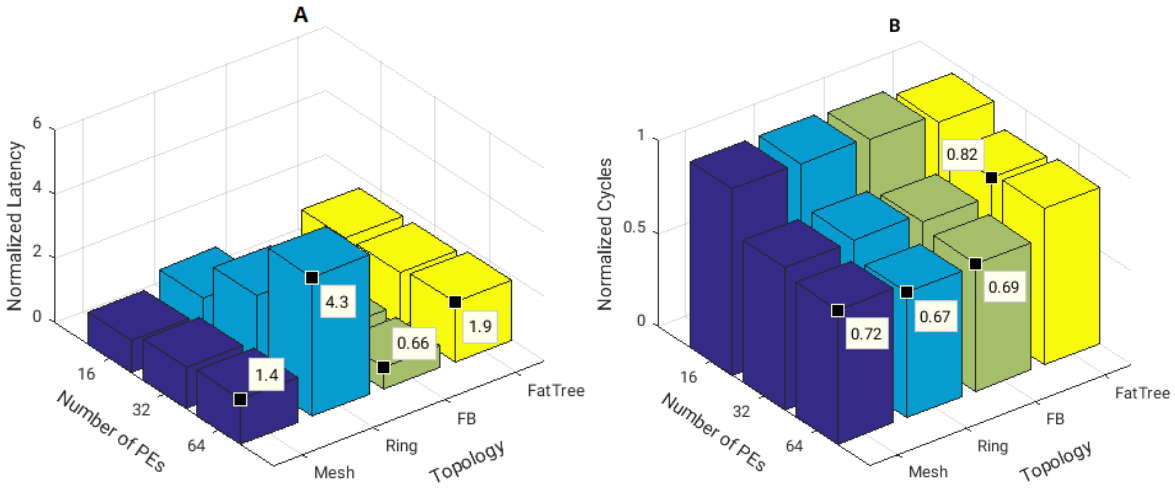


**Figure VII-2.** Performance of Fpppp benchmark mapped in different topologies. **A)** Normalized latency **B)** Normalized Cycles.

With the information presented in Figure VII-1 and Figure VII-2, it is possible to select the best topology for the Fpppp application. The objective is to maximize throughput, minimizing latency and execution cycles. Also, for the selection process, it must be noted that more PEs generally means more costs, more occupied area, and more power consumption [67]. Thus, from figures 7-1,2, a feasible topology is Flattened Butterfly of 32 PEs, because its throughput is 1.5 times higher, with a throughput per PE near 0.7 times lower, a latency also 0.7 times lower, and execution cycles 0.7 lower than for a Mesh topology of 16 PEs.

D. *Robot application*

Robot application is mapped to Mesh, Ring, Flattened Butterfly (FB), and Fat-tree topologies of 16, 32, and 64 PEs. In Figure VII-3 performance metrics of throughput and throughput per PE are presented. These metrics are normalized to those obtained for a Mesh of 16 PEs. In Figure VII-3-A throughput remains fairly constant regardless of NoC topology and size, except for Fat-tree topology. The maximum throughput for each topology is displayed in Figure VII-3-A. Thus, the topology that offers maximum throughput for the Robot application is a Mesh of 16 PEs. In Figure VII-3-B, the tendency is that throughput per PE decreases with network size, except for Fat-Tree topology. However, this decrease in throughput per PE is negligible for networks of 64 PEs and is almost the same across topologies. The maximum decrease in throughput per PE is found in Mesh of 32 PEs and Flattened Butterfly of 64 PEs.

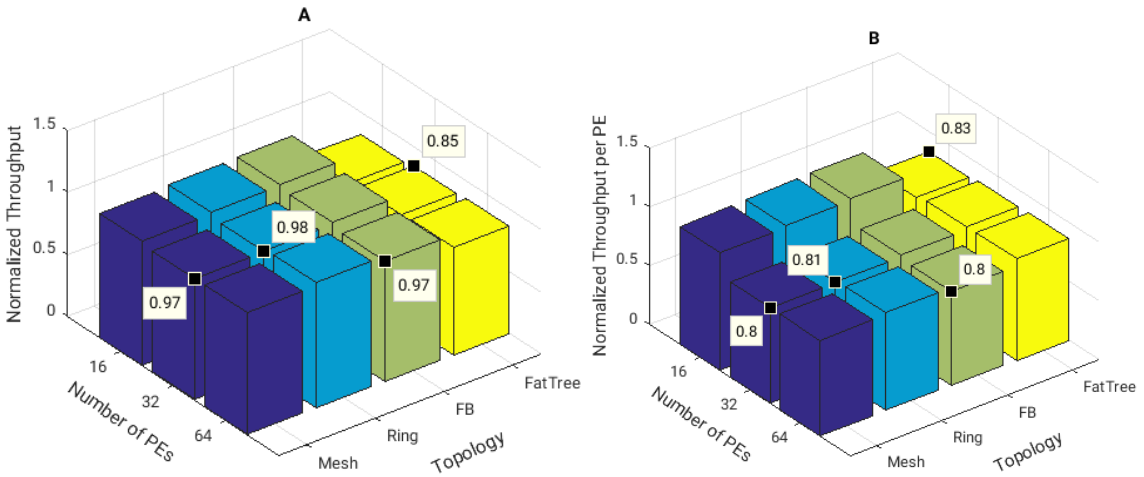


**Figure VII-3.** Performance of Robot application mapped in different topologies. **A)** Normalized throughput **B)** Normalized throughput per PE.

In Figure VII-4 latency and execution cycles are presented. These metrics are normalized to those of Mesh of 16 PEs. In Figure VII-4-A, latency is nearly constant for each topology except for Ring topology, with an increased latency as network size increases. The best latency behavior is for Flattened Butterfly with a maximum latency 0.75 times lower than that of Mesh. The highest latency is for Ring topology of 64 PEs, which is almost three times higher than for Mesh topology. Fat-tree topology has a latency almost 1.5 times higher than for Mesh topology regardless of network size. In Figure VII-4-B, execution cycles remain almost constant for all topologies and network sizes.

