Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Performance Validation of Networks on Chip

Karthik Chandrasekar

CE-MS-2009-08

Abstract

Network-on-Chip (NoC) is established as the most scalable and efficient solution for the on-chip communication challenges in the multi-core era, since it guarantees scalable high-speed communication with minimal wiring overhead and physical routing issues. However, efficiency of the NoC depends on its design decisions, which must be made considering the performance requirements and the cost budgets, specific to the target application. In the NoC design flow, merely verifying and validating the design for its adherence to the application's average communication requirements may be insufficient, when the need is to get the best performance within tight power and area budgets. This calls for NoC design validation and optimization under real-time congestions and contentions imposed by the target application. However, application availability issues (due to Intellectual Property restrictions), force us to look at alternative solutions to mimic the target application behavior and help us arrive at an efficient and optimal NoC design. This thesis is a step in the said direction, and proposes a performance analysis and validation tool (infrastructure) that employs synthetic and application trace-based traffic generators, to efficiently emulate the expected communication behavior of the target application. Novel methods are suggested to model and generate deterministic and random traffic patterns and to port reference application traces from and to different interconnect architectures (from buses to NoCs or vice versa). Further, these traffic generators are supported by efficient traffic management/scheduling schemes, that aid in effective analysis of the NoC's performance. The proposed tool, also includes a statistics collection and performance validation module that checks the designed network for adherence to the performance requirements of the target application and explores trade-offs in performance and area/power costs to arrive at optimal architectural solutions. The significance of this tool, lies in its ability to comprehensively validate a given NoC design and suggest optimizations, in the light of the target applications expected run-time communication behavior.

Faculty of Electrical Engineering, Mathematics and Computer Science

Performance Validation of Networks on Chip

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Karthik Chandrasekar
born in Chennai, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Performance Validation of Networks on Chip

by Karthik Chandrasekar

Abstract

Network-on-Chip (NoC) is established as the most scalable and efficient solution for the on-chip communication challenges in the multi-core era, since it guarantees scalable high-speed communication with minimal wiring overhead and physical routing issues. However, efficiency of the NoC depends on its design decisions, which must be made considering the performance requirements and the cost budgets, specific to the target application. In the NoC design flow, merely verifying and validating the design for its adherence to the application's average communication requirements may be insufficient, when the need is to get the best performance within tight power and area budgets. This calls for NoC design validation and optimization under real-time congestions and contentions imposed by the target application. However, application availability issues (due to Intellectual Property restrictions), force us to look at alternative solutions to mimic the target application behavior and help us arrive at an efficient and optimal NoC design. This thesis is a step in the said direction, and proposes a performance analysis and validation tool (infrastructure) that employs synthetic and application trace-based traffic generators, to efficiently emulate the expected communication behavior of the target application. Novel methods are suggested to model and generate deterministic and random traffic patterns and to port reference application traces from and to different interconnect architectures (from buses to NoCs or vice versa). Further, these traffic generators are supported by efficient traffic management/scheduling schemes, that aid in effective analysis of the NoC's performance. The proposed tool, also includes a statistics collection and performance validation module that checks the designed network for adherence to the performance requirements of the target application and explores trade-offs in performance and area/power costs to arrive at optimal architectural solutions. The significance of this tool, lies in its ability to comprehensively validate a given NoC design and suggest optimizations, in the light of the target applications expected run-time communication behavior.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2009-08

Committee Members :

Advisor:	Dr. ir. Georgi Gaydadjiev, CE, TU Delft
Advisor:	Prof. Giovanni De Micheli, LSI, EPFL
Chairperson:	Prof. Kees Goossens, CE, TU Delft
Member:	Dr. ir. Rene van Leuken, CAS, TU Delft

To my parents

Contents

List of Figures

List of Tables

Acknowledgements

Firstly, I would like to thank Prof. Giovanni De Micheli for giving me the opportunity to work on my MSc thesis, in his group (LSI) at EPFL, Switzerland. I would also like to express my gratitude to Dr. Federico Angiolini (Post. Doc at LSI, EPFL) for his guidance and support throughout the length of this thesis work. I would also like to thank Dr. Srinivasan Murali (Post. Doc at LSI, EPFL), Antonio Pullini and Ciprian Seiculescu (PhD students at LSI, EPFL), Jaume Joven (PhD student at UAB, Spain) and Dara Rahmati (PhD student at Sharif University, Iran) for their ideas, suggestions and all help during the course of my work at EPFL.

I would specially like to thank Prof. Georgi Gaydadjiev, who has been ever supportive and helpful, since my first days at TU Delft, for accepting to supervise my thesis from Delft and providing timely suggestions and ideas to improve my work.

I would also like to thank Prof. Kees Goossens for giving several suggestions to improve the thesis contents; hopefully some of them have been accomodated. I would also like to thank Prof. Rene van Leuken for accepting to judge my thesis defense and suggesting changes and corrections to the report.

Finally, I would like to thank my parents for all their love and support and Prof.Venkateswaran at WARFT for motivating me to undertake research during my bachelor's itself. Last but not the least, I would like to thank my friends Madhavan and Vinoth for the wonderful time we spent together in the last 2 years.

Karthik Chandrasekar
Delft, The Netherlands
November 30, 2009

Introduction 1

1.1 Why Networks On Chip?

The ever increasing demand for processor performance countered by the power consumption barrier has lead computer architects to design multi-processor and multi-core single chip architectures [26] [1]. As technology scales beyond deep sub-micron and offers increasing integration density, the assembling of a complete system consisting of a large number of IP blocks (e.g. processors, accelerators, memories, I/O controllers) on the same silicon die has become technically feasible.

Today, chips comprise of tens or even hundreds of these building blocks, often very heterogeneous in pin-out, performance, geometric size and shape, clocking requirements, etc. As the complexity of such MPSoC designs skyrockets, one of the crucial bottlenecks has been identified as the on-chip interconnection infrastructure [23].

Most current SoC designs are based on shared buses due to their low cost. Unfortunately, scalability is limited on shared buses due to the serialization of multiple access requests. Among the key design challenges for an efficient communication infrastructure, some of the most prominent ones include, bandwidth scalability, efficient wiring and accurate routing of data.

A solution to these challenges has been identified in Networks-on-Chip (NoC) [15], where the communication is always point-to-point and packet-switched and messages are transferred from source to destination across several links and switches (routers). While this allows unlimited bandwidth scalability (i.e. by adding more on-chip routers and links), it also ensures that the wiring is kept tidy and length bounded.

NoCs are now being considered by many, as a viable alternative to design scalable communication architecture for present and future generation SoCs [32]. In multimedia processors, inter-core communication demands often scale up to the range of GB/s and this demand is expected to peak with the integration of several heterogeneous high performance cores into a single chip. To meet such increasing bandwidth demands, state-of-the-art buses such as STBus and AMBA, instantiate multiple buses operating in parallel thereby providing a crossbar-like architecture, which however, still remains inherently non-scalable. To effectively tackle the interconnect complexity of MPSoCs, a scalable and high performance interconnect architecture is needed and hence, NoCs [24] [16].

1.2 Network on Chip Architecture

Emerging System-on-Chip (SoC) designs consist of a number of interconnected heterogeneous devices. NoCs can be described as the on-chip interconnect that connects up these heterogeneous IP blocks, provides support for multiple clock domains on a single chip and facilitates communication across the IP blocks based on predefined protocols and routing schemes. For efficient functioning of NoCs, the three components: the Network Interfaces, the Routers and the Links play very significant roles. A conceptual view of the Network on Chip architecture is depicted in Figure 1.1.

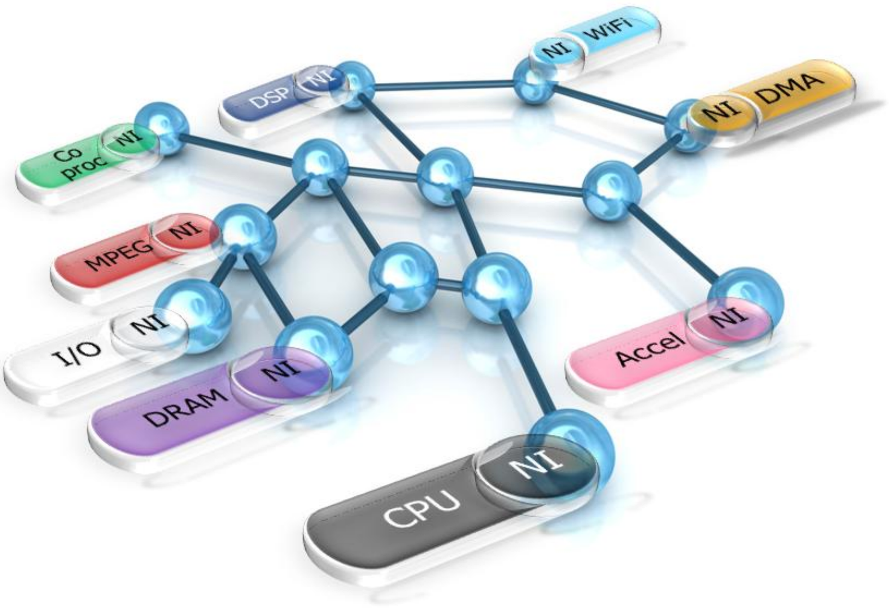


Figure 1.1: Conceptual view of Network on Chip

As can be seen in the figure above, the different IP cores such as CPU, Accelerators, DMAs, I/O etc., are connected to the NoC infrastructure through the Network Interfaces, which in turn connect to a port on any of the Routers which then connect among themselves, forming the NoC.

The Network Interface is employed to provide protocol-specific communication, by converting core-specific signals into a common packet format, performing packetization/depacketization of data and implementing the service level protocols associated with each transaction in the NoC. In simple terms, a Network Interface translates messages from its IP core to a standard protocol when its IP core sends messages into the network and from the standard protocol to that of its IP core when it receives messages, thus supporting SoCs that accommodate heterogeneous IP cores and coordinating the transmission and reception of packets from/to the core.

The Routers are used to establish the links across the IPs, such that the data packets can be transferred from any source to any destination while making routing decisions at the switches. The routers can be arbitrarily connected to each other and to NIs, based on a specified topology. They include routing, switching and flow control logic. Routing schemes help in finding a path between any source and destination IP block, while minimizing the number of hops to transport packets across the NoC infrastructure in a parallel and pipelined fashion.

The connecting links are also a critical component of NoCs. They connect NIs and routers and help in transmitting data packets over the network. The routers are used to route the packets from source to destination, while links are used to connect the various cores and routers together.

Besides these three main components, the flow control techniques also play a significant role in the working of NoCs by defining how packets should be moved through the network while providing performance guarantees in Quality of Service (QoS). Flow control techniques also help in dealing with situations where two packets arrive at the same link at the same time (contention).

1.3 Network on Chip Design Flow

Designing NoCs to meet the functional specification is a complex task and it involves lot of design trade-offs. As a consequence, the entire design process [27] [18] is categorized into several phases. The design choices made at each phase have a significant impact on the overall performance of the NoC and the following phases as well. For instance, a design choice made during topology selection phase will have an impact on the overall performance and will also influence the consequent phases like mapping, routing scheme selection etc. In general, the distinct phases in NoC Design flow can be classified as:

(a) **Application Description** - It is responsible for providing a unified representation of the communication patterns. In certain cases these patterns also include communication types, frequencies etc. The general characterization is done by means of a graph where each vertex represents a computational module in the application referred to as task and the edge denotes the dependencies between the tasks. All the entities are annotated with additional information specifying other communication characteristics. Alternatively a spreadsheet can also be used, wherein each worksheet can give a description of the applications communication requirement for a particular use case.

(b) **Topology Selection** - It involves exploring various design objectives such as average communication delay, area, power consumption etc. While advantages of resorting to regular topologies hold for homogeneous SoCs, this is no longer true in the case of heterogeneous SoCs. The design choices span between full custom topologies and standard regular topology. The designer could even adopt a hierarchical topology scheme to satisfy the system requirements. Also, the floorplan information can aid in topology design/selection process.

(c) **IP Mapping** - It is the process of determining how to map the selected IPs on the communication architecture and also satisfy the design requirements. Different approaches have been proposed to achieve efficient mapping involving branch and bound algorithm, multi-objective mapping etc.

(d) **Architecture Configuration** - It involves fixing up of buffer sizes, routing and switching schemes etc. Different strategies are adopted by various design flows to select values that suit the architecture's communication requirements. Here, since the design space considered is fairly large and complex, some heuristic based exploration techniques are employed to arrive at near-optimum solution.

(e) **Design Synthesis** - It involves description of the network components in hardware modeling language and this is achieved by using tools, in the synthesis phase. Also, standard network component libraries for switches, routers and network interfaces can be used.

(f) **Design Validation and Simulation** - Validation of the implementation of the NoC architecture is useful in verifying the design against the initial requirements in terms of communication latencies, throughput, area and power.

The cost and performance numbers are obtained by simulations and depend on the selected network components and the topology, and the setting of their corresponding parameters. This final phase of the design flow also helps tune the NoC parameters to suit the target application's behavior.

In the next section, we specifically look into the design flow of the Xpipes NoC [22] [17], which is the case study NoC being used in this thesis, for simulation purposes.

1.4 Xpipes NoC Design Flow

The Xpipes NoC Design Flow [13] is used to generate efficient NoCs using Xpipes architecture [17] with custom topology to satisfy the design constraints of the application. The objective is of the design flow is to minimize network's power consumption and the number of hops.

The Xpipes design flow also uses a floor-planner to estimate design area and wire lengths for selecting topologies that meet the requirements both in terms of power consumption and target frequency of operation. This helps achieve fewer design re-spins, as accurate floor-plan information is made available early in the design cycle. Also deadlock free routing methods are considered to ensure proper NoC operation.

In the first phase of the design flow, the constraints and objectives to be satisfied by the NoC architecture are specified. Information on application traffic characteristics, area delay and power models etc are also obtained.

In the second phase, the NoC which satisfies all the constraints is automatically synthesized. There are different steps involved in this phase. Firstly, frequency and link width are varied between a set of suitable values. Then the synthesis step is performed for each set of architectural parameters thereby exploring the various design choices. This step involves establishing connectivity between the switches and cores and finding deadlock free routes for the different traffic flows. In the last phase, RTL code required for the various network components instantiated in the design is automatically generated. It uses Xpipes library [35], which comprises of soft macros of the components and Xpipes Compiler [29]to interconnect network elements with the core. The design flow of the Xpipes NoC is shown in Figure 1.2 and is based on the Xpipes design flow suggested in Figure 1.5 in [12] and Figure 3.1 in [33].

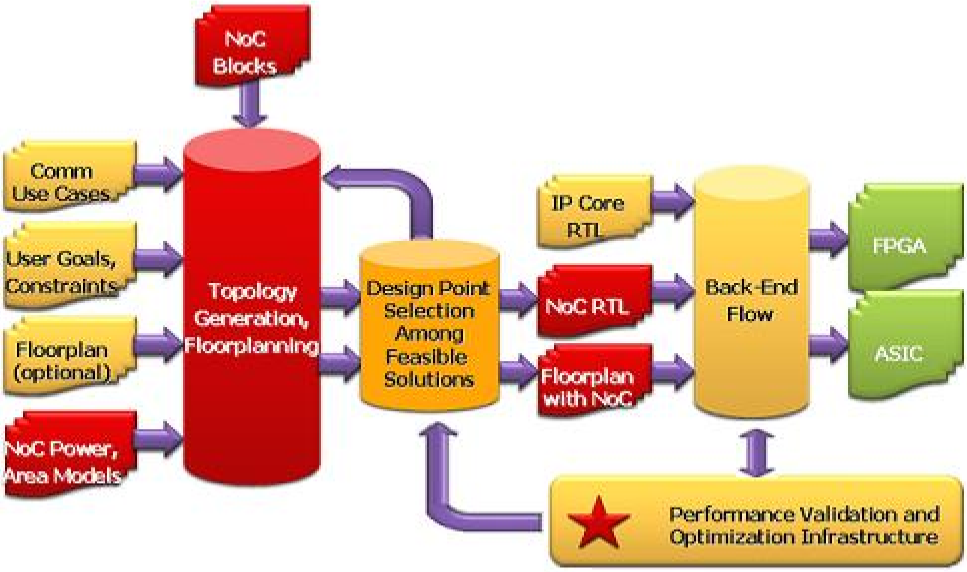


Figure 1.2: Xpipes Network on Chip Design Flow

As can be seen in the figure above, performance goals and power and area budgets are obtained from the user and the NoC components in different configurations and their corresponding power and area models are obtained from the Xpipes NoC library. Based on these requirements and constraints, a suitable architecture and topology is generated and using optimization heuristics, a set of feasible architectural solutions is obtained. The Xpipes compiler then generates the RTL for one of the design solutions.

1.5 Motivation and Objective

As mentioned in the previous section, in the Xpipes NoC Design Flow, all the performance objectives (in terms of average throughput and latency requirements) and design constraints (in terms of power and area budgets) are specified in the first phase itself and proper adherence to the same is verified throughout, with the aim to guarantee high performance and low power and area costs.

However, the design process is yet incomplete, since there is still a need to verify the network design for performance and efficiency against real-time constraints (congestions and contentions) imposed by a real application/benchmark, so as to arrive at a concrete and optimal NoC design for a given application.

It is well established that parallel injection of traffic from different IP cores or processors in an MPSoC environment, causes contention for network's resources. Although, links are designed to provide adequate bandwidth to meet the average requirements of the application, traffic injection instances with high levels of contention, lead to congestion, due to the design choices for network components, thus affecting the network's performance. To overcome or subside this issue, network's designers tend to over-design the NoC, such that despite the congestions and contentions, the throughput and latency requirements of the application are met. However, such over-design adversely impacts the power and area costs and hence, calls for an additional effort to validate and optimize the design such that the performance objectives and cost constraints are met.

In order to arrive at efficient and optimal network design solutions, it becomes essential to verify the same against the run-time behavior of the target application, since that would present a very realistic picture of the network's performance at run-time and thereby, an accurate estimate of the required over-design. Hence, it becomes extremely crucial to incorporate such a phase in the design process to arrive at an efficient and optimal design of the NoC, as can be seen in Figure 1.2.

This can be done with adequate information about the target application from the user, though it may well be very restricted due to application's Intellectual Property issues. What may be available could be information such as, expected traffic pattern or a trace obtained from a reference system, which uses a different existing interconnect such as a shared bus. Hence, it is suggested to instead employ synthetic or application trace-based traffic generators, that effectively emulate the expected behavior of the target application.

The synthetic traffic generators produce traffic based on given probability distributions or traffic shapes or patterns that can be expected for the given application in the given system setup. This description may be specified by the user, as a substitute for application details such as source code or scheduling information.

The application trace-based traffic generators re-produce traffic from a reference application trace (obtained from a reference system), by modeling and porting the same to the NoC-based simulated system. The reference trace may be made available from a cycle-accurate reference simulator system or a functional simulator system, with unknown or constant reference interconnect delays, which needs to be filtered out when porting the trace.

Modeling and porting of traces also involves deriving of application's realistic schedule, which helps maintain transaction ordering and application control flow. Such measures, assure that the process of modeling and porting the reference trace, yield a more accurate estimate of the application's expected run-time behavior, when compared to the synthetic traffic generation mechanism, by estimating and reproducing complex dependencies in the application such as during synchronization.

In comparison to a relevant recent effort in this direction in [31] where there is a need to use system-level information such as knowledge of semaphore variables and pre-defined memory address map to detect synchronization events, the suggested approach provides a more generic approach with the ability to automatically detect dependencies across all the transactions in the application, without using any such system-dependant information.

In this thesis, we address both the scenario's (synthetic and trace-based traffic generation) assuming availability of application information in both the formats (as traffic patterns/distributions and reference application traces), and can expect efficient validation and optimization. While the former method, is meant to give a direction for validation and optimization, the latter provides more accurate estimates of the network's performance and design issues.

The objective of this thesis is to come up with a performance validation tool/infrastructure which incorporates such traffic generators that help in performance validation and optimization of the NoC design. The thesis aims to addresses the following major challenges:

- Using traffic generators to model and re-generate target application's expected run-time behavior.
- Validation of the NoC design to meet the application's requirements.
- Suggest optimizations arriving at the best tradeoffs among performance and area/power constraints.

1.6 Contributions

To develop an infrastructure for performance analysis, validation and design optimization of NoCs, the tool employs synthetic and trace-based traffic generators, which effectively produce synthetic traffic and efficiently emulate traffic behavior of real applications, respectively.

In addition, novel methods are suggested to mimic non-deterministic traffic patterns in synthetic traffic generators and to arrive at traffic models that realistically capture the application communication behavior and schedule across the IP cores in trace-based traffic generators.

In synthetic traffic generation, traffic patterns using relevant probability distributions/analytical models are generated. In trace-based traffic generation, reference traces are employed and appropriate methods to migrate and emulate the same to different environments/interconnect, are suggested.

In order to obtain application traces, a set of benchmarks are executed on a cycle-accurate MPSoC emulator called MPARM [14], which employs ARM7 [30] processors.

The process of collecting statistics involves capturing the type and the timestamp of communication events at the boundary of every IP core in a reference environment. This opens up the possibility for communication infrastructure exploration and optimization and for the investigation of its impact on system performance at the highest level of accuracy under realistic workloads and different system configurations.

The performance validation tool/infrastructure proposed in this thesis helps in validating system level design decisions and verification of the implementation. It addresses the performance vs area and power tradeoffs and helps validate and optimize the NoC performance. In short, this thesis proposes a comprehensive infrastructure for performance analysis and trade-off exploration for on-chip communication architectures.

1.7 Thesis Organization

Chapter 2 of this thesis gives an overview of the Xpipes NoC architecture, the Xpipes Compiler and the MPARM MPSOC platform and their relevance to this study. Chapter 3 discusses synthetic traffic generation using relevant probability distributions, besides suggesting an efficient ‘peak and valleys’ approach for modeling non-deterministic distributions/curves in traffic patterns. It also suggests an efficient traffic management/scheduling scheme for the traffic generator, that defines the spatial distribution of the traffic in the network, in order to assure maximum possible stress on all links and to check for robustness of the NoC. Chapter 4 suggests a methodology for estimating IP processing times and deriving an application’s static schedule from a reference trace. It also suggests a method for employing an application’s dynamic schedule for better representation of the application’s behavior, besides describing the implementation of the appropriate schedule managers. Chapter 5 describes the methodology involved in statistics collection and analysis and presents a set of simulation results for a benchmark application. Chapter 6 concludes the thesis, highlighting the significance of the work and exploring opportunities for future work.

Xpipes and MPARM 2

2.1 Xpipes NoC

The Xpipes NoC [22] library provides efficient synthesizable, high frequency and low latency components (such as Network Interfaces, Routers and Links) which can be parameterized in terms of buffer depth, flit width, arbitration policies, flow control mechanisms etc. The Xpipes Compiler is employed to interconnect these network elements with the cores.

Xpipes NoC [17] is fully synchronous and yet supports multiple frequencies in the NIs. Routing is statically determined in the NIs. Xpipes uses wormhole switching [2] and best-effort services [3] for data transfers. There is also support for QoS provisions. Xpipes supports both input and/or output buffering, depending on flow control requirements and designer choices. In fact, since Xpipes supports multiple flow controls, the choice of buffering strategy is entwined with the selection of the flow control protocol. Xpipes also chooses to employ parallel links over virtual channels to resolve bandwidth issues, in order to reduce implementation costs.

2.2 Xpipes Building Blocks

The most critical components in any NoC architecture are the Network Interfaces, the Routers and the Links. An NoC is instantiated by interconnecting these network elements in different configurations to form a topology, which may either be specific, such as mesh or ring, or allow for arbitrary connectivity, matching the requirements of the target application. An overview of the Xpipes NoC architecture is depicted in Figure 2.1.

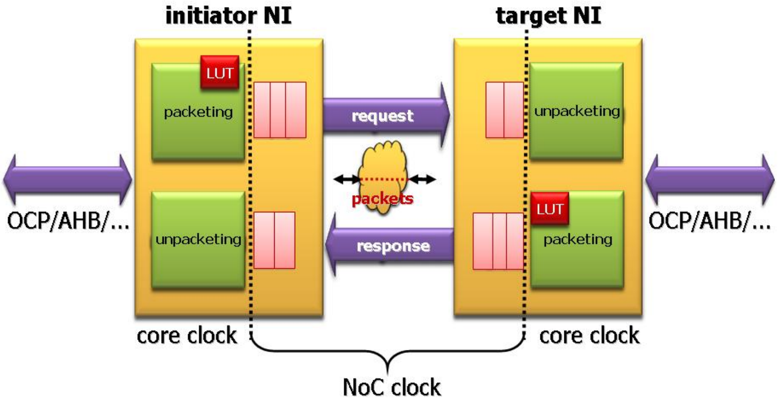


Figure 2.1: Overview of Xpipes NoC Architecture

As can be seen in the figure above, the Xpipes NoC has a simple architecture with a Network Interface for each of the sources and one for each of the targets. The Network Interface includes separate request and response paths, which include packetizing and de-packetizing logic. The arbitration happens at the routers, which decides which master/source gets priority on the links down stream. The Xpipes NIs also support multiple clock domains at the sources and targets. The Xpipes NoC building blocks are explained in detail in the following sub-sections.

2.2.1 Network Interfaces

A Network Interface is needed to connect each core to the NoC. Network Interfaces convert transaction requests into packets for injection into the network and receiving packets into transaction responses. When transmitting the packets, they are split into a sequence of FLITS (Flow Control Units), to minimize the physical wiring. This flit width in Xpipes is configurable based on the requirements. It can vary from as low as 4 wires to as high as 64-bit buses (up to 200 wires including address bus and control lines). Network Interfaces also provide buffering at the interface with the network to improve performance.

In Xpipes, two separate Network Interfaces are defined. One Network Interface is the initiator NI, which connects to the master/processor core and the other is the target NI, which connects to the target slaves. Each master and slave device requires an NI of its type (initiator or target) to be attached to it. The interface between the IP cores and Network Interfaces is defined by the OCP 2.0 [34] specification, which supports features such as non-posted and posted writes (i.e. writes with or without response) and various types of burst transactions, including single request/multiple response bursts.

Xpipes employs dedicated Look-Up Tables at NIs, which specify the possible pre-defined paths for the packets to follow to the respective destinations. This reduces the complexity of the routing logic in the switches. Two different clock frequencies can be linked to Xpipes Network Interfaces: one is connected to the front-end of the Network Interface that implements the OCP protocol, while the other is connected to the back-end of the Network Interface that connects to the Xpipes NoC. It must be noted that the back-end clock (connected to the Xpipes NoC) must run at a frequency multiple of that of the front-end (Initiator) clock. This allows the NoC to run at a faster clock than the IP cores thus keeping transaction latencies low.

2.2.2 Switches

The medium of transportation of packets in the NoC architecture are the switches, which route packets from sources to destinations. Switches are fully parameterizable in the number of input and output ports. Switches can be connected arbitrarily and hence any topology, standard or custom can be configured. A crossbar is used to connect the input and output ports.

The switches are also equipped with an arbiter to resolve conflicts among packets from different sources, when they overlap in time and request access to the same output link. It is possible to implement either the round-robin or the fixed priority scheduling policy at the arbiter. It is also possible implement parallel links between switches, thus providing an inexpensive solution to handle congestion and maintain performance.

Switches are also equipped with input and output buffering solutions to lower congestion and improve performance. The buffering resources are instantiated depending on the desired flow control protocol. If credit-based flow control is chosen, only input buffering is mandatory. In this scenario, Xpipes optionally allows the designer to do completely without output buffers, reducing the traversal latency of a switch to a single clock cycle. Output buffers can still be deployed to decouple the propagation delays within the switch and along the downstream link; the downside is a second cycle of latency and additional area and power overhead.

2.2.3 Links

Links between switches and Network Interfaces form a critical part of NoCs. In Xpipes, links are further enhanced by supporting link pipelining, with logical buffers to reduce propagation delays. Xpipes implements latency insensitive operation using appropriate flow control protocols to make the link latency transparent to the logic, thereby enabling faster clock frequencies.

2.3 Xpipes Flow Control Protocols

Flow control allocates network resources to the packets traversing the network and provide solution to resource allocations and contentions. Flow control in NoCs is crucial, as it plays a decisive role in the determination of: (a) the number of buffering resources in the system: efficient flow control protocols will minimize the number of required buffers and their idling time, (b) the latency that packets incur while traversing the network, which is useful under heavy traffic conditions, where fast packet propagation with maximum resource utilization is key and (c) the degree of support for link pipelining and the associated delay overhead.

In Xpipes, three radically different flow control protocols have been implemented. They are:

- ACK/NACK, a retransmission-based flow control protocol where a copy of the transmitted flit is held in an output buffer until an ACK/NACK signal is received. If an ACK signal is received, the flit is deleted from the buffer and if a NACK signal is received the flit is re-transmitted.
- STALL/GO, a simple variant of credit-based flow control where a STALL is issued based on the status of the buffer downstream when there is no buffer space available, else a GO signal is issued, indicating availability of buffer space to accept the next transaction.

- T-Error, a complex timing-error-tolerant flow control scheme, that enhances performance at the cost of reliability.

Each of these offer different fault tolerance features at different performance/power/area trade-offs. STALL/GO assumes reliable flit delivery. T-Error provides partial support in the form of logic to detect timing errors in data transmission. ACK/NACK supports thorough fault detection and handling, using retransmissions in case of failures. ACK/NACK and STALL/GO flow control protocols are represented in Figure 2.2.

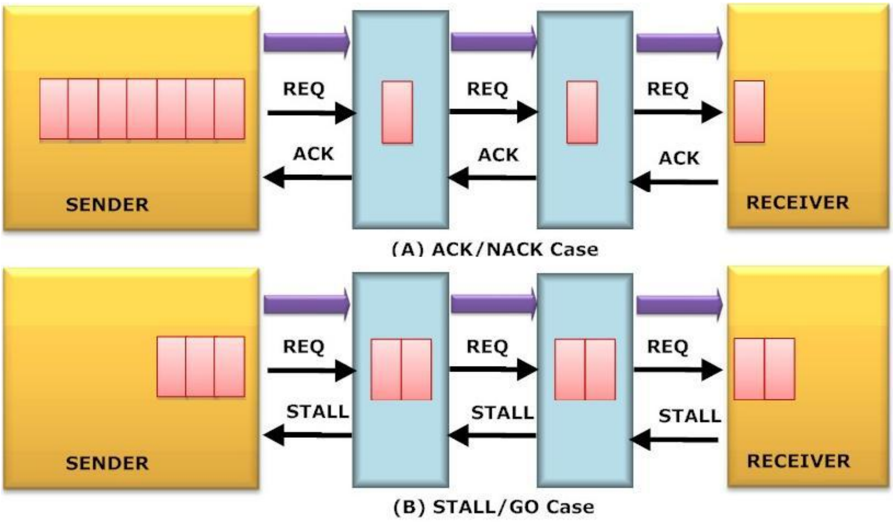


Figure 2.2: Xpipes pipelined link block diagram

In circuit-switched NoCs or those providing QoS guarantees [28], minimum buffering flow control can be used: a circuit is formed from source to destination nodes by means of resource reservation, over which data propagation occurs in a contention-free environment. Best-effort networks are normally purely packet switched and typically buffering increases the efficiency of flow control mechanisms.

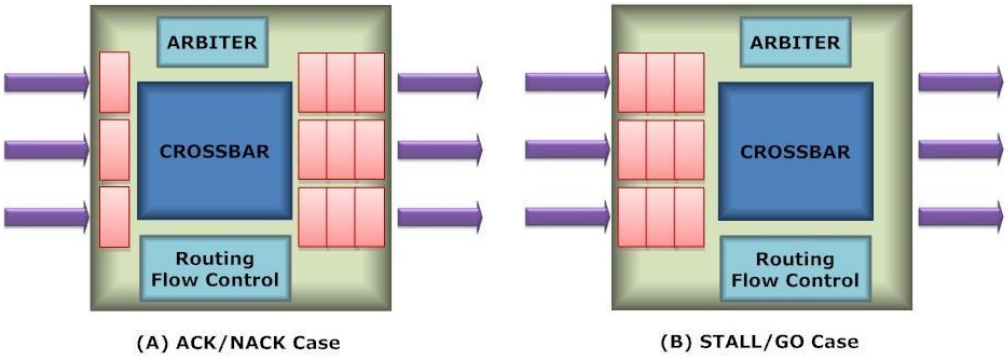


Figure 2.3: Buffering in Switches

The amount of buffering resources in the network depends on the target performance and on the implemented switching technique. The buffering in the switches when using ACK/NACK and STALL/GO flow control protocols is depicted in Figure 2.3. Switches need to hold entire packets when store-and-forward switching is chosen, but only flits when wormhole switching is used. By default, Xpipes uses wormhole and source routing, which reduces the amount of buffering required besides using STALL/GO flow control. Further details about the Xpipes flow control protocols, are presented in [12].

2.4 Xpipes Compiler

For an application-specific network on chip, there is a need to design network components (e.g. switches, links and network interfaces) with different configurations (e.g. I/Os, buffers) and interconnecting them with links supporting different bandwidths. This process requires significant design time and needs design verification of the network components for every NoC design.

The Xpipes Compiler [29] is employed to instantiate the different components of an NoC (routers, links, network interfaces) using the Xpipes library of SystemC macros, for a specific NoC topology. The Xpipes library comprises of high performance, low power parameterizable components that can be generated for a NoC, tailored to the specific communication needs of any given application. This helps the Xpipes Compiler in instantiating optimized NoCs, where significant improvements in area, power and latency are achieved in comparison to regular NoC architectures.

An overview of the SoC floorplan, including network interfaces, links and switches, clock speed, possible links and the number of pipeline stages for each link area specified as input to the Xpipes Compiler. Routing tables for the network interfaces are also specified. The tool uses the Xpipes SystemC library, which includes all switches, links and interfaces in different configurations and specifies their connectivity. The final topology is then compiled and simulated at the cycle-accurate and signal-accurate level and fed to back-end RTL synthesis tools for silicon implementation. Thus, an optimal custom network configuration is generated by the Xpipes Compiler based on the application's requirements and costs.

2.5 MPARM platform

MPARM [14], a SystemC simulation platform is developed at University of Bologna, to evaluate the performance of MPSoCs with cycle accuracy. MPARM can incorporate different platform variables, such as memory hierarchies, interconnects, IP core architectures, OSes, middleware libraries, etc., making it possible to study the macroscopic impact of small changes at the architectural or programming level. MPARM can include a large variety of IP cores, ranging from microprocessors to DSPs, from accelerators to VLIW blocks. It can also support extremely varied memory hierarchies, including caches, scratchpad memories, on-chip and off-chip SRAM and DRAM banks.

The MPARM platform can also run OS and middleware and real applications to most efficiently exploit the underlying architecture, besides supporting different communication and synchronization schemes, including message passing, DMA transfers, interrupts, semaphore polling, etc. MPARM is also an ideal platform to test our interconnect. It can support a wide range of system interconnects, including shared buses of several types, bridged and clustered buses, partial and full crossbars, up to NoCs.

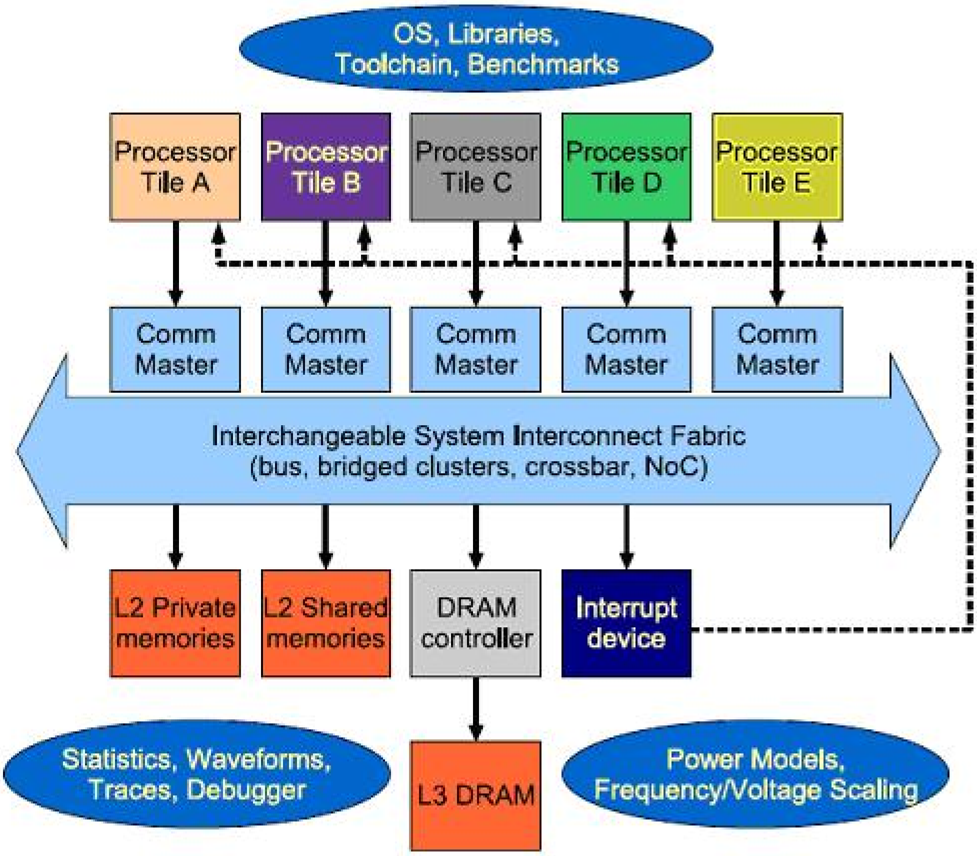


Figure 2.4: The MPARM SystemC virtual platform

The MPARM environment, as shown in Figure 2.4 (Courtesy: [12]), is designed to investigate the system level architecture of MPSoC platforms. To be able to fully assess system performance, a cycle-accurate modeling infrastructure is put into place. MPARM is a plug-and-play platform based upon the SystemC simulation engine, where multiple IP cores and interconnects can be freely mixed and composed. At its core, MPARM is a collection of component models, comprising processors, interconnects, memories and dedicated devices like DMA engines. The user can deploy different system configuration parameters by means of command line switches.

A thorough set of statistics, traces and waveforms can be collected to debug functional issues and analyze performance bottlenecks. MPARM features a choice of several IP cores to be used as system masters. These span over a range of architectures, that typically model pre-existing general purpose processors with little to no possibility of modifying the ISA and the architecture.

On top of the hardware platform, MPARM provides a port of the RTEMS [21] Operating System, offering good support for multiprocessing with efficient communication and synchronization primitives. Application code, can be easily compiled with standard GNU cross-compilers and ported to the platform.

MPARM also has special libraries for development and debugging of new applications and benchmarks. This is important for establishing a solid and flexible simulation environment. MPARM includes several benchmarks from domains such as telecommunications and multimedia, and libraries for synchronization and message passing.