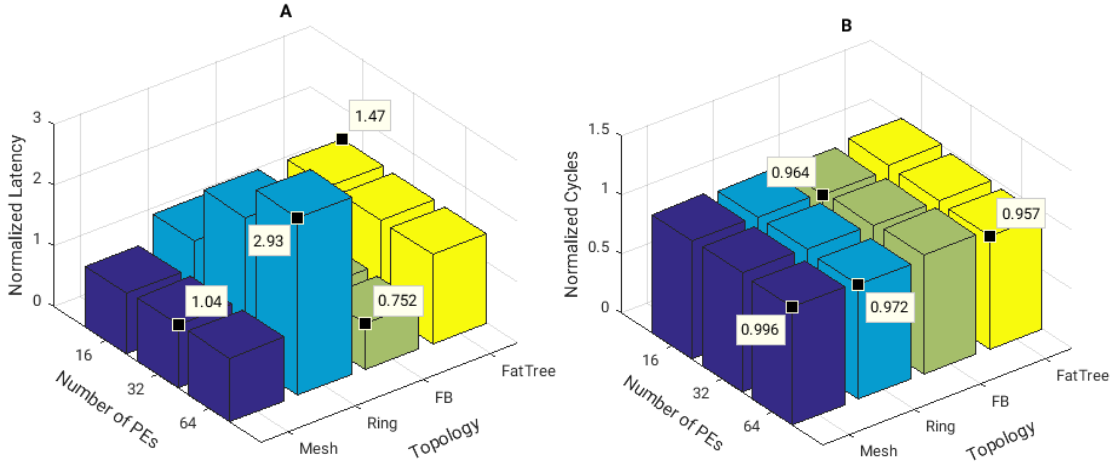


**Figure VII-4.** Performance of Robot application mapped in different topologies. **A)** Normalized latency **B)** Normalized Cycles.

With the information presented in Figure VII-3 and Figure VII-4, it is possible to select the best topology for the Robot application. The objective is to maximize throughput, minimizing latency and execution cycles. Also, for the selection process, it must be noted that more PEs generally means more costs, more occupied area, and more power consumption [67]. Thus, from figures 7-3,4 a feasible topology is Mesh of 16 PEs since its throughput is marginally higher than the other topologies and network sizes; its latency is lower than other topologies except for Flattened Butterfly (which in theory is more expensive to implement due to the use of routers with bypass ports [47]), and its execution cycles are almost the same than the other topologies and network sizes.

E. *Sparse application*

Sparse application is mapped to Mesh, Ring, Flattened Butterfly (FB), and Fat-tree topologies of 16, 32, and 64 PEs. In Figure VII-5 performance metrics of throughput and throughput per PE are presented. These metrics are normalized to those obtained for a Mesh of 16 PEs. In Figure VII-5-A the tendency is that throughput is maximum for all topologies of 32 PEs except for Fat-tree topology, which offers less throughput for all network sizes. Maximum throughput for each topology is displayed in Figure VII-5-A. As can be seen, a Mesh of 32 PEs (4x8) offers the maximum throughput, followed closely by Ring and Flattened Butterfly of 32 PEs. In Figure VII-5-B the tendency is that throughput per PE decreases with network size, with the maximum decrease in Fat-tree topology of 64 PEs.

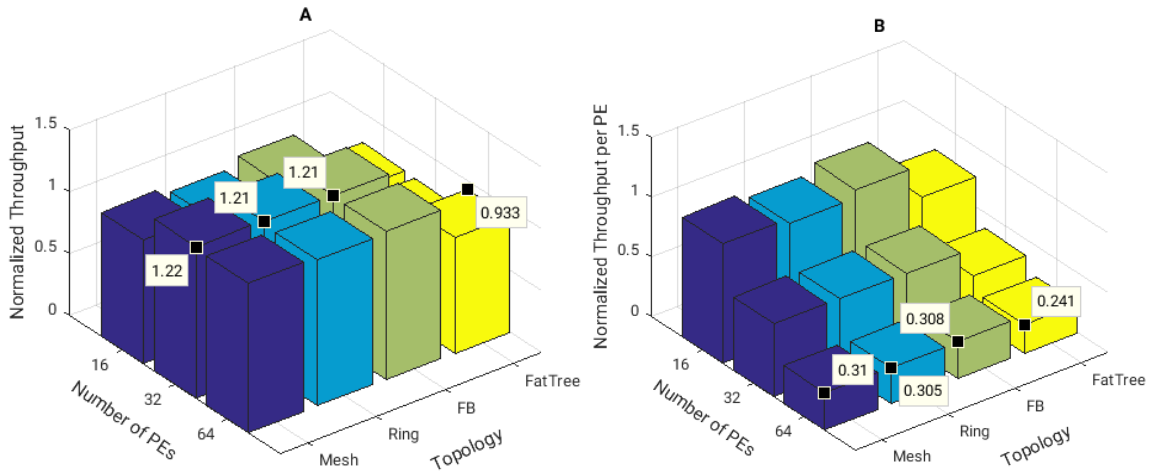


**Figure VII-5.** Performance of Sparse application mapped in different topologies. **A)** Normalized throughput **B)** Normalized throughput per PE.

In Figure VII-6 latency and execution cycles results for the Sparse application are presented. These metrics are normalized to those of Mesh of 16 PEs. In Figure VII-6-A, latency increases with network size except for Flattened Butterfly topology. As expected, Ring topology has the maximum latency of all topologies, with the Ring of 64 PEs having more than four times the latency of a Mesh of 16 PEs. Also, the Fat-tree topology of 32 and 64 PEs has almost 2 times the latency of a Mesh of 16 PEs. Thus, the best topology in terms of latency is Flattened Butterfly with similar performance for all network sizes. In Figure VII-6-B, execution cycles decrease with network size. The decrease in execution cycles is almost identical for all topologies with 64 PEs.

With the information presented in Figure VII-5 and Figure VII-6, it is possible to select the best topology for the Sparse application. The objective is to maximize throughput, minimizing latency and execution cycles. Also, for the selection process, it must be noted that more PEs generally means more costs, more occupied area, and more power consumption [67]. Thus, from figures 7-5,6 a feasible topology is Flattened Butterfly of 32 PEs because its throughput is almost the same as the other topologies of 32 PEs; its latency

is much lower than other topologies, and its execution cycles are very similar to the other topologies of the same size.