Debug functions include a built-in debugger, which allows to set breakpoints, execute code step-by-step and inspect memory content; it is additionally capable of dumping the full internal status of the execution cores. Multiple communication and synchronization paradigms are possible in MPARM, including plain data sharing on a shared memory bank, message passing among scratch pad memory resources of each processor, interrupts and semaphore polling (if OS is not used for synchronization).

MPARM stimulates the communication subsystem with functional traffic generated by real applications running on top of real processors. This opens up the possibility for communication infrastructure exploration under real workloads and for the investigation of its impact on system performance at the highest level of accuracy.

2.6 Using Xpipes Compiler and MPARM

As indicated above, the Xpipes Compiler is used to instantiate network components (routers, links, network interfaces) with different configurations for a specific NoC topology, using the Xpipes library. When employing synthetic traffic generators, the Xpipes NoC is generated this way and used directly for testing and performance validation as will be shown in Chapter 3. When there is a need to use trace-based traffic generators, the MPARM simulation platform is used. The MPARM platform is used to run benchmark applications for different interconnects and obtain the traces. Then employing the methodology proposed in Chapter 4, we model the traces and derive the application schedule, to re-generate and port the application traces for validation of the designed NoC, which is generated by the Xpipes Compiler for that application.

3 Synthetic Traffic Modeling and Generation

3.1 Need for Traffic Models

The on-chip interconnection in a Multi Processor System on Chip has a significant impact on the overall performance of the system and this necessitates the need to analyze the same. The interconnect can span over a huge variety of architectures and topologies, ranging from traditional shared buses up to packet-switching Networks-on-Chip. To evaluate design choices for a particular interconnect, the MPSoC designer needs synthetic traffic models that are realistic and representative of real-world embedded applications, to verify and validate the interconnect's performance and suggest optimizations.

3.2 Modeling Traffic Injection

In an embedded environment, a traffic source such as an IP or a processor, generates data traffic either periodically or at irregular intervals. Hence, it becomes essential to characterize the traffic injection process (traffic arrival into the network), to effectively replicate the application traffic, wherein the variation in the inter-arrival/inter-injection times between two transactions becomes the most significant component. This variation in inter-injection times may be either correlated by certain standard probability distribution or be completely non-deterministic, depicting a random pattern (curve) when plotted in time.

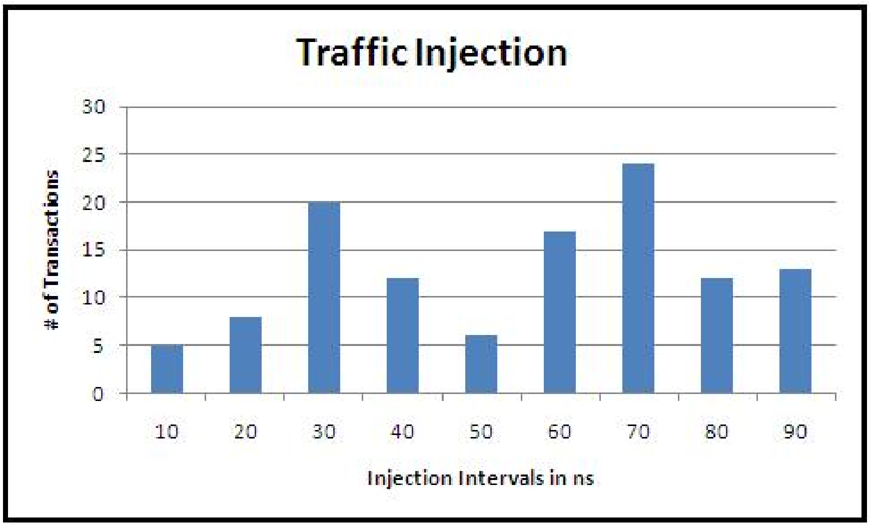


Figure 3.1: Traffic Injection Histogram

In the former scenario, these inter-injection times can be defined as randomly generated variables distributed in time, correlated to each other by a probability distribution function. In other words, the random variables (timings) generated by a given probability distribution function, in an unspecified order, will ascertain the inter-injection times in a traffic. The best way of representing such a behavior, is by plotting appropriate histograms, with the non-overlapping injection intervals on the X-axis and the number of transactions for the corresponding injection interval on the Y-axis, as shown in Figure 3.1. For traffic modeling and generation, the probability density functions of the distributions are employed to get the inter-injection times. This method is discussed in further detail in Section 3.4.

In the latter scenario, since the correlation in the inter-injection times is non-deterministic and when plotted in time represents a particular (random or familiar) pattern, there must be a way of modeling such a behavior in time as well. It must be noted that when representing such traffic shaping in time, using a histogram (frequency domain representation) is inappropriate, since the behavior is temporally and relatively defined and cannot be randomly generated. Hence, to maintain a similar temporal relevance, a time domain representation is certainly more suitable, with the transaction number on the X-axis and its corresponding injection-interval on the Y-axis, as shown in Figure 3.2. For traffic modeling and generation, a novel 'peaks and valleys' approach is suggested and later this model is employed to generate the appropriate inter-injection times. This method is further explained in Section 3.5.

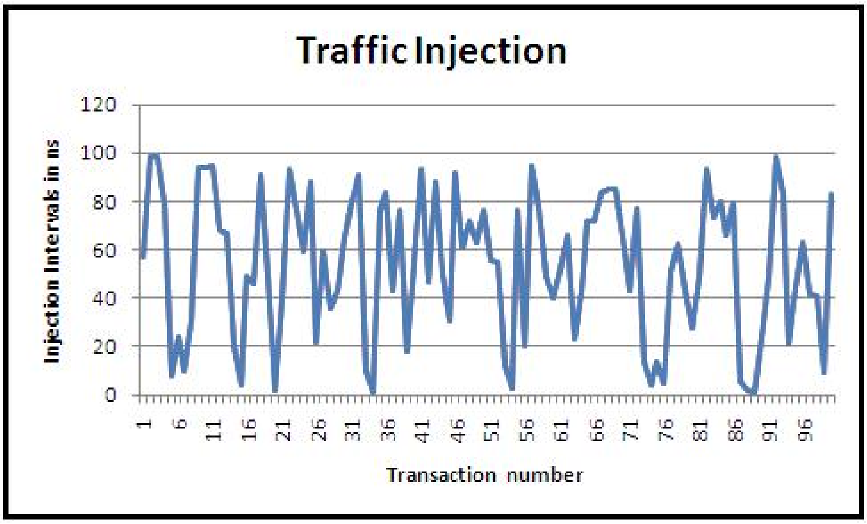


Figure 3.2: Traffic Injection Timeline

3.3 Modeling Synthetic Traffic

It must be noted that the primary purpose of using synthetic traffic generators which employ probability distributions or timing information (in the form of traffic patterns/curves) for traffic injection, is to speed up the validation process and generate flows to strain the interconnection network. It must also be noted that such distributions (observed or derived) assume a degree of correlation within the inter-injection intervals, which may not always be true in an SoC environment. Also the inherent probabilistic nature of the statistical approach itself, makes it less accurate, as each traffic generator injects traffic in complete isolation from every other. However, the simplicity and simulation speed of such stochastic models make them valuable during preliminary stages of NoC validation, but, since the characteristics (functionality and timing) of the IP core are not captured (due to lack of knowledge of the application/IP behavior), such models can only serve as a direction for analyzing and validating the performance of interconnect/NoC and not for its design optimization.

In our approach to generate synthetic traffic, as stated in the previous section, we use both standard probability distributions and traffic injection patterns (curves) to estimate the inter-injection times. The motivation for employing the former method, is that traffic behavior in certain applications are found to exhibit partial (or full) adherence to specific probability distributions. For instance, in variable-bit-rate video traffic [25], self-similar traffic pattern is observed due to longrange dependence. A heavytailed distribution such as Pareto that exhibits extreme variability, may lead to such longrange dependence and hence, self similar pattern in network traffic. Besides such specific probability distributions, we also employ certain standard probability distributions, such as Normal (Gaussian), Poisson and Exponential Distributions, to generate synthetic traffic to validate the interconnect performance. In case of the latter method, a novel ‘peaks and valleys’ approach is suggested, to model the random traffic patterns and to generate the appropriate inter-injection times.

Besides, when there is little IP information or knowledge of the application behavior, the best an interconnect validation infrastructure can do, is to model the traffic arrival/injection process on the basis of such distributions and models.

Another aspect very crucial to interconnect performance analysis and validation studies is efficient traffic management/scheduling by the traffic generator. Such traffic management/scheduling defines the spatial distribution of the traffic in the network. While the temporal distribution obtained using the probability distributions or injection patterns, determines how traffic is generated over time, the spatial distribution defines which master communicates with which slave at a particular instance in time. The significance of defining the spatial distribution of traffic is that, it helps in exploring different avenues for validation such as instances when the traffic is localized to a particular slave or evaluating hot-spot patterns in the network which can be useful in representing the application’s characteristics. The spatial distribution of traffic defines traffic distribution in amongst all slaves for each of the traffic generators.

When multiple masters simultaneously inject traffic into the network, this leads to contention and hence congestion, which adversely impacts its performance. A crude way to work around this issue, is to over-design the interconnect (NoC), such that despite the congestions and contentions, the throughput (bandwidth) and latency requirements (QoS) of the application are met.

It can be said, that to a good extent, the amount of over-design depends on the accuracy of the application traffic model in replicating the application behavior and the management/scheduling of traffic on the network, since together they dictates the temporal and spatial usage of the network resources and hence, the required over-design. In other words, arriving at an optimal network design depends on the efficiency of the traffic generator in mimicing the application and the effectiveness of the traffic management/scheduling policy of the traffic generator and its efficient scheduling of loads on different links in the network.

Having addressed the issue of temporal distribution of traffic using probability distributions/ traffic patterns, traffic management/scheduling becomes the key in effective network validation. Hence, it becomes essential that every traffic generator while scheduling the traffic, gives appropriate priorities to its connected slaves, based on individual instantaneous and overall average injection bandwidth requirements. Towards this, an approach to dynamically re-schedule transactions across the slaves is suggested and described in detail, later in this chapter.

In a nutshell, this chapter suggests the following solutions for a successful simulation study:

- A traffic generator employing probability distributions to get injection intervals.
- A traffic generator using the ‘peaks and valleys’ approach to model traffic patterns.
- An efficient traffic management/scheduling scheme for appropriate load distribution and effective validation.

3.4 Modeling Traffic using Probability Distributions

While employing standard probability distributions to characterize the traffic injection process (temporal behavior), the inter-injection times can be estimated using the corresponding probability density function (pdf). In general, employing such probability density functions, is made feasible by plotting appropriate histograms, as suggested in Section 3.2.

The probability distributions discussed in the previous section, are analyzed in this section and are employed by the traffic model to determine inter-injection intervals. Their corresponding continuous probability density functions and the generated discrete representations are depicted in Figure 3.3.

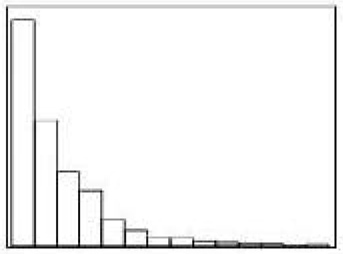
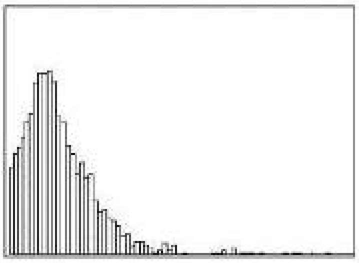
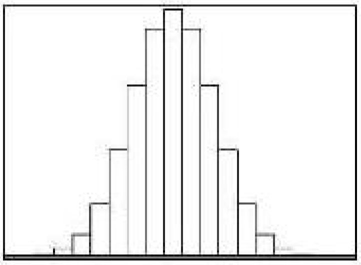
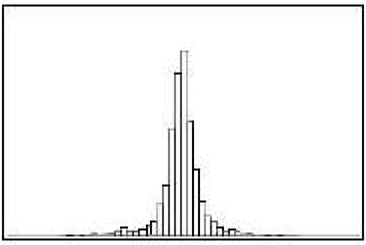
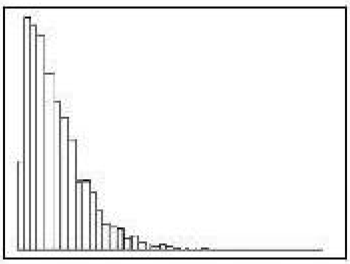
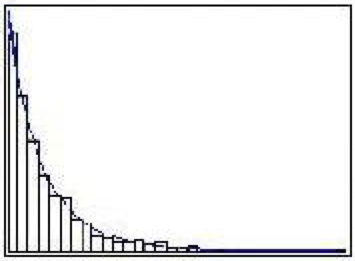
PROBABILITY DISTRIBUTION	PROBABILITY DENSITY FUNCTION	GENERATED HISTOGRAM
EXPONENTIAL DISTRIBUTION	$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}$	
POISSON DISTRIBUTION	$f(k;\lambda t) = \frac{e^{-\lambda t}(\lambda t)^k}{k!}$	
NORMAL DISTRIBUTION	$f(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}$	
CAUCHY DISTRIBUTION	$f(x;x_0,\gamma) = \frac{1}{\pi}\left[\frac{\gamma}{(x-x_0)^2+\gamma^2}\right]$	
WEIBULL DISTRIBUTION	$f(x;\lambda,k) = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} \\ 0 \end{cases}$	
PARETO DISTRIBUTION	$f_X(x) = \begin{cases} \alpha \frac{x_{\mathrm{m}}^{\alpha}}{x^{\alpha+1}} & \text{for } x > x_{\mathrm{m}} \\ 0 & \text{for } x < x_{\mathrm{m}} \end{cases}$	

Figure 3.3: Probability Distributions

The different probability distributions used to generate synthetic traffic include the following:

(a) **Exponential Distribution** - In an exponential distribution [4], the inter-injection times represent a Poisson process, i.e. a process in which events occur at a constant average rate, continuously and independently.

(b) **Poisson Distribution** - In a Poisson distribution [6], the injection intervals are calculated using the probability of the number of packets to be injected in that fixed period of time independently of the time since the last event.

(c) **Normal (Gaussian) Distribution** - In a Normal (Gaussian) distribution [5], most of the inter-injection intervals cluster around the mean or average. The probability density function has its peak at the mean and is known as the Gaussian (or bell) curve.

(d) **Pareto Distribution** - The Pareto distribution [7], is a heavytailed distribution that exhibits extreme variability and can be used to represent self similar pattern in traffic, as shown in [25].

(e) **Cauchy Distribution** - The Cauchy distribution [8], is observed in GPRS networks as shown in [20], in text traffic, where the maximum number of transactions are injected at the mean interval.

(f) **Weibull Distribution** - The Weibull distribution [9], is used to represent the ON/OFF process in a bursty VoIP traffic, as shown in [19] and [36].

(g) **Combination of Probability Distributions** - Besides using different probability distributions for the entire duration of the simulation, it may be useful to support a configurable combination of such probability distributions, to generate multi-dimensional traffic. This feature is incorporated in the proposed synthetic traffic generator and a sample histogram is presented in Figure 3.4.

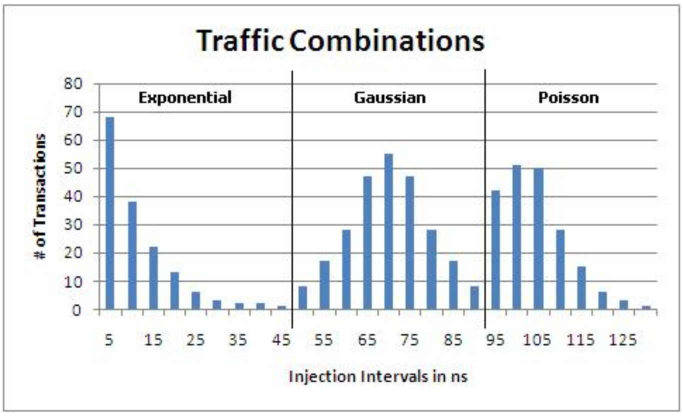


Figure 3.4: Combination of Probability Distributions

As can be observed in Figure 3.4, a combination of different probability distributions including as Exponential, Gaussian and Poisson can also be employed. Besides these special distributions (or combination of distributions), it is also possible to inject traffic uniformly in time, where all injection intervals are of the same length/duration, for instance to represent a uniformly generated sequence of transactions.

3.5 Modeling Traffic using Traffic Patterns

Modeling traffic injection behavior, that exhibits non-deterministic patterns (curves) in time and does not follow a standard probability distribution, calls for a time-domain based approach, that preserves the relative temporal correlation. In comparison to the probability-distributions based approach, where histograms are used to indicate possible injection intervals, for non-deterministic traffic injection patterns/shapes, it becomes essential to preserve the relative temporal spacing between successive injections for all transactions to maintain transaction ordering and sequence.

When employing such detailed patterns of injection intervals for all transactions, it becomes crucial to model such patterns to speed up traffic generation, while maintaining accuracy (in terms of adherence to the original pattern) as well. A naive solution to this can be, sampling of injection intervals using a time-window based approach, although, that would have a noticeable impact on the accuracy of traffic regeneration. Instead, a novel and simple 'peaks and valleys' approach is suggested for this purpose, where the key effort in modeling the pattern, is to store all the local peaks and local valleys along the injection pattern curve, as depicted in Figure 3.5. In other words, when plotting the traffic injection pattern in time (i.e. injection intervals between successive transactions), one needs to identify all the local peaks and local valleys, to re-generate a similar looking curve or injection pattern. This approach would significantly reduce the amount of information that needs to be stored for generating such traffic patterns.