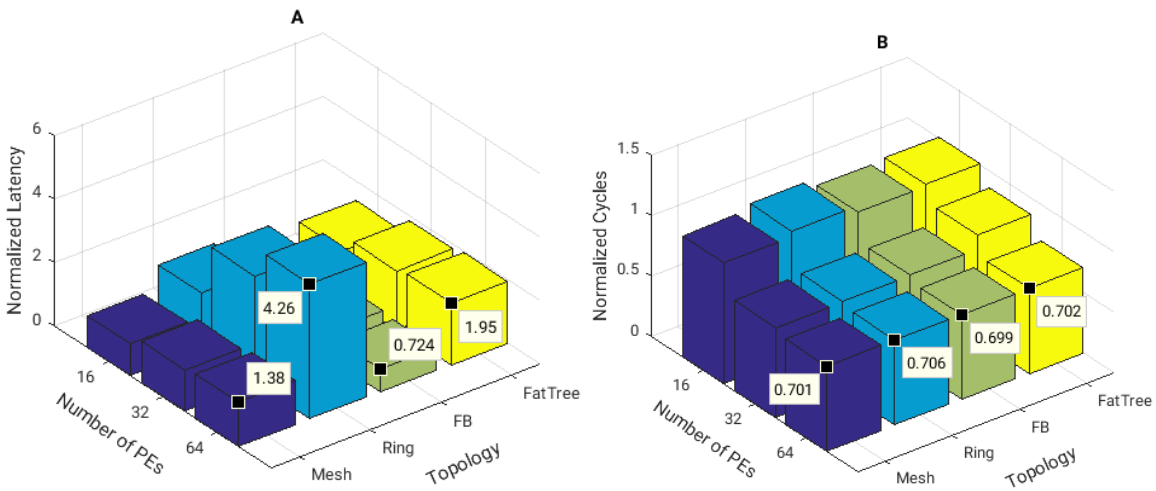


**Figure VII-6.** Performance of Sparse application mapped in different topologies. A) Normalized latency B) Normalized Cycles.

Table VII-1 presents a summary of the results obtained during the DSE exercise.

**Table VII-1**. Summary results of the DSE exercise.

| Application | Selected topology | Performance metrics (normalized to Mesh_16) |
|---|---|---|
| **Fpppp** | **FB_32** | Throughput: 1.5 |
| | | Throughput_per_PE: 0.7 |
| | | Latency: 0.7 |
| | | Execution cycles: 0.7 |
| **Robot** | **Mesh_16** | Throughput: 1.0 |
| | | Throughput_per_PE: 1.0 |
| | | Latency: 1.0 |
| | | Execution cycles: 1.0 |
| **Sparse** | **FB_32** | Throughput: 1.2 |
| | | Throughput_per_PE: 0.5 |
| | | Latency: 0.7 |
| | | Execution cycles: 0.7 |

## F. *CONCLUSION*

In this chapter, results from NoCSimulator have been used to do Design Space Exploration for three different applications. Each application (Fpppp, Robot and Sparse) was mapped to Mesh, Ring, Flattened Butterfly and Fat-tree topologies of several sizes (16, 32 and 64 PEs). The objective of these experiments was to find a topology that maximized throughput, while minimizing latency and execution cycles for each application. From these tests it was found that for the Robot application the topology that best met the design constrains was Mesh 4x4 (16 PEs); and for both the Fpppp and Sparse applications the best topology was Flattened Butterfly 4x8 (32 PEs). This chapter illustrates how NoCSimulator can be used in early stages of the DSE process, where multiple topologies must be investigated to find prospect system configurations

that meet a set of design constrains. Having in mind that NoCSimulator runs considerable faster than a cycle-accurate simulator as shown section F, the DSE process can be speedup considerably. With this chapter, specific objective *f* (section b)) is met.

# VIII. CONCLUSIONS AND FUTURE WORK

In this thesis, the path leading to the development of the NoCSimulator was described. From the appropriation of NoC concepts done in Chapter 2, the discussion that led to the selection of Queueing Theory as the formal model to use in NoCSimulator in Chapter 3, the selection of gem5 as the validation tool in Chapter 4, and finally the development of NoCSimulator as described in Chapter 5.

NoCSimlator is a NoC performance estimation tool based on Queueing Theory. NoCSimulator supports four NoC topologies, Mesh, Ring, Flattened Butterfly, and Fat-tree. When compared with the gem5 cycle-accurate simulator, it was shown that NoCSimulator can estimate accurately the throughput, throughput per PE, and execution cycles for traffic patterns based on real applications. However, it was found a considerable discrepancy in latency estimations. As discussed in VI, this discrepancy can be attributed to how gem5 (specifically, the garnet module) models packet contention at the cache memories. NoCSimulator can work with traffic patterns based on real applications, thanks to the MCSL traffic suite, that offers high-level representations of the traffic patterns of several applications mapped on several NoC topologies. The ability to model several topologies, and work with traffic patterns based on real applications sets apart NoCSimulator from other tools based on Queueing Theory reported in the literature, that can only handle synthetic traffic patterns and only support one topology (mainly Mesh). Also, it was shown that NoCSimulator is faster than the gem5 cycle-accurate simulator when modeling traffic patterns of real applications, something that makes it attractive for NoC performance estimation at early stages of the DSE process. From the work presented in this document, the general objective of this thesis (see section a)) is met.

## A. *Future work*

After presenting NoCSimulator, a future research path can be how it can be integrated into automatic DSE tools, like the ones presented in section F. This to take advantage of its faster execution when compared with a cycle-accurate simulator. Also, performance estimation results obtained from NoCSimulator, can be used to estimate other performance metrics, like power consumption, cost, reliability, etc. This can be done, integrating NoCSimulator with other tools available in the literature like ORION [134], [135], and [136], [137]. Additionally, the modular design of NoCSimulator permits its evolution. For example, new topologies or routing protocols can be incorporated.

# REFERENCES


[1] N. E. Jerger, T. Krishna, and L. Peh, "Chapter 1 - Introduction," in *On-chip Networks*, 2nd ed., Morgan & Claypool, 2017, pp. 1–7.

[2] W. J. Dally and B. Towles, "Chapter 1 - Introduction to Interconnection Networks," in *Principles and Practices of Interconnection Networks*, 1st ed., Morgan Kaufmann, 2004, pp. 1–25.

[3] T. Kempf, G. Ascheid, R. Leupers, T. Kempf, G. Ascheid, and R. Leupers, "Chapter 3 - Principles of Design Space Exploration," in *Multiprocessor Systems on Chip*, 1st ed., Springer Science+Business Media, 2011, p. 23.