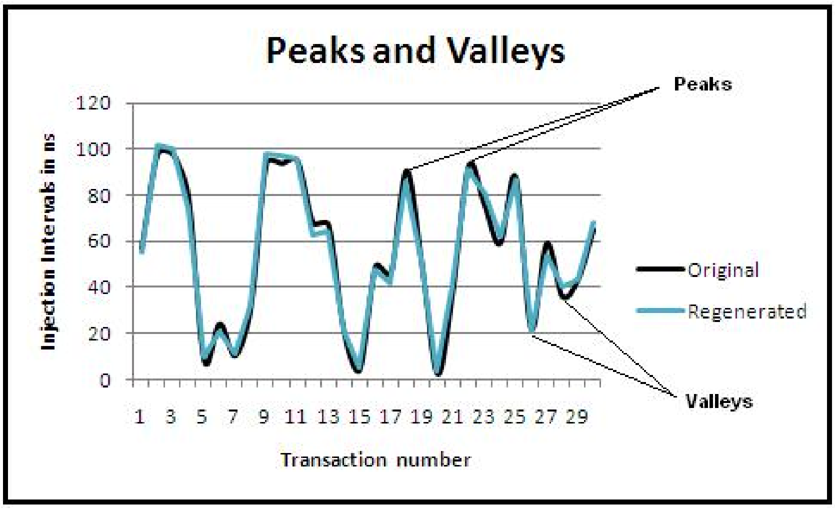
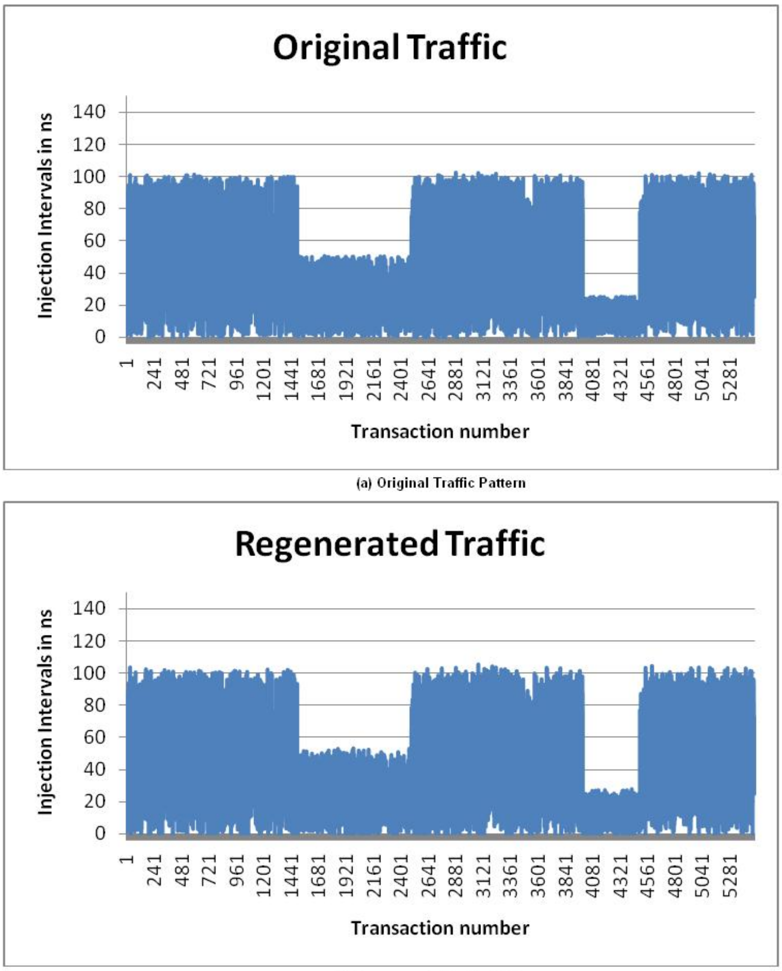


Figure 3.5: Peaks and Valleys Approach

Once, all the local peaks and valleys are obtained, this information can then be used to re-generate the curve (traffic injection pattern), by employing appropriate exponential curves between peaks and subsequent valleys and reverse exponential curves between valleys and subsequent peaks. Employing such a simple modeling technique by using exponential curves between peaks and valleys make sense, since it is safe to assume that all injection intervals between a local peak and a subsequent local valley, will always decrease or stay stable and never increase. The same logic can be applied to the usage of reverse-exponential curves between valleys and subsequent peaks, where injection intervals tend to keep increasing.

To observe the accuracy of this modeling, we use a synthetic traffic trace to get a reference injection pattern and model and re-generate the same using the 'peaks and valleys' approach. A comparison of the original and re-generated plots of the injection intervals is depicted in Figure 3.6 (a) and (b).



(a) Original Traffic Pattern

(b) Re-generated Traffic Pattern

Figure 3.6: Application Traffic (Original and Regenerated)

As can be seen in the figure, there is only a marginal difference between the original and the re-generated curves. The efficiency of this method can be clearly observed, since this approach re-generated a very similar pattern (curve) and at the same time, significantly reduced the memory required to store the curve info to about 15% of the original (though this gain cannot be assured on all curves).

Another interesting, yet not so-successful extension to this approach was tested, where the obtained ‘peaks and valleys’ curve was subjected to another iteration of the ‘peaks and valleys’ optimization. In this approach, all the local peaks were considered together in a ‘peaks only’ curve and all the local valleys in a ‘valleys only’ curve. These two curves where then subjected to another iteration of the ‘peaks and valleys’ approach, and all the peaks and valleys in both the ‘peaks only’ and ‘valleys only’ curves were stored. Again, as specified earlier, using the exponential and reverse-exponential models, the ‘peaks only’ and ‘valleys only’ curves were re-generated and were further used to re-generate the original traffic pattern. However, this approach was not so successful, since it was off the mark (from the original traffic injection intervals), by plus or minus 20%, and is hence not reported with results in this report.

3.6 Traffic Management/Scheduling Scheme

As mentioned in Section 3.3, efficient traffic management/scheduling by the traffic generators is also extremely crucial for interconnect/NoC performance validation. Traffic management or scheduling by the traffic generator defines the spatial distribution of the traffic on the network and helps in validating the network’s performance in different scenarios such as when the traffic is localized to a particular slave or hot-spots exist in the network. It must be noted, that this traffic management method implemented by the traffic generator is very different from the traffic management policies implemented by the network on chip. While the former gives a spatial distribution of traffic across the NoC, the latter addresses network issues such as handling flow control, queuing of transactions or traffic regulation.

For efficient traffic management, an approach for enabling dynamic re-scheduling of transactions is suggested in this section. The rationale behind using a traffic management/scheduling system with dynamic re-scheduling, is that, since the instantaneous bandwidth (throughput) requirements of the slaves keep varying, to maximize the link utilization, the priorities need to keep changing online as well. Such changing priorities rule out using a uniform random spatial traffic distribution. From the implementation point of view, in order to perform dynamic re-scheduling, one needs to monitor the injected bandwidth from a traffic generator to each of the slaves and compare it against the expected instantaneous injection bandwidth. To check for adherence to the inter-injection intervals, a Bandwidth (throughput) Satisfaction Monitor is proposed, which monitors the injected transactions and evaluates overall average bandwidth (throughput) satisfaction levels. It must be noted, that this metric gives an unbiased comparison of the status of all links, by normalizing the usage against the individual bandwidth requirements.

Such dynamic re-scheduling of transactions across slaves would help in performance analysis of the network under different traffic conditions. One possible condition, could be when the traffic exhibits spatial locality (many masters communicating to the same slave) or a hot-spot traffic pattern. This can be handled by stressing the busiest links (hot-spots) to the limits determined by the bandwidth requirements. Another possible condition, could be a possible yet realistic worst-case traffic injection, which will help test the interconnect's performance and robustness. This can be handled by stressing all the links heavily and proportionally. Such performance analyses of the network will help in validating the network's performance and also in figuring out the optimal (over)design for the target application.

Existing simulation studies employing synthetic simulation traffic models, targeted towards NoC design, build either a worst-case or an optimistic traffic model, which unfortunately, have highly over or under-specified constraints often leading to awkward over or under-design of the NoC.

An obvious improvement to such worst-case or optimistic models, is using online re-scheduling schemes, which to a certain extent make sure that the system is not significantly under or over-designed, though under acceptable performance penalties. The suggested solution of equipping the synthetic traffic generator with a bandwidth (throughput) satisfaction monitor, coupled with online re-scheduling algorithms, is an effort in this direction and aids in reducing the over-design expected with existing traffic generators.

In order to perform the analyses suggested above, appropriate dynamic scheduling algorithms are employed as suggested below:

(a) To perform the first analysis, an adaptation of the 'maximum throughput' scheduling algorithm [11] is employed, with a view to maximize the total throughput of the network. This algorithm would prioritize data injection to the links with high bandwidth demands (hot-spots) and stress them more than the others thereby increasing the overall system throughput.

(b)To perform the second analysis, an adaptation of the 'weighted fairness' scheduling algorithm [10] is employed to appropriate priorities to links based on fair (weighted) sharing of load across all links, to check for robustness of the NoC. This algorithm would give fair (weighted) priorities to all links, by constantly re-evaluating priorities based on the bandwidth (throughput) satisfaction levels of all links since they last received transactions.

The bandwidth (throughput) satisfaction monitor plays a significant role in facilitating the evaluation of the cost-functions associated with the two online scheduling algorithms.

3.6.1 Maximum Throughput Scheduling

The maximum throughput algorithm [11] is adapted to incorporate a cost function that determines the expected throughput gains by scheduling transactions on particular links. It employs the logic that a slave/link needing data at higher bandwidth (to exhibit hot-spots or spatial locality) should get more priority over the other slaves with lower bandwidth requirements, provided this slave/link has a bandwidth (throughput) satisfaction level of less than 100%. Hence, this policy of dynamic re-scheduling, checks for bandwidth (throughput) satisfaction levels across all slaves/associated links and then amongst those with less than 100% bandwidth (throughput) satisfaction, selects the one with highest bandwidth requirement at that instant in time and sends the next transaction to it. This method gives priorities to high bandwidth links with hot-spots and while ensuring maximum network throughput, checks for robustness of specific links.

3.6.2 Weighted Fairness Scheduling

The drawback of the maximum throughput algorithm is that, a comprehensive view of the network's performance is not observed. In order to stress all the links to their limits, to analyze the entire network's performance under realistic worst-case scenario's, by being fair to all links (though under lower network throughput), an adaptation of the proportional (weighted) fairness algorithm [10] is employed. This algorithm re-evaluates scheduling priorities to users that have achieved lowest bandwidth (throughput) satisfaction levels since they became active or were last answered. The cost function used in the proportional (weighted) fairness algorithm calculates the cost per data bit of data flow and in effect estimates the expected loss of not scheduling traffic on a particular link. Using this cost function to re-evaluate priorities dynamically, would lead to higher bandwidth (throughput) satisfaction levels of all links, thus achieving a realistic depiction of a scenario, when all links are heavily loaded.

3.6.3 Analyzing Scheduling Impact

To evaluate the proposed solutions, we employ the topology suggested in Section 2.6 and inject synthetic traffic into the network with the pre-defined characteristics. Over several runs of the simulations, we observed the following:

(a) The traffic generator employing an adaptation of the 'maximum throughput' scheduling algorithm, was able to inject up to a maximum of around 92% of the required bandwidth (throughput) on the link with the highest bandwidth demand. On the link with the lowest bandwidth (throughput) requirement, it was able to inject at approximately 80% of the required bandwidth (throughput).

(b) The traffic generator employing an adaptation of the 'weighted fairness' scheduling algorithm, was able to inject up to a maximum of around 87% of the required bandwidth on the link with the highest bandwidth (throughput) demand. On the link with the lowest bandwidth (throughput) requirement, it was able to inject at approximately 84% of the required bandwidth.

The exact bandwidth (throughput) satisfaction levels on all links from Master 0, while employing the modified online re-scheduling algorithms is shown in Figure 3.7. As expected, the 'maximum throughput' algorithm injects stresses particular high demand links to the possible limits (under conditions of congestion), while the 'weighted fairness' fairly distributes the traffic over all links.

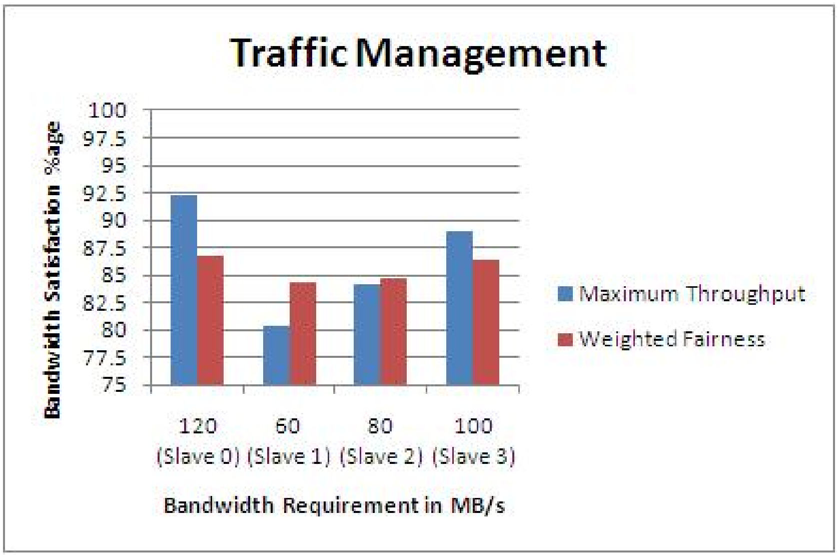


Figure 3.7: Efficient Traffic Management Schemes

These bandwidth injection values, give an idea of the required over-design of the links, based on how real-time congestion impacts the network's performance. It must be noted that the proposed solutions adhere to the traffic injection pattern as specified by the probability distributions, to the extent that the injections intervals are at least as long as the ones specified and the traffic distribution and characteristics are maintained as indicated by the user/application. This is to highlight that the injection intervals were not compromised upon, by simply allowing traffic to overflow, since that would provide an incorrect validation of the NoC.

3.7 Challenges in Synthetic Traffic Generation

In the proposed synthetic traffic generation, the IP traffic injection behavior is statistically represented by means of Exponential, Normal, Poisson or relevant probability distributions or by models of non-deterministic traffic patterns obtained by using the 'peaks and valleys' approach. It must also be noted, that the inter-injection times obtained from these distributions and models,indirectly represent the instantaneous injection bandwidth requirements and the traffic generator must check for adherence to these bandwidth (throughput) and latency requirements at all times. The traffic generator must also take into account the nature of MPSoC traffic such as short-data access, bursty, etc., while employing the injection rates governed by these distributions/models.

The traffic generator must also address the following set of issues:

(a) **Handling Transactions**

It is imperative that the generated traffic is representative of real IP core in terms of the characteristics and the mix of the transactions injected into the network. The traffic generator must maintain multiple traffic threads and combinations which can be invoked/employed based on the expected traffic characteristics. For instance, using traffic information such as ‘x’ number or percentage of transactions are 2 word burst Reads etc.

It is also important to keep in mind, that the transactions such as Reads and Non-posted Writes expect a response and the traffic generator must block subsequent transaction injection until such a response is received.

(b) **Injection Intervals**

The traffic generator must be capable of issuing conditional sequences of traffic composed of different communication transactions separated by co-related or independent wait-periods as indicated by the probability distributions (or derived models for non-deterministic distributions), thus emulating a typical (blocking) processor.

(c) **Buffering to avoid Data loss**

Once it is established that the traffic generator can replicate a blocking IP core and can inject different combinations of the transactions into the network separated by varying wait-periods, it may be claimed that the traffic generator can emulate an IP with similar properties and features. However, the real test of a traffic generator is when such an IP/Traffic generator is plugged into the network and is made to work in real-time under the restrictions of the network in the presence of congestions and contentions, which do not help in the ‘ideal’ working of the traffic generator. A typical processor in such a situation would halt injection of data into the network (and buffer it instead) to avoid data loss. When designing a traffic generator as a replacement of such a processor, it must be kept in mind that adequate buffering of transactions is provided to avoid loss of data (due to congestion in the network) whilst the processor is able to inject the transactions at intervals as closely as possible to the ones indicated by the probability distributions.

Given the fact that this traffic generator only emulates a processor and does not replicate its exact architecture, it gives us enough scope for handling the buffering issues. For instance, instead of storing entire transactions, only possible transaction types are stored in the transaction buffer, which makes it defined and limited. However, in order to avoid the network’s influence on traffic injection, theoretically, an indefinitely long outstanding request buffer is employed at the output of the traffic generator. This buffer holds the transaction injection requests till the master receives a response of its previous injected transaction in the form of a ‘Send Next’ signal.

3.8 Synthetic Traffic Generator Architecture

The architecture of the synthetic traffic generator is developed taking into account all the requirements, restrictions, traffic management schemes and traffic injection methods specified in this chapter and is designed to be robust and IP protocol independent.

The traffic characteristics are obtained from the user in terms of the bursts and transactions supported and their distribution and composition in the traffic. Other details including the address space of the masters and slaves are also obtained from the user. This information is used to setup the traffic injection module, which includes separate queues (transaction buffers) of traffic from the particular master traffic generator to all the connected slaves, that hold the possible transactions types. The injection intervals characterizing the temporal behavior of the traffic, are specified through an input file which is generated either by employing the probability distributions or by the 'peaks and valleys' approach for traffic pattern modeling. Average bandwidth requirements for all slaves are also obtained as input from the user, and using both the injection intervals and the average bandwidth requirements, dynamic throughput requirement estimates are calculated. This is forwarded as input to the Bandwidth (Throughput) Satisfaction Monitor and Slave Scheduler module. The former module captures injected transactions, which serve as the subsequent control input, at the output of the traffic generator and reports the same to the latter. The latter in turn, employs an adaptation of either the 'maximum throughput' algorithm or the 'weighted fairness' algorithm and using the dynamic throughout requirements, schedules the transactions across the slaves, thus defining the spatial distribution of traffic.

The slave scheduler module employs the dynamic bandwidth (throughput) requirements of the individual slaves, checks out the current bandwidth satisfaction metrics, re-calculates appropriate priorities using the re-scheduling algorithms and schedules the next transaction for injection at the appropriate time, using the injection intervals. This triggers the transaction selector to select a transaction type from one of the transaction queues, with the help of the randomizer and load the appropriate transaction into the outstanding transaction request buffer. This buffer is used to control the injection of the transaction into the network based on the status of the network and the response of the previous injected transaction. As soon as it gets a go-ahead (Send Next) to inject the next transaction, it injects the transaction at head of the outstanding transaction requests queue into the network, while this is monitored by the bandwidth satisfaction monitor. As stated before, for simulation purposes, in order to avoid the network's influence on traffic injection, this outstanding request buffer is indefinitely long. The FSM defined outside the traffic generator, acts as a middle-man between a Network Interface (which may implement any standard protocol) and the protocol-independent traffic generator and is used to translate the traffic generator's output to the description of the protocol. It is designed to keep track of the status of the transactions and the response from the network, for injecting the next transaction and converting the output of the traffic generator to OCP 2.0 protocol. The architecture of the synthetic traffic generator as described above, is depicted in Figure 3.8.

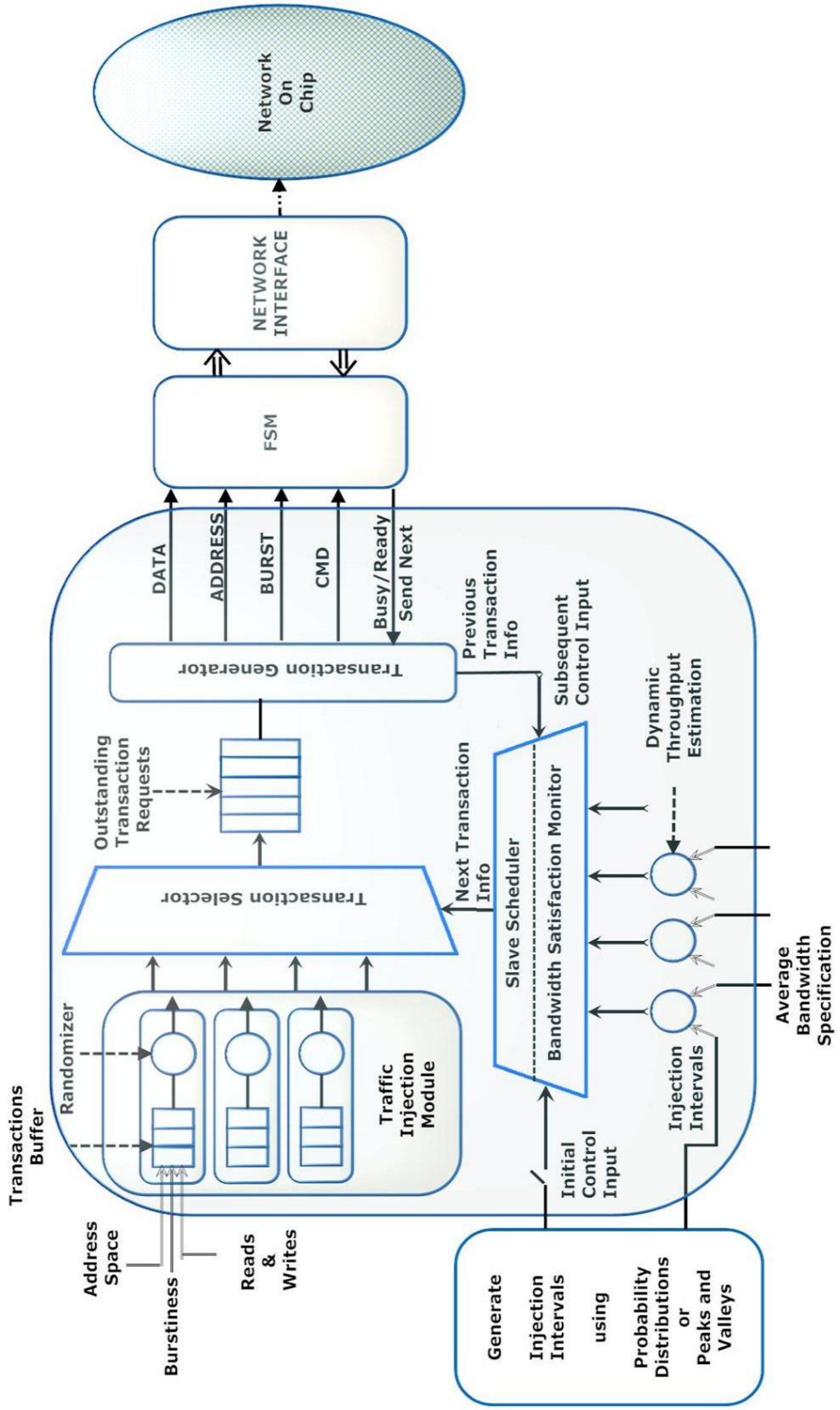


Figure 3.8: Synthetic Traffic Generator Architecture

Application Trace Modeling and Regeneration 4

4.1 Why model application traces?

The most important aspect of traffic generation is that the generated traffic should be a realistic representation of real-world embedded applications and hence, modeling application traces for traffic generation makes sense. In developing such a traffic model, there is a need to address the random IP requests for network resources, interspersed by randomly varying wait-periods. In short, traffic modeling must be able to re-generate the chaos in the network due the randomness, in the traffic generated by IPs.

One method of modeling the traces, can be by capturing the correlation in the injection times between successive transactions and then analytically modeling these inter-injection times with known probability distributions, besides storing information about the transaction types. However, this approach will be effective only for a few applications and simulation platform/interconnect architecture and cannot be generalized to all cases.

Therefore, it is suggested to use the given application traces (obtained from a simulator or a real-system with an existing interconnect) and model them with support for future porting and re-generation, on different platforms for different interconnects. For such modeling and porting of the trace, it becomes essential to derive the application's realistic flow and schedule which can help in reproducing complex dependencies and timing-sensitive events such as synchronization. Such a modeling must also ensure that the re-generated traffic adheres to the bandwidth and latency requirements, relative temporal behavior of the transactions, application schedule and transaction ordering and thereby, being a realistic representation of an application.

4.2 Issues in Modeling Traces

The modeling of IP traces can be handled at varying levels of complexities. At the most basic level, a trace with timestamps and inter-injection timings can be collected from the reference system and then be independently replayed. This approach is clearly inadequate due to the following reasons:

(a) When collecting traces from the reference system, the timings obtained include the delays associated with the base interconnect employed in the reference system, which may not reflect in the NoC being validated. This necessitates the need to filter out the base interconnect delays and employ the IP processing times alone as depicted in Figure 4.1.

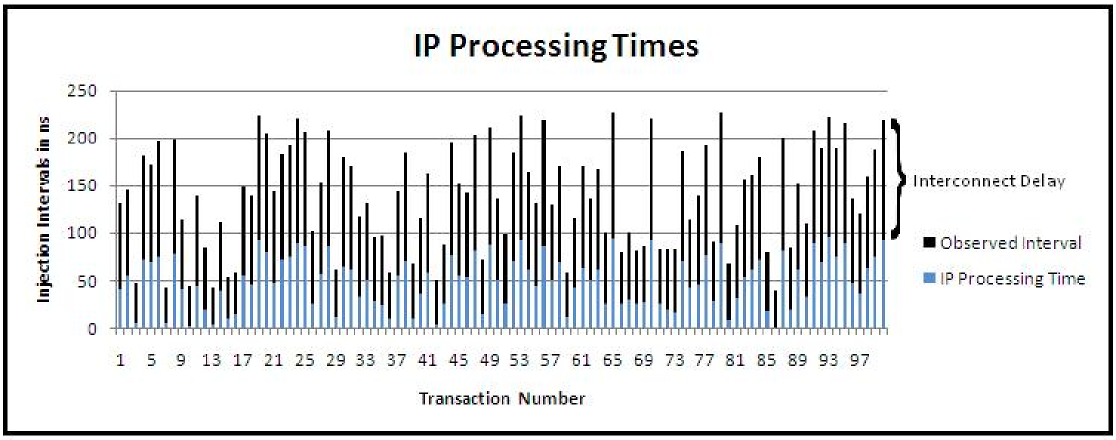


Figure 4.1: IP processing times and Interconnect Delays

As can be observed in the figure above, the reference interconnect delay is reflected on the inter-injection intervals obtained from a reference trace. This needs to be filtered out and only the IP processing delays (indicated in blue) must be employed.

(b) When observing the transaction injection times from all the masters in a global timescale, for an application comprising of cross-IP dependencies and timing-sensitive events such as synchronization, it is easy to incorrectly assume that a certain set of transactions across masters may be dependent on another, as depicted in Figure 4.2.

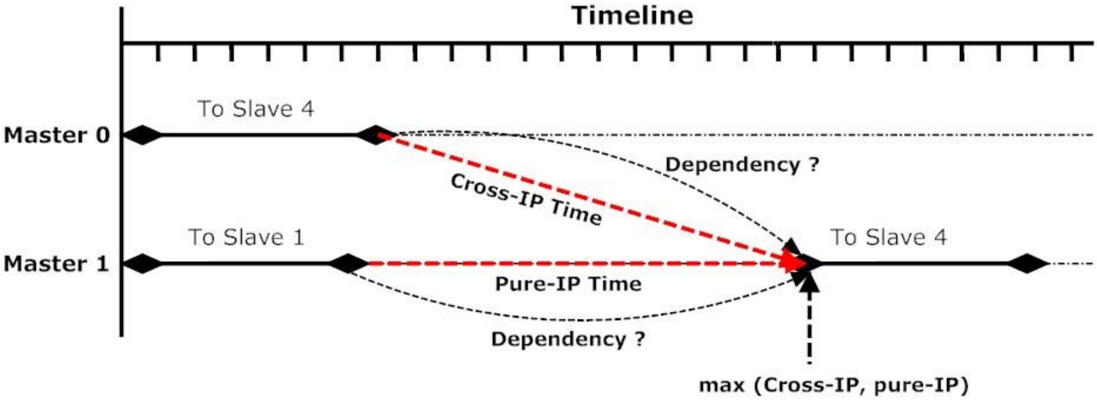


Figure 4.2: Dependencies between transactions

However, for re-generating a traffic pattern that gives an accurate representation of the application, such incorrect assumptions must be avoided. Hence, it becomes necessary to understand the application schedule and data dependencies across transactions and only then employ its schedule and transaction ordering information, to re-generate traffic that has similar temporal behavior of the transactions as the original application.

As can be seen in figure 4.2, when observing all the transactions on a global timescale, it becomes almost impossible to determine the dependencies across transactions and any false assumption of dependencies may impact the transaction ordering and application schedule and hence, lead to incorrect analysis.

As depicted in figure 4.2, transaction 2 from Master 1 to Slave 4 is injected after the responses for transaction 1 from Master 1 to Slave 1 and transaction 1 from Master 0 to Slave 4 are received. This gives rise to the confusion in assuming dependencies between transaction 2 for Master 1 and transactions 1 of Masters 0 and 1. It is nearly impossible to determine this dependency merely using information from the traces and hence calls for resolving the same at run-time.

In the figure, it must be noted that dependencies between transactions generated from the same master are characterized by pure-IP processing times, while those between transactions generated from different masters to the same slaves are characterized by cross-IP processing times.

To handle these issues in porting a trace from the reference system to the one being validated, a method for deriving an application's approximate static schedule, along with extracting IP processing times, from the reference trace is suggested. Solutions for effective modeling and porting of the traces, are addressed in detail in this chapter.

4.3 Trace Modeling Methodology

When the inter-injection times from a reference trace are considered, they do not reflect the IP processing times alone, instead they also add up the latencies associated with the base interconnect employed in the reference system, as indicated before. This factor is unwelcome, especially when there is a need to model only the inter-injection IP processing times. This implicitly means that we first have to filter out the base interconnect delays and employ the IP processing times, to analyze the behavior and latencies associated with the NoC being validated.

A deeper look into the transaction request-response times in the reference trace on a global time-scale, gives rise to the assumptions that a transaction (or a sequence of transactions) may be dependent on another (issued by the same master or by another) and can be issued only after a certain delay (pure-IP time or cross-IP time) after the completion of the previous sequence of transactions.

However, this assumption may be incorrect, since the observation may be due to congestion on the reference interconnect, thus hiding the actual dependencies in the application. However, resolving this dependency by merely employing the application's traces, is not be easy and hence, there is a need to arrive at a method to identify the application's approximate static schedule from the reference trace and thereby, address the issue of identifying true dependencies.

4.3.1 Estimating IP processing times

In order to successfully identify IP processing/wait times from the reference trace, one must read timestamps both for injections of processor-generated transactions and for their corresponding responses. The essential point is to capture the delay between the response for a transaction and the injection of the subsequent transaction, which reflects the IP processing times (within the same master or across masters). This can then be modeled by explicit idle waits in the traffic generator, while the network latencies will depend on the NoC model being validated.

Another way to obtain IP times, is when the reference simulation is performed in the presence of an ideal interconnect, which has zero or constant delay (in case of functional simulators).

4.3.2 Deriving Application's Approximate Static Schedule

In order to derive an approximate static schedule of a given application, from its reference trace, the following approach is suggested:

(a) The injection and response times of all the transactions in the system recorded at the masters, are observed together in a global timescale.

(b) Transactions generated from the same processor are considered to be dependent on each other, if the pure-IP processing/wait time between them is more than one clock cycle, else they are considered to be a part of a sequence of transactions. Subsequently, such a sequence of similar transactions from the given master to the same slave are clubbed together as one transaction set.

(c) Using the timescale analysis in step (a), a dependency between a transaction 'x' from master 'i' and transaction 'y' from master 'j' is assumed, if 'x' and 'y' refer to the same slave/destination and the response for transaction 'x' is received by master 'i' before transaction 'y' is issued by master 'j'. In such a case, the IP time for 'y' is expressed as the maximum of the cross-IP processing time (between reception of response for 'x' at 'i' and injection of 'y' and 'j') and the pure-IP processing time (between the response for 'y-1' and injection of 'y' at 'j') [max(cross-IP, pure-IP)]. However, this dependency is not assumed, if under similar conditions, response for 'x' is received by 'i' before the response for 'y-1' is received by 'j' and in this case, the IP time for 'y' is expressed as the pure-IP time from 'j'.

(d) Using the analysis in steps (a), (b) and (c), all transactions (or transaction sets) are allotted global sequence numbers, idle wait times (IP processing times) and dependency information.

It must be noted that by using the proposed heuristics (especially steps (c) and (d)), all existing data dependencies are recorded by default, although some dependencies may also be falsely assumed.

If the assumptions of dependencies are accurate, they are reflected when re-generating the traffic. However, if these assumptions are incorrect (falsely assumed cross-IP dependencies), appropriate measures must be taken to make sure that they are not used. To solve this issue, lets analyze the two circumstances under which the data dependencies may be falsely assumed and provide solutions to handle them:

The first scenario is when unwarranted delays (due to congestion on the reference interconnect) may incorrectly project a particular transaction ordering in the reference trace and hence, may lead to incorrect assumption of dependencies. As a solution, when porting such reference traces to another interconnect, in order to maintain transaction ordering, it is suggested to evaluate the dependencies at run-time and not when modeling the traces. In other words, for dependencies that are unclear while modeling the traces, for instance, when cross-dependencies exist, instead of trying to resolve the dependencies then and there, this information can be forwarded to a schedule manager, which at run-time can re-evaluate the dependency based on the responses from the NoC.

To support this solution and to provide the schedule manager with sufficient information for resolving such dependencies at run-time, the max(cross-IP, pure-IP) delay function suggested in step(c) can be used, which accommodates both the expected cross-IP and pure-IP processing times. By employing this max() delay function, if the cross-IP dependencies are true, they would be used as wait times for subsequent transaction injection. If the cross-IP dependencies are falsely assumed, the pure-IP times would be greater than the cross-IP times on the NoC and hence, the pure-IP times will be used to determine the next transaction injection.

It must be noted, that by using this max(cross-IP, pure-IP) delay function, there exists little scope for incorrect transaction ordering and execution, since, the next transaction is not issued unless both the previous transactions get their respective responses. This method also assures, that in case of synchronization events or cross-IP dependencies across masters, the dependencies are not over-looked and are respected while maintaining the transaction ordering.

The second scenario is when, the very existence of certain transactions is based on the performance of interconnect. These may refer to transactions, which depend on how long, another set of transactions take for completion, when using a given interconnect. In other words, a sequence of transactions may take longer time for completion on one interconnect when compared to the other and when the duration of the execution of those transactions, define the number of transactions of another type, this number would vary based on the interconnect being used. Hence, when porting from one interconnect to another, merely using the same number of transactions would carry no semantic meaning and hence, such transactions must be uniquely defined and addressed.

For instance, in a synchronization process, when a master is polling a semaphore slave to check for the status of the semaphore variable, the number of transactions to the slave, depends on how long another master takes to finish its operation and unlocks the semaphore variable. This duration depends on the reference interconnect's performance and the polling master generates an appropriate number of transactions that would not necessarily reflect on the new interconnect. Although the transaction ordering and dependencies are respected by the aforementioned method, the scenario of handling time(duration) dependent number of transactions, can be addressed only by using the application's original dynamic schedule, which simply cannot be derived by employing the application traces and will need application schedule information from the user. Such cases (scheduling with user assistance) are discussed with implementation details in the next sub-section.

However, it must be noted, that the previously suggested method of deriving the application's static schedule, gives an approximate yet almost accurate estimation of application's expected behavior and holds its share of significance in modeling traces. It must also be noted that the suggested method is more generic than the one suggested in [31] due to its ability to automatically detect dependencies across all the transactions in the application, without using any system-dependant information. The trace-derived approximate static application schedule and the heuristics to calculate IP processing times, prove efficient enough, since they maintain the transaction ordering, relative inter-injection times and satisfy both the pure-IP and cross-IP processing times, although, they do allow a few additional transactions that may not hold any semantic meaning to slip through. It must be noted, that porting and employing derived application schedules, result in very different looking transaction patterns, when compared to the reference traces.

4.3.3 Employing Application's Dynamic Schedule

Modeling traffic re-generation, by employing the application's schedule along with the IP processing times and transaction description, will result in a very accurate representation (model) of the application. Towards this, in the previous sub-section, an approach for deriving the application's approximate static schedule from the reference trace was suggested. Further, it was analyzed that the suggested heuristics, though effective, would still not handle synchronization events with 100% accuracy, since the number of transactions vary based on the interconnect being used and hence, it was suggested to obtain the application's original dynamic schedule from the user.

It can be claimed, that a dynamic schedule would give a better and more accurate representation of the application than a derived approximation of the static schedule. Under the assumption, that the application's dynamic schedule is available (not being a restricted Intellectual Property), some of the inherent advantages include the ability to correlate the application's behavior to specific sets of transactions and the ability to efficiently and accurately reproduce dynamic traffic patterns such as synchronization.

An illustration of a sample synchronization process, where the transaction count for the polling events is time (duration) dependent on another set of transactions, is presented below. Also, the significance of employing an application's dynamic schedule versus static schedule to handle such an event, is highlighted.

In a synchronization event, let's consider two master devices attempting to gain access to a shared slave and in the process, polling a semaphore variable. If master 1 arrives first and locks the resource; the attempt by master 2 will fail. Hence, master 2 will regularly issue read transactions to poll the semaphore till it is granted. Only after unlocking of the semaphore variable by master 1, will master 2 be granted the semaphore and additional polling events will not be required. The unlock event issued by master 1 and the last poll event by master 2 are dependent. Such synchronization events (depicted in Figure 4.3) are dependent on the network properties and hence, variable amount of transactions might be observed based on the interconnect in use.

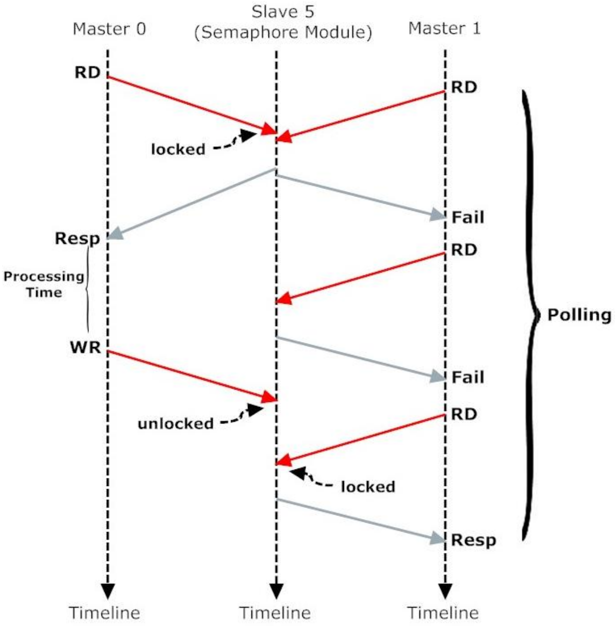


Figure 4.3: Synchronization Event

Another example of unpredictable events are the system-initiated interrupts towards a processor, will typically happen by means of writing to the interrupt device, can also be handled with the application's dynamic schedule. The concept of 'barriers' in multi-threaded architectures, where each thread waits upon completion until all of the other threads of a group reach the barrier point, is another example of such events. The ability to handle such synchronization events more accurately and to re-create unpredictable traffic patterns is, as established, one of the advantages of employing the application's dynamic schedule versus static schedule.

4.4 The Schedule Manager

As has been established in the previous section, employing a derived static schedule or a user-provided dynamic schedule presents an almost realistic depiction of an application's behavior. In both cases, an appropriate schedule manager is required, to make use of the schedule information and efficiently handle the transaction injections across masters.

The architecture and organization of the schedule manager depends on the type of schedule (static or dynamic) it is meant to handle. The key difference between the two types of schedule managers essentially lies in the representation of the schedule information available from the two solutions and the appropriate logic that is needed to handle the same.

To highlight this difference, let us consider scheduling a synchronization event, where the process of checking the status of the semaphore variable is characterized by a sequence of polling events.

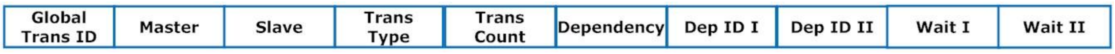
In case of a derived static schedule, the polling event is seen as a set of fixed number of transactions without any semantic value and although appropriate cross-IP dependencies are handled effectively. However, such an event is clearly defined in a dynamic schedule, where the number of the polling event transactions is considered dependent on the duration of another parallel sequence of transactions, that ends in unlocking the semaphore variable.

As can be seen in this example, the difference in implementation of the schedule manager, varies from handling merely a number and order of transactions, to handling a time (duration) dependent relation between two sets of transactions. The detailed implementation of the two schedule managers is described in the following sub-sections.

4.4.1 Static schedule manager

The static schedule manager, in order to handle derived static schedules, holds the injection order of all the transactions in the system, to be injected by any of the masters to its corresponding slaves. Based on the temporal ordering of the transaction sets/sequences made available by the static schedule, it maintains them as records in as many transaction tables as the number of masters, sorted by the injection order, according for every master.