[4] P. Kansakar and A. Munir, "A Design Space Exploration Methodology for Parameter Optimization in Multicore Processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9219, no. c, pp. 1–14, 2017, doi: 10.1109/TPDS.2017.2745580.

[5] Y. Paik, M. Han, K. H. Choi, M. Kim, and S. W. Kim, "Cycle-Accurate full system simulation for CPU+GPU+HBM computing platform," *Int. Conf. Electron. Inf. Commun. ICEIC 2018*, vol. 2018-Janua, pp. 1–2, 2018, doi: 10.23919/ELINFOCOM.2018.8330603.

[6] M. Z. Ataie and O. Elahi, "Analysis of a Parallel/Distributed Application Using a Cycle-Accurate Parallel/Distributed Simulator," *26th Iran. Conf. Electr. Eng. ICEE 2018*, pp. 1523–1529, 2018, doi: 10.1109/ICEE.2018.8472447.

[7] M. Kitou, T. Sasaki, and K. Ohno, "Performance evaluation of dynamic cell allocation cache using cycle accurate simulator," *Proc. - 2018 6th Int. Symp. Comput. Netw. Work. CANDARW 2018*, pp. 555–557, 2018, doi: 10.1109/CANDARW.2018.00109.

[8] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, doi: 10.1145/2024716.2024718.

[9] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," 2011, doi: 10.1145/2063384.2063454.

[10] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," *ISPASS 2007 IEEE Int. Symp. Perform. Anal. Syst. Softw.*, pp. 23–34, 2007, doi: 10.1109/ISPASS.2007.363733.

[11] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez, "Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors," in *19th International Symposium on Computer Architecture and High Performance Computing*, 2007, pp. 62–68, doi: 10.1109/sbac-pad.2007.17.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *2001 IEEE Int. Work. Workload Charact. WWC 2001*, pp. 3–14, 2001, doi: 10.1109/WWC.2001.990739.

[13] Standard Performance Evaluation Corporation, "SPEC CPU™ 2006," 2006. https://www.spec.org/cpu2006/ (accessed Aug. 20, 2020).

[14] A. Akram and L. Sawalha, "×86 computer architecture simulators: A comparative study," *Proc. 34th IEEE Int. Conf. Comput. Des. ICCD 2016*, pp. 638–645, 2016, doi: 10.1109/ICCD.2016.7753351.

[15] J. J. Murillo, "HW-SW components for parallel embedded computing on Noc-based MPSoCs," Univeristat Autòma de Barcelona, 2009.

[16] M. Palesi, V. Catania, S. Monteleone, D. Patti, and A. Mineo, “Cycle-Accurate Network on Chip Simulation with Noxim,” *ACM Trans. Model. Comput. Simul.*, vol. 27, no. 1, pp. 1–25, 2016, doi: 10.1145/2953878.

[17] N. Jiang *et al.*, “A detailed and flexible cycle-accurate Network-on-Chip simulator,” *ISPASS 2013 - IEEE Int. Symp. Perform. Anal. Syst. Softw.*, pp. 86–96, 2013, doi: 10.1109/ISPASS.2013.6557149.

[18] T. Krishna, “Garnet 2.0: A Detailed On-Chip Network Model Inside a Full-System Simulator,” in *gem5 workshop ARM Research Summit*, 2017, pp. 33–42.

[19] M. Khamis and S. El-Ashry, “A Configurable RISC-V for NoC-Based MPSoCs: A Framework for Hardware Emulation,” 2018.

[20] M. Gries, “Methods for evaluating and covering the design space during early design development,” *Integr. VLSI J.*, vol. 38, no. 2, pp. 131–183, 2004, doi: 10.1016/S0167-9260(04)00032-X.

[21] A. B. Kahng, B. Lin, K. Samadi, and R. Sunkam Ramanujam, “Efficient trace-driven metaheuristics for optimization of networks-on-chip configurations,” *IEEE/ACM Int. Conf. Comput. Des. Dig. Tech. Pap. ICCAD*, pp. 256–263, 2010, doi: 10.1109/ICCAD.2010.5654164.

[22] Y. C. Chang, Y. S. C. Huang, T. C. Tsai, Y. Y. Chang, C. T. King, and J. M. Lu, “Retailoring for fast, on-the-fly trace generation for NoC design space exploitation,” *Proc. - 2014 Int. Symp. Comput. Consum. Control. IS3C 2014*, pp. 1030–1033, 2014, doi: 10.1109/IS3C.2014.269.

[23] H. Matsutani, M. Koibuchi, Y. Yamada, D. F. Hsu, and H. Amano, “Fat H-Tree: A cost-efficient tree-based on-chip network,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 8, pp. 1126–1141, 2009, doi: 10.1109/TPDS.2008.233.

[24] L. S. Indrusiak *et al.*, “Evaluating the impact of communication latency on applications running over on-chip multiprocessing platforms: A layered approach,” *IEEE Int. Conf. Ind. Informatics*, pp. 148–153, 2010, doi: 10.1109/INDIN.2010.5549443.

[25] Y. Zhang, W. Zheng, X. Dong, and S. Gan, “A performance analytical approach based on queuing model for network-on-chip,” *Proc. - 3rd Int. Symp. Parallel Archit. Algorithms Program. PAAP 2010*, pp. 354–359, 2010, doi: 10.1109/PAAP.2010.46.

[26] K. Smiri and A. Jemai, “NoC-MPSoC performance estimation with Synchronous Data Flow (SDF) graphs,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6752 LNAI, pp. 406–415, 2011, doi: 10.1007/978-3-642-21538-4_40.

[27] G. Du, Y. Zhang, Z. Li, G. Liu, D. Zhang, and Y. Ouyang, “On the Accuracy of Stochastic Delay Bound for Network on Chip,” in *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*, 2017, pp. 1–8, doi: 10.1145/3130218.3130229.

[28] W. Liu *et al.*, “A Systematic and Realistic Network-on-Chip Traffic Modeling and Generation Technique for Emerging Many-Core Systems,” *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 2, pp. 113–126, 2018, doi: 10.1109/TMSCS.2017.2768362.

[29] G. M. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, p. 114, 1965, [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf.

[30] J. Hennessy and D. Patterson, “Chapter 1 - Fundamentals of Quantitative Design and Analysis,” in *Computer Architecture - A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2019, pp. 2–5.