Each of these records are stored in the transaction table with the description depicted in Figure 4.4:



Global Trans ID	Master	Slave	Trans Type	Trans Count	Dependency	Dep ID I	Dep ID II	Wait I	Wait II

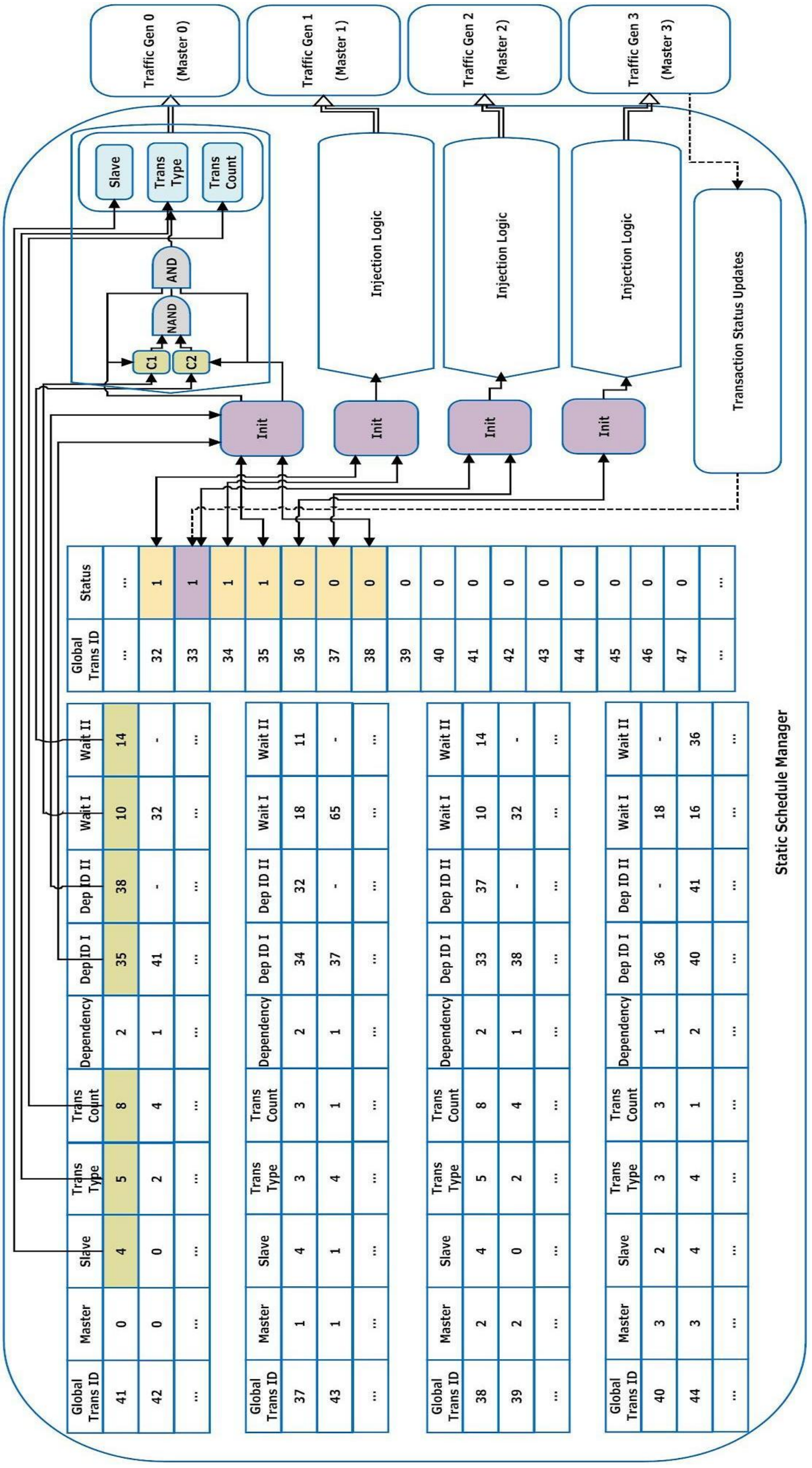
Figure 4.4: Static Record Description

In the figure 4.4, 'Global Trans ID' corresponds to the global sequence number allotted by the procedure used to derive the static schedule specified in section 4.3.2. 'Master' corresponds to the master that injects the transaction and 'Slave' corresponds to the destination of the transaction. 'Trans Type' corresponds to the type/characteristic of this transaction out of all the possible types of transactions that can be injected from the master and 'Trans Count' defines the number of transactions of this same type to be injected as a set/sequence. 'Dependency' corresponds to the number of transactions, the current transaction may be dependent on (1 or 2). Accordingly, appropriate Dependency ID(s) (I and/or II) are read, and these indicate(s) the transactions on which the current transaction is derived to be dependent on. Wait(s) (I and II) correspond to the pure-IP and cross-IP processing times, that this transaction is supposed to wait before being injected into the network upon reception of the appropriate responses.

Besides maintaining these records to handle application schedules, as can be seen in Figure 4.5, a transaction status table is also maintained in the schedule manager for the purpose of checking for responses, resolving dependencies and scheduling transactions for injection. If dependency exists from the previous transaction of the same master alone (Dependency = 1), when the response for the corresponding transaction is received (Status = 1), the schedule manager transfers control to the transaction injection logic, which starts a down counter with initial value from the 'Wait I' field (pure-IP processing time) and sends the transaction for injection when the counter reaches zero. It must be noted that the status of transactions is updated by a transaction status update handler, which acts based on the responses received from the traffic generators.

In case of unresolved dependencies, where the dependency may be due to the previous transaction of the same master or some other master and can be determined only at run-time, the schedule manager starts appropriate down counters upon reception of either of the responses, initialized with appropriate wait times. The transaction injection logic then checks for the start/init signals of both the counters to be 1 and both the counter values to be down to 0, before forwarding the next transaction information to the corresponding master for injection into the network.

This assures adherence to the max(pure-IP, cross-IP) wait function, derived in step(c) of the process defined in subsection 4.3.2, to resolve dependencies at run-time. The transaction information which is forwarded to the traffic generator, includes details about the target slave, the transaction type and the number of transactions in that sequence. This helps the traffic generator in injected an appropriate sequence of transactions to the specified slave. The architecture of the static schedule manager presented above, is depicted in Figure 4.5. The static schedule manager proposed here, effectively manages derived static schedules and injects transactions across the masters based on the schedule, thus maintaining the transaction ordering.



Global Trans ID	Master	Slave	Trans Type	Trans Count	Dependency	Dep ID I	Dep ID II	Wait I	Wait II
41	0	4	5	8	2	35	38	10	14
42	0	0	2	4	1	41	-	32	-
...	...	...	...	...	...	...	...	...	...

Global Trans ID	Master	Slave	Trans Type	Trans Count	Dependency	Dep ID I	Dep ID II	Wait I	Wait II
37	1	4	3	3	2	34	32	18	11
43	1	1	4	1	1	37	-	65	-
...	...	...	...	...	...	...	...	...	...

Global Trans ID	Master	Slave	Trans Type	Trans Count	Dependency	Dep ID I	Dep ID II	Wait I	Wait II
38	2	4	5	8	2	33	37	10	14
39	2	0	2	4	1	38	-	32	-
...	...	...	...	...	...	...	...	...	...

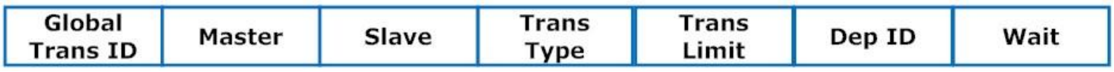
Global Trans ID	Master	Slave	Trans Type	Trans Count	Dependency	Dep ID I	Dep ID II	Wait I	Wait II
40	3	2	3	3	1	36	-	18	-
44	3	4	4	1	2	40	41	16	36
...	...	...	...	...	...	...	...	...	...

Global Trans ID	Status
...	...
32	1
33	1
34	1
35	1
36	0
37	0
38	0
39	0
40	0
41	0
42	0
43	0
44	0
45	0
46	0
47	0
...	...

Figure 4.5: Static Schedule Manager

4.4.2 Dynamic schedule manager

The dynamic schedule manager, has a simpler transaction description and injection logic when compared to the static schedule manager, due to the fact that in a user-provided dynamic schedule, the dependencies are clearly defined. The dynamic schedule manager also holds the injection order of all the transactions in the system, based on the temporal ordering of the transaction sets/sequences defined in the dynamic schedule. It also maintains the transactions as records in as many transaction tables as the number of masters, sorted by the injection order, according for every master. Each of these records are stored in the transaction table with the description depicted in Figure 4.6. Compared to the static transaction description, here, the dependency is clearly defined and appropriate wait time (pure-IP or cross-IP processing time) is provided. The 'Trans Limit' defines a limit on the number of transactions in that sequence, however, for synchronization events, this limit is undefined (set as 0). It is well established, that employing the application's dynamic schedule almost precisely represents its run-time behavior and efficiently reproduces dynamic traffic patterns such as during synchronization events. The transactions resulting from these events are very hard to predict in advance, since they are dependent on the interconnect's performance and hence, giving a limit to the transactions for such events makes little sense.

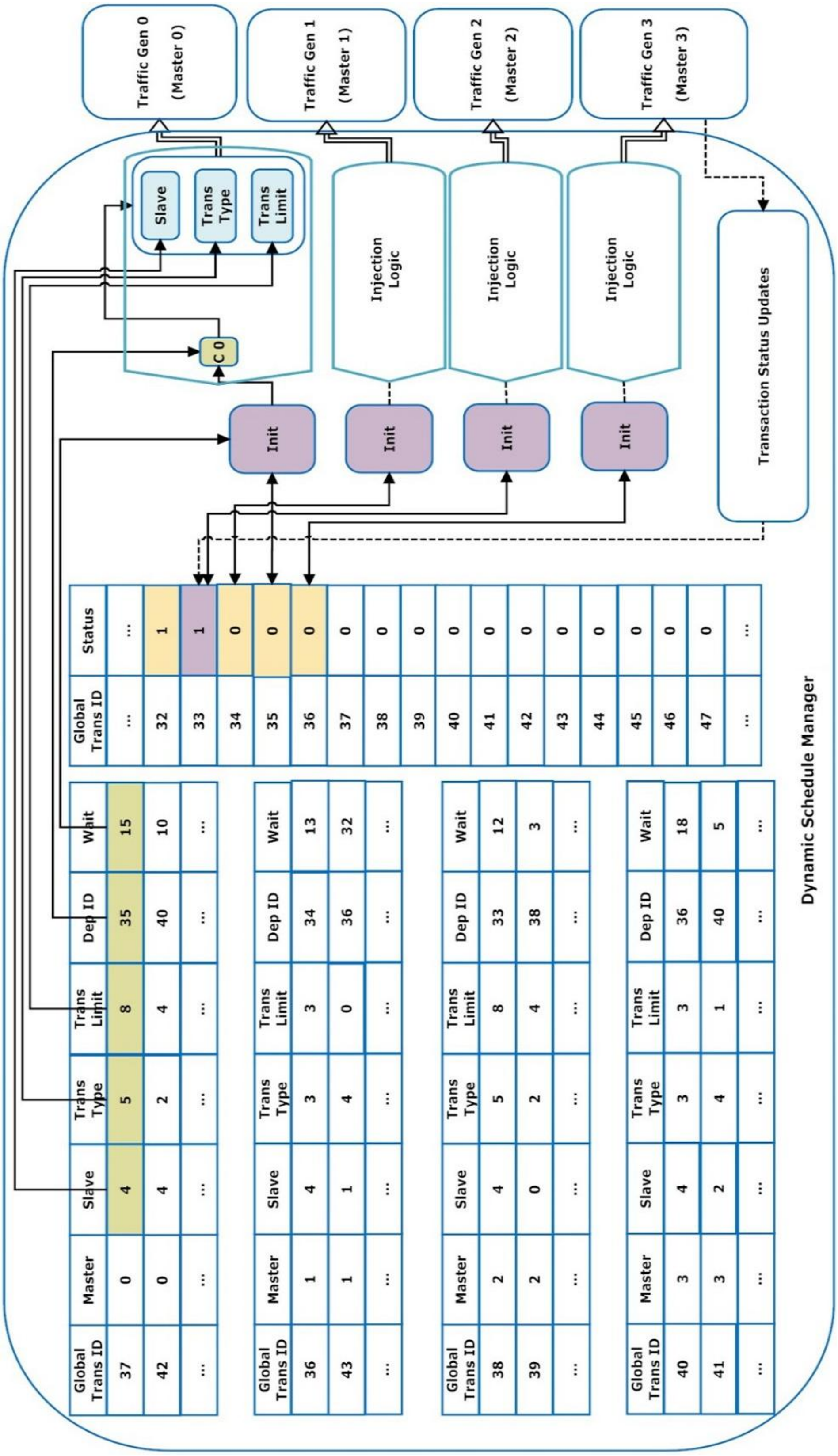


Global Trans ID	Master	Slave	Trans Type	Trans Limit	Dep ID	Wait

Figure 4.6: Dynamic Record Description

In the case of static schedule manager, subsequent transactions for a given master were scheduled for injection only after receiving the response/transaction complete signal for the previous transaction. This still holds true for the dynamic schedule manager, when the transaction limit is defined. However, as stated earlier, for synchronization events, the transaction limit is undefined and hence, such a response for the previous transaction injected by the same master is not awaited. This is due to the fact that the subsequent transactions are dependent on a sequence of transactions initiated by another master and not the same master.

As an illustration, looking back into the synchronization example in section 4.3.3, the 'unlocking' of the semaphore variable by master 1, grants master 2 the semaphore and brings an end to the 'polling' events of master 2 on the semaphore slave. This implies that the subsequent transactions of master 2 can be seen as being dependent on the response of the 'unlocking' event by master 1. Hence, to handle transactions when the transaction limit is undefined, the schedule manager enables the traffic generator to keep injecting transactions, till it receives a response for the defining event (transaction) at another master. In the example, this refers to receiving a response for 'unlocking' transaction from master 1. This causes the dynamic schedule manager to update the transaction status table entry for the 'polling' transaction to 1 and interrupt the current master to start its next set of transactions.



Global Trans ID	Master	Slave	Trans Type	Trans Limit	Dep ID	Wait
37	0	4	5	8	35	15
42	0	4	2	4	40	10
...	...	...	...	...	...	...

Global Trans ID	Master	Slave	Trans Type	Trans Limit	Dep ID	Wait
36	1	4	3	3	34	13
43	1	1	4	0	36	32
...	...	...	...	...	...	...

Global Trans ID	Master	Slave	Trans Type	Trans Limit	Dep ID	Wait
38	2	4	5	8	33	12
39	2	0	2	4	38	3
...	...	...	...	...	...	...

Global Trans ID	Master	Slave	Trans Type	Trans Limit	Dep ID	Wait
40	3	4	3	3	36	18
41	3	2	4	1	40	5
...	...	...	...	...	...	...

Global Trans ID	Status
...	...
32	1
33	1
34	0
35	0
36	0
37	0
38	0
39	0
40	0
41	0
42	0
43	0
44	0
45	0
46	0
47	0
...	...

Figure 4.7: Dynamic Schedule Manager

To initiate a new sequence of transactions, details about the slave, the transaction type and the limit of transactions are forwarded to the traffic generator. This also applies when interrupting the traffic generator to halt its current sequence of transactions. The schedule manager merely sends in new transaction information to replace the current as input to the traffic generator. This helps the traffic generator in injected an appropriate sequence of transactions to the specified slave. The architecture of the dynamic schedule manager, is depicted in Figure 4.7.

4.5 Challenges in Traffic Generation from Traces

In the proposed trace-based traffic generation, the inter-injection times are collected from the reference system and modeled to be ported to any platform or interconnect by, deriving an approximate static schedule of the application. The challenges in implementing the same are discussed below:

(a) **Handling Transactions**

The traffic composition must be representative of real IP core and hence, the traffic generator must maintain all possible transaction types which can be employed based on the schedule manager's request. The transaction limit indicated by the schedule manager, must be verified every time a transaction is injected into the network and when the given transaction sequence is completed, an appropriate response must be sent back to the schedule manager.

(b) **Portability**

The inter-injection times obtained from the reference trace must be scanned, to filter out the latencies associated with the base interconnect employed in the reference system and the IP processing times must be obtained. This would make the trace portable to any interconnect validation tool.

(c) **Handling Dependencies**

The application's schedule (static or dynamic) presents dependencies across transactions and this must be handled efficiently. The schedule manager is employed for this purpose, which requires responses from the traffic generator for proper scheduling of transactions. In the case of dynamic schedule, when the transaction limit is not mentioned or set to 0, the traffic generator must continue injecting the transaction till it receives an interrupt from the schedule manager. Also it must be noted that for such transactions no response is sent back to the schedule manager. It must be noted that unlike the probability distributions based traffic generator, the trace-based traffic generator, does not have to handle injection intervals on its own, instead it is handled by the schedule manager. Besides, since the schedule is pre-defined or derived, there is no need to employ re-scheduling algorithms or to monitor bandwidth (throughput) satisfaction levels.

4.6 Trace-based Traffic Generator Architecture

The trace-based traffic generator, employs application schedules (static or dynamic) with the aid of appropriate schedule managers, which indicate to the traffic generator, when and what type of transaction needs to be injected next into the network. The trace-based traffic generator is much simpler, when compared to the synthetic traffic generator, since all the schedule handling, injection-interval adherence and bandwidth monitoring issues are moved out of the traffic generator architecture to the schedule manager.

The trace-based traffic generator obtains traffic composition in terms of the bursts and transactions supported and the address space of the masters and slaves to setup the traffic injection module, which includes holds transactions types in separate transaction buffers for all the connected slaves. When the schedule manager requests the traffic generator for injection of a particular transaction type, in a sequence of transactions, this module selects the appropriate transaction for injection and forwards it to the transaction selector.

The transaction manager is in charge of monitoring the state of the network and triggers the transaction selector to forward the next transaction for injection, when the 'send next' signal is set by the network, indicating the possibility of the network accepting the next transaction. Every time a transaction is injected into the network, the transaction manager increments the transaction counter and checks for the transaction limit indicated by the schedule manager. When the given transaction sequence is completed, an appropriate response is sent back to the schedule manager.

If the transaction limit is not defined or set to 0, the transaction limit check is disabled and so is the response to the schedule manager. The traffic generator is interruptable, and hence, when a new set of transactions are scheduled to it by the schedule manager, the transaction manager resets the transaction counter and re-initializes the transaction limit check with the new limit, indicated by the schedule manager in the transaction information. The transaction selector forwards the transaction to the outstanding request queue, which is then injected into the network by the transaction generator. For simulation purposes, in order to avoid the network's influence on traffic injection, this outstanding request buffer is indefinitely long.

The FSM defined outside the traffic generator, connects a Network Interface (which may implement any standard protocol) and the protocol-independent traffic generator and is used to convert the traffic generator's output to the format defined by the protocol. It is designed to handle keep track of the status of the transactions, the response from the network for injecting the next transaction.

As stated before, this traffic generator architecture is also independent of any IP protocol and hence acts like a plug and play module that can be employed with any architecture. The design of the traffic generator is depicted in Figure 4.8.