[31] M. Ruggiero, “Chapter 3 - NoC Architectures,” in *Communication Architectures for Systems-on-Chip*, 1st ed., J. Ayala, Ed. CRC Press, 2011, pp. 84–124.

[32] “Snapdragon 801 Processor Brochure.” Qualcomm, p. 2, 2014, [Online]. Available: https://www.qualcomm.com/products/snapdragon-processors-801.

[33] L. Benini and G. De Micheli, “Networks on chips: A new SoC paradigm,” *Computer (Long. Beach. Calif).*, vol. 35, no. 1, pp. 70–78, 2002, doi: 10.1109/2.976921.

[34] W. Wolf, A. A. Jerraya, and G. Martin, “Multiprocessor system-on-chip (MPSoC) technology,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1701–1713, 2008, doi: 10.1109/TCAD.2008.923415.

[35] N. Nikitin and J. Cortadella, “A performance analytical model for Network-on-Chip with constant service time routers,” *Proc. 2009 Int. Conf. Comput. Des. - ICCAD ’09*, p. 571, 2009, doi: 10.1145/1687399.1687506.

[36] H. Chang, R. Karne, and A. Wijesinha, “Migrating a Bare PC Web Server to a Multi-core Architecture,” *Proc. - Int. Comput. Softw. Appl. Conf.*, vol. 2, pp. 216–221, 2016, doi: 10.1109/COMPSAC.2016.15.

[37] R. Krishnamurthy and G. N. Rouskas, “Performance evaluation of multi-core, multi-threaded SIP proxy servers (SPS),” *2016 IEEE Int. Conf. Commun. ICC 2016*, 2016, doi: 10.1109/ICC.2016.7511427.

[38] K. M. Hosny, M. M. Darwish, K. Li, and A. Salah, “Parallel multi-core CPU and GPU for fast and robust medical image watermarking,” *IEEE Access*, vol. 6, pp. 77212–77225, 2018, doi: 10.1109/ACCESS.2018.2879919.

[39] M. Rodrigues, N. Roma, and P. Tomas, “Fast and scalable thread migration for multi-core architectures,” *Proc. - IEEE/IFIP 13th Int. Conf. Embed. Ubiquitous Comput. EUC 2015*, pp. 9–16, 2015, doi: 10.1109/EUC.2015.36.

[40] T. Alexandru, “Networks on Chips,” Nov. 29, 2020. https://sites.google.com/site/alexandrutopirceanu/research/networks-on-chips (accessed Jun. 26, 2020).

[41] S. El-Ashry, M. Khamis, H. Ibrahim, A. Shalaby, M. Abdelsalam, and M. W. El-Kharashi, “On Error Injection for NoC Platforms: A UVM-Based Generic Verification Environment,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 39, no. 5, pp. 1137–1150, 2020, doi: 10.1109/TCAD.2019.2908921.

[42] A. S. Eissa *et al.*, “A reusable verification environment for NoC platforms using UVM,” *17th IEEE Int. Conf. Smart Technol. EUROCON 2017 - Conf. Proc.*, no. July, pp. 239–242, 2017, doi: 10.1109/EUROCON.2017.8011112.

[43] N. E. Jerger, T. Krishna, and L.-S. Peh, “Chapter 3 - Topology,” in *On-chip Networks*, 2nd ed., Morgan & Claypool, 2017, pp. 27–42.

[44] A. Shrivastava and S. K. Sharma, “Various arbitration algorithm for on-chip(AMBA) shared bus multi-processor SoC,” *2016 IEEE Students’ Conf. Electr. Electron. Comput. Sci. SCEECS 2016*, no. Fig 1, 2016, doi: 10.1109/SCEECS.2016.7509330.

[45] A. Kamath, G. Saxena, and B. Talawar, “Analysis of ring topology for NoC architecture,” *2015 Int. Conf. Comput. Netw. Commun. CoCoNet 2015*, pp. 381–388, 2016, doi: 10.1109/CoCoNet.2015.7411214.

[46] C. E. Leiserson, “Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing.,” *Proc. Int. Conf. Parallel Process.*, pp. 393–402, 1985.

[47] J. Kim, J. Balfour, and W. J. Dally, “Flattened butterfly topology for on-chip networks,” *Proc. Annu. Int. Symp. Microarchitecture, MICRO*, pp. 172–182, 2007, doi: 10.1109/MICRO.2007.29.

[48] C. Panem *et al.*, “Performance analysis of 16x16, 32x32, 64x64 2-D mesh topologies for network on chip application of MIMO,” pp. 1757–1764, 2017.

[49] A. Chariete, M. Bakhouya, J. Gaber, and M. Wack, “A design space exploration methodology for customizing on-chip communication architectures: Towards fractal NoCs,” *Integr. VLSI J.*, vol. 50, pp. 158–172, 2015, doi: 10.1016/j.vlsi.2014.11.007.

[50] F. Rhamdani, N. A. Suwastika, and M. A. Nugroho, “Equal-cost multipath routing in data center network based on software defined network,” *2018 6th Int. Conf. Inf. Commun. Technol. ICoICT 2018*, vol. 0, no. c, pp. 222–226, 2018, doi: 10.1109/ICoICT.2018.8528730.

[51] M. Thilagavathi and J. S. Sadiq, “High Performance Energy Efficient Grid Based Routing Algorithm for Multi Network on Chip,” *IEEE Int. Conf. Intell. Tech. Control. Optim. Signal Process. INCOS 2019*, pp. 1–3, 2019, doi: 10.1109/INCOS45849.2019.8951411.

[52] A. Kumar and B. Talawar, “Floorplan Based Performance Estimation of Network-on-Chips using Regression Techniques,” *2019 IEEE 5th International Conference for Convergence in Technology, I2CT 2019 (2019)*. IEEE, Pune, India, pp. 1–6, 2019.

[53] C. Chen, Q. Li, N. Li, H. Liu, and Y. Dai, “Link-Sharing: Regional Congestion Aware Routing in 2D NoC by Propagating Congestion Information on Idle Links,” *2018 IEEE 3rd Int. Conf. Integr. Circuits Microsystems, ICICM 2018*, no. 1, pp. 291–297, 2018, doi: 10.1109/ICAM.2018.8596400.

[54] G. Du, X. Yang, F. Chen, D. Zhang, Y. Song, and C. Peng, “MPCC: Multi-path routing packet connect circuit for network-on-chip,” *Proc. Int. Conf. Anti-Counterfeiting, Secur. Identification, ASID*, vol. 2016-Febru, pp. 86–91, 2016, doi: 10.1109/ICASID.2015.7405667.

[55] N. E. Jerger, T. Krishna, and L.-S. Peh, “Chapter 4 - Routing,” in *On-chip Networks*, 2nd ed., Morgan & Claypool, 2017, pp. 43–56.

[56] L. Wang, X. Wang, H. F. Leung, and T. Mak, “A Non-Minimal Routing Algorithm for Aging Mitigation in 2D-Mesh NoCs,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 38, no. 7, pp. 1373–1377, 2019, doi: 10.1109/TCAD.2018.2855149.

[57] M. Kumar *et al.*, “A novel non-minimal/minimal turn model for highly adaptive routing in 2D NoCs,” *Proc. - 2014 8th IEEE/ACM Int. Symp. Networks-on-Chip, NoCS 2014*, pp. 184–185, 2015, doi: 10.1109/NOCS.2014.7008785.

[58] P. Bahrebar and D. Stroobandt, “Hamiltonian path strategy for deadlock-free and adaptive routing in diametrical 2D mesh NoCs,” *Proc. - 2015 IEEE/ACM 15th Int. Symp. Clust. Cloud, Grid Comput. CCGrid 2015*, vol. 1, pp. 1209–1212, 2015, doi: 10.1109/CCGrid.2015.112.

[59] C. P. Young and Y. B. Lin, “A deadlock-free NoC architecture for the sea of heterogeneous processing elements,” *6th Int. Conf. Inf. Sci. Technol. ICIST 2016*, vol. 1, pp. 199–203, 2016, doi: 10.1109/ICIST.2016.7483410.

[60] Y. R. Chen, Z. R. Wangt, P. A. Hsiunqt, S. J. Chen, and M. H. Tsai, “Backward probing deadlock detection for networks-on-chip,” *2013 7th IEEE/ACM Int. Symp. Networks-on-Chip, NoCS 2013*, pp. 1–2, 2013, doi: 10.1109/NoCS.2013.6558396.

[61] R. Al-Dujaily, T. Mak, F. Xia, A. Yakovlev, and M. Palesi, “Run-time deadlock detection in networks-on-chip using coupled transitive closure networks,” *Proc. -Design, Autom. Test Eur. DATE*, pp. 497–502, 2011, doi: 10.1109/date.2011.5763086.

[62] W. J. Dally and B. Towles, “Chapter 14 - Deadlock and Livelock,” in *Principles and Practices of Interconnection Networks*, 1st ed., Morgan Kaufmann, 2004, pp. 257–285.

[63] N. E. Jerger, T. Krishna, and Li-Shiuan Peh, “Chapter 5 - Flow Control,” in *On-chip Networks*, 2nd ed., Morgan & Claypool, 2017, pp. 57–73.

[64] L. Thiele, E. Wandeler, and W. Haid, “Performance Analysis of Distributed Embedded Systems,” in *Embedded Systems Handbook*, 2nd ed., R. Zurawski, Ed. CRC Press, 2009, pp. 10-5-10–17.

[65] E. K. Ardestani and J. Renau, “ESESC: A fast multicore simulator using Time-Based Sampling,” *Proc. - Int. Symp. High-Performance Comput. Archit.*, pp. 448–459, 2013, doi: 10.1109/HPCA.2013.6522340.

[66] H. Zhao, M. Kandemir, W. Ding, and M. J. Irwin, “Exploring heterogeneous NoC design space,” *IEEE/ACM Int. Conf. Comput. Des. Dig. Tech. Pap. ICCAD*, pp. 787–793, 2011, doi: 10.1109/ICCAD.2011.6105419.

[67] W. Zuo *et al.*, “Accurate High-level Modeling and Automated Hardware/Software Co-design for Effective SoC Design Space Exploration,” *Proc. 54th Annu. Des. Autom. Conf. 2017 - DAC ’17*, pp. 1–6, 2017, doi: 10.1145/3061639.3062195.

[68] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” *Int. Symp. Code Gener. Optim. CGO.*, no. c, pp. 75–86, 2004, doi: 10.1109/CGO.2004.1281665.

[69] L. Jain, “NIRGAM - A Simulator for NoC Interconnect Routing and Application Modeling.” http://nirgam.ecs.soton.ac.uk/ (accessed Jul. 14, 2020).

[70] Y. Liu, L. Barford, and S. S. Bhattacharyya, “Generalized Graph Connections for Dataflow Modeling of DSP Applications,” *IEEE Work. Signal Process. Syst. SiPS Des. Implement.*, vol. 2018-Octob, pp. 275–280, 2018, doi: 10.1109/SiPS.2018.8598305.

[71] W. Reisig, *Understanding Petri nets: Modeling Techniques, Analysis Methods, Case Studies*, vol. 3, no. 3. Springer-Verlag, 2013.

[72] G. Kahn, “The semantics of a simple language for parallel programming,” *Inf. Process. Proc. IFIP Congr.*, vol. 74, pp. 471–475, 1974, doi: 10.1007/BF00288686.

[73] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij, “Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis,” *IET Comput. Digit. Tech.*, vol. 3, no. 5, p. 398, 2009, doi: 10.1049/iet-cdt.2008.0093.

[74] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*, 2nd ed. CRC Press, 2009.

[75] D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Trans. Comput.*, vol. C–36, no. 1, pp. 24–35, 1987, doi: 10.1109/TC.1987.5009446.

[76] S. Sriram and S. S. Bhattacharyya, “Chapter 3 - BACKGROUND TERMINOLOGY AND NOTATION,” in *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed., CRC Press, 2009, pp. 35–58.

[77] A. H. Ghamarian, M. C. W. Geilen, T. Basten, and S. Stuijk, “Throughput Analysis of Synchronous Data Flow Graphs,” *2008 Des. Autom. Test Eur.*, 2008, doi: 10.1109/DATE.2008.4484672.

[78] S. Sriram and S. S. Bhattacharyya, “Chapter 8 - ANALYSIS OF THE ORDERED-TRANSACTIONS STRATEGY,” in *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed., CRC Press, 2009, pp. 168–173.