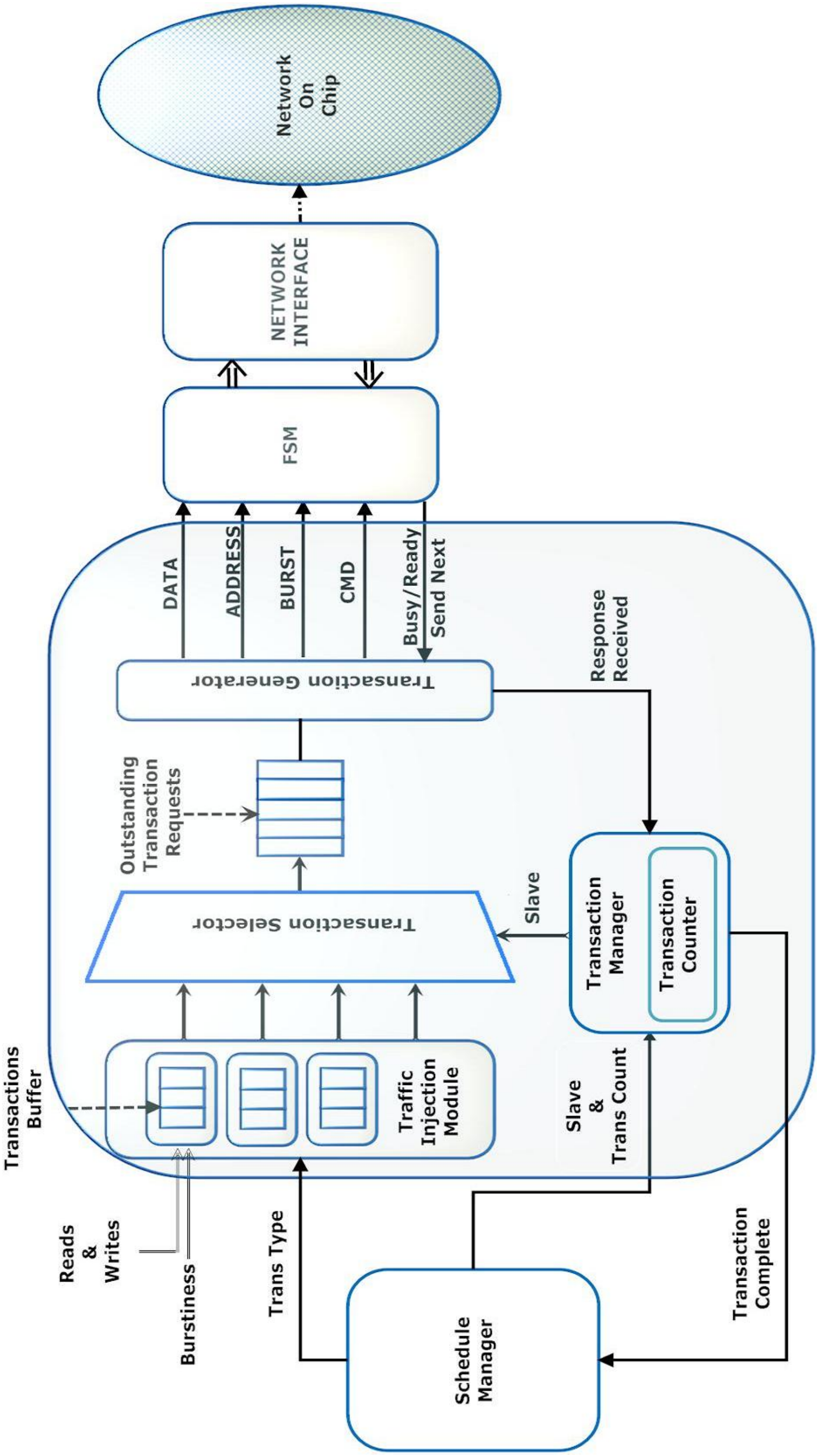


Figure 4.8: Trace-based Traffic Generator

5 Performance Validation and Simulation Results

5.1 Why Performance Validation?

With the ever increasing number of IP cores in MPSoCs, the communication interconnect must be able to keep up with the increasing performance demands due to the additional bandwidth requirements and yet stay within the area and power budget. Hence, while designing NoCs, it becomes absolutely essential to arrive at optimal architectural solutions. It must be noted that most optimizations depend on the target application and merely checking for adherence to objectives and constraints of the application does not suffice.

There still exists a need to verify the network design against real-time constraints imposed by the application, so as to arrive at a concrete and optimal NoC design for the given application. This necessitates the need to check for adherence to performance requirements against the application and guaranteeing low cost in terms power and area and hence, calls for performance validation and design optimization of NoC architectures. The objective of this thesis is to come up with such an infrastructure for validating the NoC design and implementation.

5.2 Challenges in Statistics Collection

The process of statistics collection is the most crucial aspect of the performance validation tool, since any bug or error or incompleteness or any form of incorrect reporting of statistics and latency information, will give rise to inaccurate conclusions. Reporting such erroneous results will not only fail to address the objective of design validation and optimization, but also mess up the design process (flow) and result in inappropriate network design and configuration, thereby affecting the working of the MPSoC. Hence, in order to design an efficient and accurate statistics collection framework, the following primary challenges must be addressed:

(a) **System Understanding** - It is absolutely essential to have a perfect understanding of the system, the behavior of its IPs, slaves and the network configuration (including switch design, buffers, flow control etc.) to get accurate measurements and statistics. It is also important to obtain appropriate statistics which can prove helpful in analyzing the system's performance. Care must be taken to note the different clock domains in the system to read the timings accurately. An easy way out is to use the appropriate clock periods from the masters and slaves and obtain the relevant times in nanoseconds.

(b) **Robustness** - As indicated earlier, any bug or error in reporting of statistics can lead to the failure of the validation tool, it becomes essential to maintain accurate numbers in spite of unwarranted events such as clock glitches. It must also be made sure that all values are recorded independent of parameters such as flit width and buffering.

(c) **Distributed Observation and Centralized Calculation** - More the statistics, more the accuracy, more the useful observations and more the validation tool complexity. For small NoCs, a centralized monitoring service is possible, however, it may become a bottleneck in large NoCs. In a distributed monitoring service, the observations are made in a distributed fashion at the different NoC components. While recording all of the events across the NoC at different locations can be extremely useful, there is also a need to report the numbers to a central data structure which continuously updates all the relevant timings and latency information between the transmission and reception of a packet.

(d) **Availability** - As indicated above, all the latency and buffer occupancy observations need to monitored and reported continuously. The latency and performance values must be constantly updated and numbers such as average, minimum and maximum latency, composition of the traffic in terms of number of reads/writes and burstiness etc., must be made available during the entire length of the simulation.

The information/statistics collected from the network to analyze its performance, gives us useful numbers which in effect, reflect the expected performance of the network in the presence of realistic load. Besides, it must be possible to have different loads in the network at different points in time to characterize the latencies versus the load and to observe the effects of varying buffer-depths on latencies, for a complete validation of the system.

5.3 Performance Metrics

Performance metrics that may be useful for network performance benchmarking/validation are bandwidth, throughput, latency, buffer usage, power dissipation, silicon area, etc.

To collect statistics and estimate metrics, the type and the timestamp of communication events are captured at the boundary of an IP core in a reference environment. Such an approach would ensure that the statistics are independent of the overheads in the master and slave cores and reflect the true NoC performance. This can also help model resource contention in a system which is dependent on the interconnect architecture, decoupling the effects of IP cores on latencies.

This opens up the possibility for communication infrastructure optimization and for the investigation of its impact on system performance under realistic workloads and different architecture configurations. To calculate these metrics, a set of statistics needs to be collected across the NoC at all times and simultaneously reported to a central metrics evaluation module.

An example of statistics collected at a Network Interface (NI) in the form of time-stamped events is shown below:

Master : 04
Slave : 01
CMD : RD
Data : 0xdeadbeef
Addr : 0x01001010
Burst : 4
Time : 180ns (Time of Injection/Reception at NI)

Statistics can also be collected individually for different sections of the application execution identifying for instance critical sections and synchronization patterns within the application.

Performance monitoring and validation requires effective metrics for distinguishing congested links from normal or underutilized connections. The two significant measures of performance viz., latency and buffer occupancy and their impact on NoC performance are discussed in the following sub-sections.

5.3.1 Latency Measures

One of the most significant metrics being measured is the network latency, which represents the delay between the initiation of a packet transmission by a sender and the receipt of that transmission by a receiver. When accuracy is the key, one-way latency for a link can be defined as the time from the start of packet transmission to the start of packet reception. This makes it independent of the link's throughput and the size of the packet, and is the absolute minimum delay possible with that link. The time from the start of packet reception to the end is measured as serialization delay.

In practice, this is further augmented by queuing and processing delays. Queuing delay occurs when a router receives multiple packets from different sources heading towards the same destination and needs to queue the packets for transmission. Processing delays are incurred while a router determines what to do with a newly received packet.

In a two-way communication, the network latency may be measured as the time from the transmission of a request for a message, to the time when the message is successfully received by the requester. While measuring the round trip latency, one generally excludes the amount of time that a destination spends processing the packet.

For simulation purposes, the slave is modeled with no packet processing and merely sending a response back when it receives a packet, to get a relatively accurate measure of the network latency.

Though measuring network latency gives us an idea of the network's performance, the end-to-end-latency measure evaluates the network taking elements beyond pure networking into account while evaluating the network's performance. End-to-end latency defines the difference between the time at which the first word of a message has been offered to the network and the time the last word has been delivered to the destination. Such a latency generally takes into account the IP processing (wait) times at all masters.

Analyzing end-to-end latency gives us a comprehensive picture of the performance of the network in a real working environment. Such an analysis would evaluate and validate the network's performance under the expected real-time spatial and temporal behavior of the system as the injected traffic would reflect the application schedule, transaction ordering and IP processing times.

In this validation tool, both these latencies are measured effectively with additional information from the proposed traffic generators, which are capable of handling application schedules, ordering and IP processing times.

Another significant performance metric is the throughput offered at the NI ports which is measured by the bandwidth satisfaction metric proposed in section 3.6. This measure of throughput is an indication of the average amount of data accepted by the network on that port in a certain amount of time.

5.3.2 Buffering

Buffers are employed in Switches and Network Interfaces to handle data and address flow-control policies.

Switches have buffers at their input and output ports. The size of these buffers impact the delay at the switch, as packets may have to wait for the output buffer to become available downstream. The simulation also collects information on latency measures and buffer utilization by varying the buffer sizes.

In general, the average latencies decreases as the buffer size increases. This is because as the buffer size increases, less packets have to wait for the output buffers to become available. The decrease is significant since higher loads show higher sensitivity in packet delay as the buffer size change. Although an increase in buffer size improves the performance, after certain point (as the load increases), further increasing the buffer size does not improve the response time of the network. This means, that from response time point of view, buffer delay is not a bottleneck. The percentage of delayed packets depends on the size of the output buffers. Hence, the output buffer of a switch should be larger than the input buffer. Larger buffer sizes at each port add to the cost of the switch. Hence, when designing a network, one should consider the appropriate buffer size and delay of the switch for the offered traffic load of every port of the switch.

In other words, the tradeoff between buffer size, switching delay, and number of ports must be considered when designing or reconfiguring a network in order to come up with the most cost-effective switch that fits the needs of the network.

Another set of buffers, used at the NIs, are useful in providing latency and throughput guarantees. Buffering in NIs is required to hide the end-to-end latency for the flow control mechanism and to provide full throughput operation. If the buffers are too small, then the throughput and latency are affected and no end-to-end guarantees can be given.

Buffers in the NIs absorb differences in speed and burstiness between the IP and the NoC and hide network internals, such as packetization. If the NI buffers are not sufficiently large, the guarantees are violated. The size must, however, be minimized, as the buffers are a major contributor to NoC power and silicon area. Approaches to dimension NI buffers in the design flow give an estimate of the expected buffer sizes, but is further verified and optimized in this work, since we dimension the buffers given the spatio-temporal behavior of the application.

5.4 Benchmarks Description

A couple of micro-benchmarks (asm-matrixdep and asm-matrixind) developed at the University of Bologna and a synthetic benchmark, are considered to verify the proposed performance validation tool. The first two benchmarks are executed on the MPARM platform to generate reference traces, which are then modeled using the heuristics suggested in Chapter 4 and appropriate set of transactions are re-generated. The synthetic benchmark, is generated employing the synthetic traffic generator proposed in Chapter 3.

These benchmarks vary in terms of the communication behavior across the cores and are useful in reflecting different communication events such as synchronization, parallel traffic injection to shared slaves etc., and complex network behavior due to congestions, thus, aiding in a comprehensive analysis of the network's performance.

The first two benchmarks are used to highlight synchronization events (such as polling, due to dependency between transactions) and parallel injections to private memory (independent of other processors). Complementing these two benchmarks, a synthetic benchmark generated using a combination of standard probability distributions is employed to inject transactions in parallel to shared slaves (assuming no inter-processor dependency), to analyze the impact of arbitration policies and buffering.

Since these simulations are used to analyze the impact of the benchmarks on the performance of the network, the system does not use OS synchronization techniques. Synchronization is instead handled by the application, using a pre-defined semaphore memory and semaphore checking is performed by polling. Also, all the benchmarks are designed for a topology with four masters/processors each having its private memory, besides, a shared memory, a semaphore memory and an interrupt module.

Below is the description of the benchmarks (source code is attached in appendix):

(a) asm-matrixdep : This benchmark implements a pipeline of matrix multiplications. Each processor executes a matrix multiplication between an input matrix from the shared memory and another matrix from its private memory, then feeds its output to the logically following processor (via the shared memory). Every core follows a fixed execution pattern: (i) copies an input matrix from shared space to private space; (ii) multiplies it with a matrix already in private space; (iii) copies the resulting matrix back to shared space. During all of the process, semaphore and/or interrupt slaves are queried to maintain synchronization.

Each matrix is of dimensions 8x8 and the simulation runs for 100 iterations, with each iteration involving execution of one 8x8 matrix multiplication at each of the processors (i.e. 400 pipelined 8x8 matrix multiplications in all). Initially, an input matrix is stored in the shared memory at a defined location with a fixed offset from the base address of the shared memory. Subsequently, the output matrix from the first processor is stored with a constant offset from the input matrix, required to hold 64 integer elements of the matrix. This new address being at a constant offset, is accessed by the logically following processor to receive its input, and upon completion of its processing, the output is stored with the same constant offset, from its input location. This goes in cycles and each processor is required to access a fixed set of memory locations based on its logical ID. Polling is handled using a dedicated semaphore memory.

(b) asm-matrixind : This benchmark implements a set of matrix multiplications in parallel. Each processor executes a matrix multiplication between two input matrices, both stored in its private memory. It then stores its output back in its private memory with a standard offset. In this benchmark, since all processors act independently of each other, there is no use of a semaphore and/or interrupt slave. Again, as in case of the previous benchmark, each matrix is of dimensions 8x8 and the simulation runs for 100 iterations, with each iteration involving execution of one 8x8 matrix multiplication at each of the processors (i.e. 400 parallel 8x8 matrix multiplications in all). Both the input matrices are stored in the private memory at a defined location with pre-defined offsets from the base-address and the output matrix is also stored with a constant offset required to hold 64 integer elements of the matrix.

Both of these benchmarks are executed on the MPARM platform with ARM7 processors and the corresponding traces are obtained. These traces are further modeled and the application schedule is derived and used to re-generate the transactions. It must be noted that the 32-bit ARM7 processors can only generate single bursts and 4-word bursts, for reads and writes (posted) alike, which does not significantly impact the network's performance, when there is congestion. Also, these two micro-benchmarks, do not reflect the scenario, where multiple masters/processors inject traffic to the same slave at the same time (independent of each other). Hence, to address these two issues, a synthetic benchmark is employed, which injects bursty traffic from all masters to a shared slave with equal priority.

(c) synthetic : The synthetic benchmark employed here, generates traffic in parallel using four different probability distributions from the four masters, to a shared memory, with similar average bandwidth requirements and equal priorities at the routers.

Further, the traffic is bursty (including 8 and 16 word bursts) and assumes no dependency across processors. The synthetic traffic generator with its bandwidth monitoring and slave scheduling policies is employed. This benchmark generates sufficient bursty traffic that significantly impacts the network's performance due to congestion and contention and helps analyze the impact of buffer depths.

For the purpose of traffic generation, Poisson, Gaussian, Exponential and Pareto distributions are employed respectively at the four masters and average bandwidth requirement of 50 MBps is assumed and injected towards the shared slave from each of the masters. All masters initiate transactions at the same time and access the shared slave with equal priority and hence, their transactions are interspersed.

5.5 Topology Specification and Simulation Setup

For simulation purposes, the architecture and topology considered, comprises of: (i) four master cores (32-bit ARM7 [30] processors in MPARM and equivalent traffic generators in the proposed validation infrastructure), (ii) their private memories, (iii) a shared memory, (iv) a hardware interrupt module, (v) a hardware semaphore module and (vi) the 32-bit interconnection network (NoC). Routers are used to connect the masters to their private slaves, the shared slave and the synchronization slaves.

To evaluate the network, memories are kept as simple as possible and do not feature any kind of internal buffering: every access, even when part of a burst, requires the same number of cycles to get a response. The interrupt device is used as a medium for processors to send interrupt signals to each other. This hardware primitive is needed for inter-processor communication and is mapped in the global addressing space. To generate an interrupt, a write is issued to a proper address of the interrupt device. A semaphore device is employed for synchronization among the processors; it implements test-and-set (locking/unlocking) operations for this purpose. This topology and the corresponding simulation setup is depicted in Figure 5.1.

As can be seen in the figure, statistics are collected from across the network at the Network Interfaces and Buffers. Every event/transaction that is generated and initiated from a master and received at the slaves is recorded. The statistics from the masters is collected at the 'initiator stats' module, while those from the slaves are collected at the 'target stats' module. These along with the occupancy information from all the buffers in the network are forwarded to the centralized evaluation module, which analyzes the data and reports the different latencies in the network and the average buffer occupancies. These results help in validating the network's performance and suggest possible optimizations by trading-off performance for area and power.

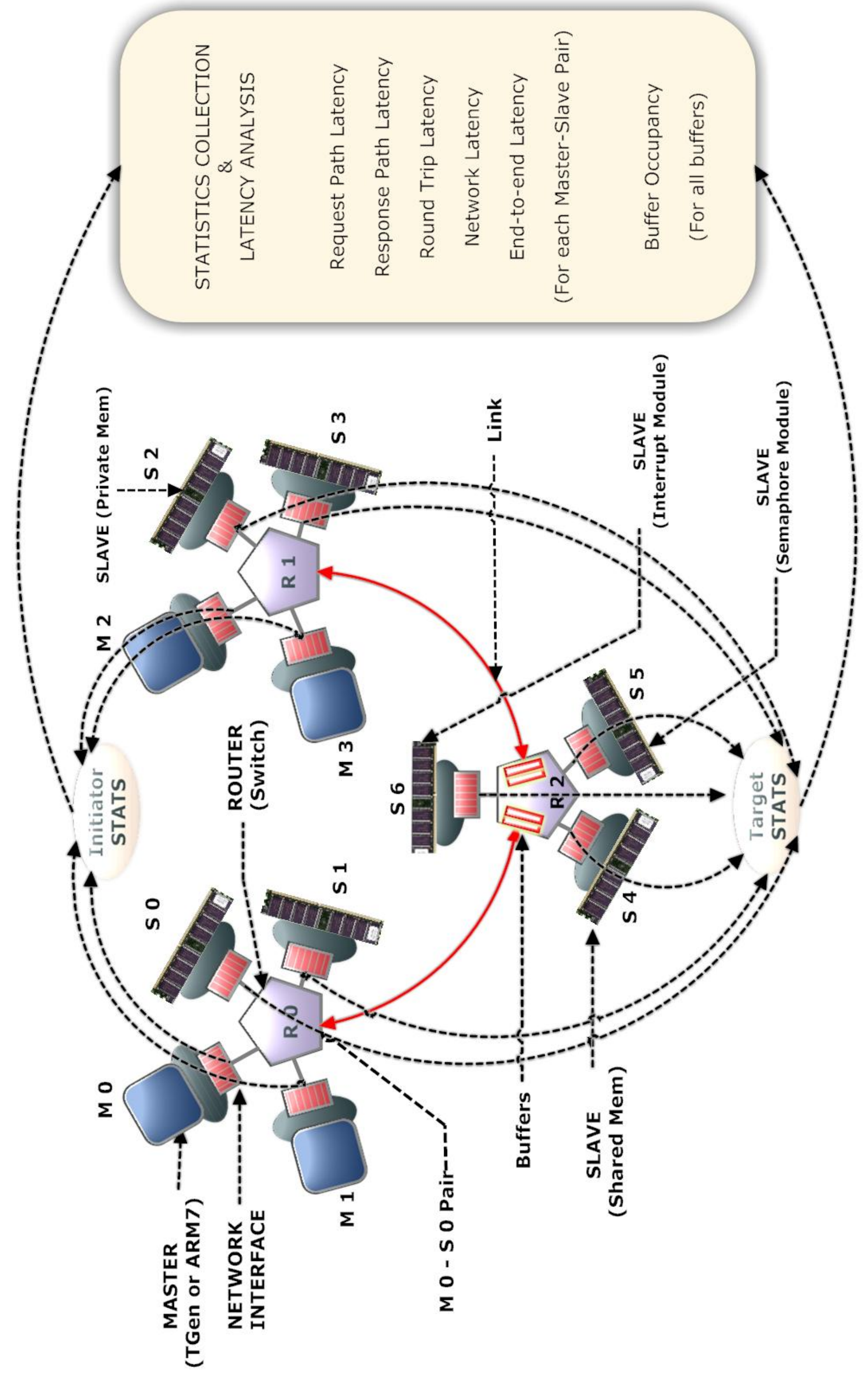


Figure 5.1: Simulation Setup

5.6 Simulations

The simulation results give us the expected performance and buffer occupancy values for the given network design under the traffic characteristics specified. The steps involved in the simulation process are as follows:

(a) Employ traffic generators, to inject the network with the specified application traffic (synthetic or application/benchmark trace). In case of application trace-based traffic generation, the traces need to be made available prior to this step, i.e. the benchmarks have to be executed on the MPARM platform first.