[79] P. Glanon, S. Azaiez, and C. Mraidha, “Estimating latency for synchronous dataflow graphs using periodic schedules,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2019, vol. 11847 LNCS, pp. 79–94, doi: 10.1007/978-3-030-35092-5_6.

[80] O. Marchetti and A. Munier-Kordon, “A sufficient condition for the liveness of weighted event graphs,” *Eur. J. Oper. Res.*, vol. 197, no. 2, pp. 532–540, 2009, doi: 10.1016/j.ejor.2008.07.037.

[81] A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen, “Latency minimization for synchronous data flow graphs,” in *Proceedings - 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, DSD 2007*, 2007, pp. 189–196, doi: 10.1109/DSD.2007.4341468.

[82] G. Kuiper and M. J. G. Bekooij, “Latency analysis of homogeneous synchronous dataflow graphs using timed automata,” in *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, 2017, pp. 902–905, doi: 10.23919/DATE.2017.7927116.

[83] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij, “Applying dataflow analysis to dimension buffers for guaranteed performance in networks on chip,” *Proc. - Second IEEE Int. Symp. Networks-on-Chip, NOCS 2008*, pp. 211–212, 2008, doi: 10.1109/NOCS.2008.4492742.

[84] M. S. B and M. Geilen, “Towards Component-Based (max,+) Algebraic Throughput Analysis of Hierarchical Synchronous Data Flow Models,” in *SAFECOMP 2017 Workshops*, 2017, vol. 10489, pp. 462–476, doi: 10.1007/978-3-319-66284-8.

[85] a. Bonfietti, L. Benini, M. Lombardi, and M. Milano, “An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms,” *Des. Autom. & Test Eur. Conf. & Exhib. (DATE), 2010*, pp. 897–902, 2010, doi: 10.1109/DATE.2010.5456924.

[86] R. Berry, “Queueing Theory,” *Whitman Coll.*, no. 1, pp. 1–12, 2010.

[87] R. G. Gallager, *Stochastic Processes - Theory for Applications*, 1st ed. New York: Cambrige University Press, 2013.

[88] R. Nelson, *Probability, Stochastic Processes, and Queueing Theory - The Mathematics of Computer Performance Modeling*, 1st ed. New York: Springer Science+Business Media, 1995.

[89] H. Ouyang and B. L. Nelson, “Simulation-based predictive analytics for dynamic queueing systems,” in *Proceedings - Winter Simulation Conference*, 2018, pp. 1716–1727, doi: 10.1109/WSC.2017.8247910.

[90] Y. Ben-Itzhak, I. Cidon, and A. Kolodny, “Average latency and link utilization analysis of heterogeneous wormhole NoCs,” *Integr. VLSI J.*, vol. 51, pp. 92–106, 2015, doi: 10.1016/j.vlsi.2015.07.002.

[91] D. Bhattacharya and N. K. Jha, “Analytical Modeling of the SMART NoC,” *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 3, no. 4, pp. 242–254, 2017, doi: 10.1109/TMSCS.2017.2704101.

[92] T. Krishna, C. H. O. Chen, W. C. Kwon, and L. S. Peh, "Breaking the on-chip latency barrier using SMART," *Proc. - Int. Symp. High-Performance Comput. Archit.*, vol. 1, no. 2, pp. 378–389, 2013, doi: 10.1109/HPCA.2013.6522334.

[93] R. Sabbaghi-Nadooshan and A. Patooghy, "Analytical performance modeling of de Bruijn inspired mesh-based network-on-chips," *Microprocess. Microsyst.*, vol. 39, no. 1, pp. 27–36, 2015, doi: 10.1016/j.micpro.2014.12.002.

[94] J.-Y. Le Boudec and P. Thiran, *NETWORK CALCULUS A Theory of Deterministic Queuing Systems for the Internet*, 1st ed. Springer-Verlag, 2004.

[95] A. Bouillard, M. Boyer, and E. Le Corronc, *Deterministic Network Calculus - From Theory to Practical Implementation*, 1st ed. Hoboken, NJ: WYLEY, 2018.

[96] M. Bakhouya, S. Suboh, J. Gaber, and T. El-Ghazawi, "Analytical modeling and evaluation of on-chip interconnects using network calculus," *Proc. - 2009 3rd ACM/IEEE Int. Symp. Networks-on-Chip, NoCS 2009*, pp. 74–79, 2009, doi: 10.1109/NOCS.2009.5071447.

[97] F. Verbeek and N. Van Vugt, "Estimating worst-case latency of on-chip interconnects with formal simulation," *Proc. 17th Conf. Form. Methods Comput. Des. FMCAD 2017*, pp. 204–211, 2017, doi: 10.23919/FMCAD.2017.8102261.

[98] L. Paulson and T. Nipkow, "Isabelle." https://isabelle.in.tum.de/ (accessed Apr. 04, 2021).

[99] G. Du, Y. Zhang, and Z. Li, "On the Accuracy of Stochastic Delay Bound for Network on Chip," in *Proceedings of NOCS'17*, 2017, p. 8.

[100] F. Giroudot and A. Mifdaoui, "Buffer-aware worst-case timing analysis of wormhole NoCs using network calculus," *Proc. IEEE Real-Time Embed. Technol. Appl. Symp. RTAS*, no. i, pp. 37–48, 2018, doi: 10.1109/RTAS.2018.00010.

[101] Mellanox Technologies, "TILE-Gx8036." https://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx36.pdf (accessed Apr. 04, 2021).

[102] U. Y. Ogras and R. Marculescu, "Analytical router modeling for networks-on-chip performance analysis," *Proc. -Design, Autom. Test Eur. DATE*, pp. 1096–1101, 2007, doi: 10.1109/DATE.2007.364440.

[103] W. Lui and Z. Tian, "MCLS - Realistic Network-on-Chip Traffic Patterns." https://eexu.home.ece.ust.hk/traffic.html (accessed May 31, 2021).

[104] W. Liu and J. Xu, "MCSL Network-on-Chip Traffic Suite User Manual." Mobile Computing System Lab - Hong Kong University of Science and Technology, Hong Kong, 2014, [Online]. Available: https://eexu.home.ece.ust.hk/traffic.html.

[105] D. U. Becker and W. J. Dally, "Allocator implementations for network-on-chip routers," *Proc. Conf. High Perform. Comput. Networking, Storage Anal. SC '09*, 2009, doi: 10.1145/1654059.1654112.

[106] D. U. Becker, *Efficient Microarchitecture for Network-on-Chip Routers*, no. August. 2012.

[107] "GitHub - booksim/booksim2: BookSim 2.0." https://github.com/booksim/booksim2 (accessed Apr. 29, 2021).

[108] N. Jiang, G. Michelogiannakis, D. Becker, B. Towles, and W. J. Dally, "BookSim 2.0 User's Guide," 2013. Accessed: Apr. 29, 2021. [Online]. Available: https://nocs.stanford.edu/cgi-bin/svn.cgi/booksim2.0.

[109] Accellera Systems Initiative, “SystemC,” 2020. https://www.accellera.org/downloads/standards/systemc (accessed Aug. 14, 2020).

[110] K. Duraisamy and P. P. Pande, “Performance Evaluation and Design Trade-offs for wireless-enabled SMART NoC,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017, pp. 1360–1365.

[111] H. Zhao, X. Cheng, S. P. Mohanty, and J. Fang, “Designing scalable hybrid wireless NoC for GPGPUs,” *Proc. IEEE Comput. Soc. Annu. Symp. VLSI, ISVLSI*, vol. 2018-July, pp. 703–708, 2018, doi: 10.1109/ISVLSI.2018.00132.

[112] D. Patti, “Noxim: Network-on-Chip Simulator.” https://github.com/davidepatti/noxim (accessed May 27, 2021).

[113] A. Akram and L. Sawalha, “A Comparison of x86 Computer Architecture Simulators,” *Sch. WMU*, 2016, [Online]. Available: http://scholarworks.wmich.edu/casrl_reportshttp://scholarworks.wmich.edu/casrl_reports/1.

[114] J. Lowe-Power, “Ruby memory subsystem.” http://www.gem5.org/documentation/general_docs/ruby/ (accessed May 29, 2021).

[115] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha, “GARNET: A detailed on-chip network model inside a full-system simulator,” *ISPASS 2009 - Int. Symp. Perform. Anal. Syst. Softw.*, pp. 33–42, 2009, doi: 10.1109/ISPASS.2009.4919636.

[116] T. Krishna, “Garnet2.0 - gem5.” http://www.m5sim.org/Garnet2.0 (accessed May 29, 2021).

[117] J. Lowe-Power, “Garnet Interconnection network.” http://www.gem5.org/documentation/general_docs/ruby/interconnection-network/ (accessed Jun. 04, 2021).

[118] “public/gem5 - Git at Google.” https://gem5.googlesource.com/public/gem5 (accessed May 29, 2021).

[119] J. Lowe-Power, “gem5: The gem5 simulator system.” http://www.gem5.org/ (accessed May 29, 2021).

[120] D. J. Sorin, M. D. Hill, and D. A. Wood, “A primer on memory consistency and cache coherence,” *Synth. Lect. Comput. Archit.*, vol. 16, pp. 1–212, Jan. 2011, doi: 10.2200/S00346ED1V01Y201104CAC016.

[121] J. Lowe-Power, “Garnet standalone.” https://www.gem5.org/documentation/general_docs/ruby/Garnet_standalone/ (accessed Jun. 06, 2021).

[122] J. Lowe-Power, “gem5: SLICC.” http://www.gem5.org/documentation/general_docs/ruby/slicc/ (accessed Jun. 14, 2021).

[123] J. Lowe-Power, “gem5: MSI example cache protocol.” http://www.gem5.org/documentation/learning_gem5/part3/cache-intro/ (accessed Jun. 14, 2021).

[124] Python Software Foundation, “Welcome to Python.org,” 2021. https://www.python.org/ (accessed Sep. 08, 2021).

[125] S. Scherfke and O. Lünsdorf, “Overview — SimPy 3.0.7 documentation.” https://simpy.readthedocs.io/en/latest/index.html (accessed Aug. 21, 2021).

[126] R. Sedgewick and K. Wayne, “Chapter 4 - Graphs. Section 4.4 Shortest Paths,” in *Algorithms*, 4th Ed., Addison-Wesley, 2011.

[127] N. E. Jerger, T. Krishna, and L.-S. Peh, *On-chip networks, 2nd edition*, 2nd ed. New York: Morgan & Claypool, 2017.

[128] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, 1st ed. London: Morgan Kaufmann, 2004.

[129] “Tushar Krishna | School of Electrical and Computer Engineering at the Georgia Institute of Technology,” 2021. https://www.ece.gatech.edu/faculty-staff-directory/tushar-krishna (accessed Sep. 10, 2021).

[130] “Re: [gem5-users] Changing buffer depth in virtual channels - Garnet.” https://www.mail-archive.com/gem5-users@gem5.org/msg16653.html (accessed Sep. 07, 2021).

[131] J. Kim and H. Kim, “Router microarchitecture and scalability of ring topology in on-chip networks,” p. 5, 2009, doi: 10.1145/1645213.1645217.

[132] Y. Demir and N. Hardavellas, “SLaC: Stage laser control for a flattened butterfly network,” *Proc. - Int. Symp. High-Performance Comput. Archit.*, vol. 2016-April, pp. 321–332, 2016, doi: 10.1109/HPCA.2016.7446075.

[133] J. Kim, W. J. Dally, and D. Abts, “Flattened butterfly: A cost-efficient topology for high-radix networks,” *Proc. - Int. Symp. Comput. Archit.*, pp. 126–137, 2007, doi: 10.1145/1250662.1250679.

[134] J. Fong and S. Nath, “ORION3.0,” 2012. https://vlsicad.ucsd.edu/ORION3/index.html (accessed Jul. 17, 2018).

[135] A. B. Kahng, Bin Li, Li-Shiuan Peh, and K. Samadi, “ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration,” *2009 Des. Autom. Test Eur. Conf. Exhib.*, pp. 423–428, 2009, doi: 10.1109/DATE.2009.5090700.

[136] A. Y. Yamamoto and C. Ababei, “Unified reliability estimation and management of NoC based chip multiprocessors,” *Microprocess. Microsyst.*, vol. 38, no. 1, pp. 53–63, 2014, doi: 10.1016/j.micpro.2013.11.009.

[137] M. Valinataj, S. Mohammadi, and S. Safari, “Reliability assessment of networks-on-chip based on analytical models,” *J. Zhejiang Univ. A*, vol. 10, no. 12, pp. 1801–1814, 2009, doi: 10.1631/jzus.A0820853.