(b) Record all events (i.e. description for all transactions) and the relevant times, across the NoC using the independent statistics collection modules and simultaneously report them to the centralized metrics evaluation module.

(c) Calculate the metrics using a centralized data-structure to observe the latency measures and the buffer occupancies and with appropriate post-processing evaluate and report the metrics continuously.

The network latencies are obtained from the proposed validation tool, using simulations of the modeled benchmarks, on the topology described in the previous section. Such validation helps in effectively identifying link utilizations and performance numbers and suggesting design optimizations. The corresponding results and analysis are presented in detail in the following sub-sections.

5.6.1 Latency Observations

The performance numbers presented in this sub-section, give the average one-way and round-trip latency numbers on all the possible paths between all the masters and slaves for all the three benchmarks. Besides, these average latency metrics, additional post-processing yields minimum and maximum latency values, end-to-end latency values, network throughput etc., which can also be useful in validating the network. It must be noted that in tables 5.1, 5.2 and 5.3, SR corresponds to Single Reads, BR to Burst Reads, SWP to Single Writes (Posted) and BWP to Burst Writes (Posted). The table presents average one-way and round-trip latencies for the corresponding transaction types. The simulations were performed at a clock of 200 Mhz and the observed latency measures are reported in nanoseconds (ns).

Table 5.1 lists the observed latencies for the modeled asm-matrixdep benchmark, while Table 5.2 lists the same for the modeled asm-matrixind benchmark and Table 5.3 lists the same for the synthetic benchmark. In the first benchmark, all masters communicate with their private slaves, the shared slave and the semaphore slave. In the second benchmark all masters communicate only to their private slaves and in the third benchmark, all masters communicate only to the shared slave.

Transaction Type	**SR**	**BR**	**SWP**	**BWP**
Packets - Master 0 To Slave 0 (Private)	0	81	65001	0
1-way Request Latency Avg (ns)	0.00	37.28	35.02	0.00
R-Trip Network Latency Avg (ns)	0.00	137.28	0.00	0.00
Packets - Master 0 To Slave 4 (Shared)	6400	0	6400	0
1-way Request Latency Avg (ns)	36.34	0.00	36.58	0.00
R-Trip Network Latency Avg (ns)	76.58	0.00	0.00	0.00
Packets - Master 0 To Slave 5 (Semaphore)	1547	0	100	0
1-way Request Latency Avg (ns)	35.96	0.00	35.55	0.00
R-Trip Network Latency Avg (ns)	75.96	0.00	0.00	0.00
Packets - Master 1 To Slave 1 (Private)	0	77	63815	0
1-way Request Latency Avg (ns)	0.00	37.21	35.03	0.00
R-Trip Network Latency Avg (ns)	0.00	137.21	0.00	0.00
Packets - Master 1 To Slave 4 (Shared)	6336	0	6272	0
1-way Request Latency Avg (ns)	35.14	0.00	37.03	0.00
R-Trip Network Latency Avg (ns)	75.14	0.00	0.00	0.00
Packets - Master 1 To Slave 5 (Semaphore)	2350	0	197	0
1-way Request Latency Avg (ns)	35.17	0.00	35.48	0.00
R-Trip Network Latency Avg (ns)	75.17	0.00	0.00	0.00
Packets - Master 2 To Slave 2 (Private)	0	77	62660	0
1-way Request Latency Avg (ns)	0.00	37.21	35.03	0.00
R-Trip Network Latency Avg (ns)	0.00	137.21	0.00	0.00
Packets - Master 2 To Slave 4 (Shared)	6208	0	6144	0
1-way Request Latency Avg (ns)	35.31	0.00	36.28	0.00
R-Trip Network Latency Avg (ns)	75.31	0.00	0.00	0.00
Packets - Master 2 To Slave 5 (Semaphore)	1991	0	193	0
1-way Request Latency Avg (ns)	35.59	0.00	36.48	0.00
R-Trip Network Latency Avg (ns)	75.59	0.00	0.00	0.00
Packets - Master 3 To Slave 3 (Private)	0	75	61494	0
1-way Request Latency Avg (ns)	0.00	37.20	35.02	0.00
R-Trip Network Latency Avg (ns)	0.00	137.20	0.00	0.00
Packets - Master 3 To Slave 4 (Shared)	6080	0	6016	0
1-way Request Latency Avg (ns)	35.52	0.00	37.55	0.00
R-Trip Network Latency Avg (ns)	75.52	0.00	0.00	0.00
Packets - Master 3 To Slave 5 (Semaphore)	1689	0	95	0
1-way Request Latency Avg (ns)	35.08	0.00	35.00	0.00
R-Trip Network Latency Avg (ns)	75.08	0.00	0.00	0.00

Table 5.1: Latency measures for asm-matrixdep

As can be seen in the table above, in all iterations, all masters ‘Read’ an input matrix from the share slave and multiply it with the one ‘Read’ from the private slave. After the execution is completed, they ‘Write’ the output to the shared slave.

Transaction Type	**SR**	**BR**	**SWP**	**BWP**
Packets - Master 0 To Slave 0 (Private)	0	61	58601	0
1-way Request Latency Avg (ns)	0.00	39.84	35.07	0.00
R-Trip Network Latency Avg (ns)	0.00	139.84	0.00	0.00
Packets - Master 1 To Slave 1 (Private)	0	61	58601	0
1-way Request Latency Avg (ns)	0.00	39.84	35.07	0.00
R-Trip Network Latency Avg (ns)	0.00	139.84	0.00	0.00
Packets - Master 2 To Slave 2 (Private)	0	61	58601	0
1-way Request Latency Avg (ns)	0.00	39.84	35.07	0.00
R-Trip Network Latency Avg (ns)	0.00	139.84	0.00	0.00
Packets - Master 3 To Slave 3 (Private)	0	61	58601	0
1-way Request Latency Avg (ns)	0.00	39.84	35.07	0.00
R-Trip Network Latency Avg (ns)	0.00	139.84	0.00	0.00

Table 5.2: Latency measures for asm-matrixind

As can be seen in the table above, in all iterations, all masters ‘Read’ both input matrices from the private slave and multiply them ‘Write’ the output back to the private slave. In both the above benchmarks, all the other transactions correspond to those in the initialization phase and those used for storing the temporary computations.

Transaction Type	**SR**	**BR**	**SWP**	**BWP**
Packets - Master 0 To Slave 4 (Shared)	0	8155	0	11863
1-way Request Latency Avg (ns)	0.00	75.21	0.00	73.56
R-Trip Network Latency Avg (ns)	0.00	235.21	0.00	0.00
Packets - Master 0 To Slave 4 (Shared)	0	8153	0	11866
1-way Request Latency Avg (ns)	0.00	78.16	0.00	78.43
R-Trip Network Latency Avg (ns)	0.00	238.16	0.00	0.00
Packets - Master 0 To Slave 4 (Shared)	0	8146	0	11794
1-way Request Latency Avg (ns)	0.00	82.74	0.00	79.69
R-Trip Network Latency Avg (ns)	0.00	242.74	0.00	0.00
Packets - Master 0 To Slave 4 (Shared)	0	8149	0	11852
1-way Request Latency Avg (ns)	0.00	79.64	0.00	78.13
R-Trip Network Latency Avg (ns)	0.00	239.64	0.00	0.00

Table 5.3: Latency measures for synthetic benchmark

As can be seen in the table above, all masters simultaneously inject several bursty transactions only to the shared slave. These can be single or bursty ‘Reads’ or ’Writes’. A detailed analysis of the impact of varying buffer depths on the network’s performance, when employing this synthetic benchmark, is presented in the next sub-section.

5.6.2 Performance Validation and Optimization

A detailed analysis of the performance measures presented in the previous sub-section proves helpful, not only in validating the network against the initial latency and throughput requirements, but also in identifying modifications and possible optimizations to the network design.

For instance, if a particular link is found to be heavily loaded, a parallel link may be suggested. By increasing the buffer depths in the routers, if latencies can be brought down (within power and area budgets), suitable changes to buffer depths may be suggested. Alternatively, if excess buffer depths are observed on light-weight links, buffer depth optimizations may be suggested.

By employing the synthetic benchmark presented in Section 5.4, where all masters simultaneously inject several bursty transactions to the shared slave, it is possible to analyze the impact of varying the buffer depths on links with heavy loads and thereby, suggest optimizations.

Table 5.4 presents the observed average buffer occupancy numbers of the input buffers at the ports 0 and 2 in Router 2, in the topology described in Section 5.5, by changing the buffer depths (2, 6 and 10), when implementing the suggested synthetic benchmark. It also presents the power and area values for the different buffers depths (obtained from appropriate synthesis tools).

Buffer depth	2	6	10
Area (sq. um)	7934	24891	40988
Power (mw)	3.4	3.9	6.0
Avg. Buffer Occupancy (Port 0)	1.92	5.42	6.89
Avg. Buffer Occupancy (Port 2)	1.96	5.73	7.04

Table 5.4: Buffer Occupancy and Buffer Area and Power

Table 5.5 presents the performance values (average round-trip latencies) from all masters to the shared slave (Slave 4), when employing different buffer depths (2, 6 and 10) using the synthetic benchmark. These buffers are heavily occupied due to the continuous injection of traffic through them by the four masters. Hence, changing buffer depths improves the performance of the paths between the masters and the shared slave.

Buffer depth/Performance	2	6	10
Avg. RT Latency (ns) - Master 0 to Slave 4	235.21	201.12	192.91
Avg. RT Latency (ns) - Master 1 to Slave 4	238.16	201.83	193.64
Avg. RT Latency (ns) - Master 2 to Slave 4	242.74	203.37	197.35
Avg. RT Latency (ns) - Master 3 to Slave 4	239.64	203.20	196.92

Table 5.5: Impact of Buffer Depth on Performance

Power and area values for the different buffers are obtained from appropriate synthesis tools and are employed to find optimal trade-off points, taking into consideration, the performance benefits and the corresponding increase in area and power.

The graph depicted in Figure 5.2, shows the performance gains (based on round-trip latency measures between Master 0 and Slave 4) and the corresponding increase in power and area (all normalized to those of the 2-Deep buffers), when employing 6-deep and 10-deep buffers, as against 2-deep buffers, using values from table 5.5.

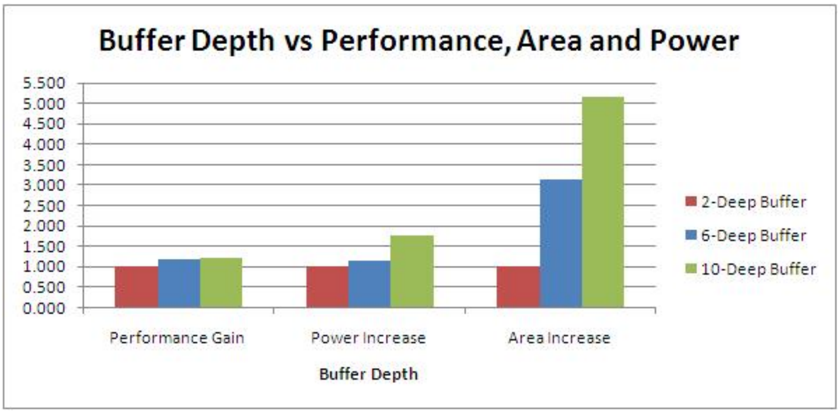


Figure 5.2: Performance Gains, Area Increase and Power Increase

The graph in Figure 5.3, shows the proportional decrease in round-trip latency (between Master 0 and Slave 4), when increasing buffer depths (normalized to that of 6-deep buffers), along with relative power and area increase.

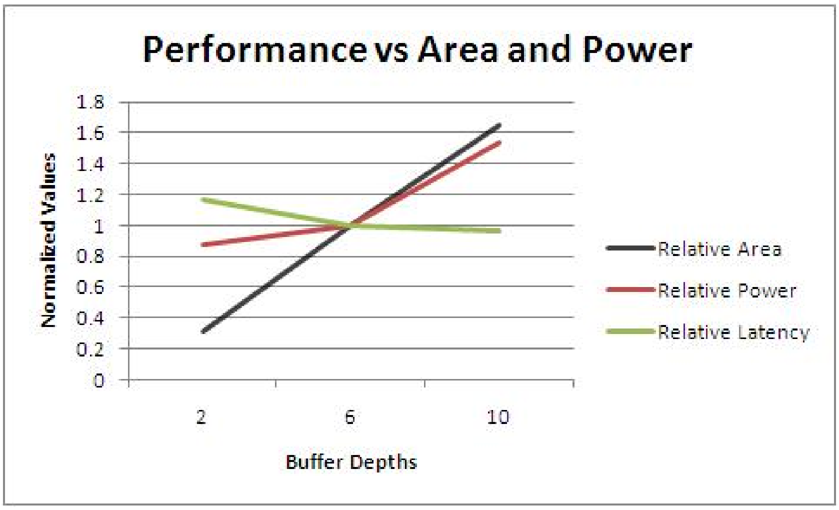


Figure 5.3: Performance vs Area and Power

As can be observed from the Table 5.5, there is a noticeable impact of using different buffer depths, on the round-trip latencies between the different masters and the shared slave (Slave 4). It must be noted in the graph in Figure 5.2, that though there is a difference in performance when using a 6-deep buffer instead of a 2-deep buffer, there does not seem to be a significant performance gain (or reduction in latency), when using a 10-deep buffer instead of a 6-deep buffer.

Now shifting focus on to the graph in Figure 5.3 and the Table 5.4, the power consumed by a 10-deep buffer is significantly more than that by a 6-deep buffer or a 2-deep buffer. Interestingly, a 6-deep buffer does not seem to consume a lot more power than a 2-deep buffer. When comparing the area occupied by these three buffers, there is an almost proportional increase in the area when we move from a 2-deep buffer to a 6-deep buffer and through to a 10-deep buffer.

Reflecting on these observations, one may suggest that selecting a 6-deep buffer, would serve the purpose by providing good performance gains, within acceptable area and power costs. Analyzing such trade-offs in performance versus buffer area and power, is useful in suggesting optimizations, for instance, increasing or decreasing buffer depths, etc., for the given application, based on the real-time congestions and contentions imposed.

In a nutshell, the infrastructure suggested in this thesis, proves very useful in validating the NoC design and optimizing its architecture for a given application, under real operating conditions.

6 Conclusion and Future Work

This thesis addresses ‘Validation of NoC designs’, for performance and efficiency against real-time constraints, by designing an infrastructure for performance analysis, validation and design optimization of NoCs.

6.1 Conclusion

With the technology scaling into tens of nanometers, chips comprising of hundreds of heterogeneous IP cores with complex functionalities, are now realizable. With the data bandwidth demands within the chip, shooting up drastically, Networks-on-Chip have been identified to provide a scalable and efficient routing alternative, promising high communication performance within area and power bounds.

As the technology matures, NoC designers are evolving newer and more efficient design flows, to arrive at accurate architectural solutions and to optimize design trade-offs. One of the factors that limits these goals, is that, most design choices and optimizations depend on the target application and merely checking for adherence to objectives and constraints of the application do not suffice. There still exists a need to verify the network design against real-time constraints imposed by the application, so as to arrive at a concrete and optimal NoC design for the given application. This calls for a comprehensive validation infrastructure that evaluates and verifies the design choices and aids in optimizing the architectural parameters.

The architecture of the proposed infrastructure comprises of three distinct phases: (a) Traffic Modeling and Generation (b) Traffic Scheduling and Management (c) Statistics Collection, Performance Validation and Optimization. For traffic modeling and generation, on one hand, synthetic traffic generators are developed to employ probability distributions and random traffic patterns, while on the other, trace-based traffic generators model and port application traces from a reference system and emulate the expected application behavior. For traffic scheduling and management, synthetic traffic generators are supported by efficient online re-scheduling algorithms, while trace-based traffic generators are supported by distinct schedule handlers, that maintain application schedules. In the last phase, an efficient methodology for statistics collection and performance validation is presented, besides exploring trade-offs in performance under area and power constraints, for design optimization.

Although this tool, does not provide 100% accurate design solutions, it plays a significant role in validating and optimizing the NoC design, since it takes into account the the target application’s expected communication behavior.

6.2 Future Work

Though the proposed infrastructure gives a comprehensive performance analysis of the NoC design, it does leave scope for future extensions and improvements. For instance, tweaking the schedule manager to provide dedicated priority on certain links, can extend the analysis to understand the impact of architectural modifications to NoC such as Quality of Service (QoS). Further, in principle, the suggested trace modeling methodology can be employed to study the impact of the arbitration policy, topology etc., on the performance of the NoC and appropriate optimizations may be suggested.

In the tool, it is also possible to extend the applicability of the schedule manager to define dependencies/schedules for synthetic traffic generators. A possible improvement can be optimizing the implementation of this infrastructure, to move it from the design phase to provide on-line NoC monitoring to support online routing schemes. An extension to this, can be using the NoC itself to collect statistics and update the centralized analysis data-structure.

Bibliography

[1] *System-on-a-chip*, `http://en.wikipedia.org/wiki/System-on-a-chip`.

[2] *Wormhole Switching*, `http://en.wikipedia.org/wiki/Wormhole_switching`.

[3] *Best Effort Services*, `http://en.wikipedia.org/wiki/Best_effort_delivery`.

[4] *Exponential Distribution*, `http://en.wikipedia.org/wiki/Exponential_distribution`.

[5] *Normal Distribution*, `http://en.wikipedia.org/wiki/Normal_distribution`.

[6] *Poisson Distribution*, `http://en.wikipedia.org/wiki/Poisson_distribution`.

[7] *Pareto Distribution*, `http://en.wikipedia.org/wiki/Pareto_distribution`.

[8] *Cauchy Distribution*, `http://en.wikipedia.org/wiki/Cauchy_distribution`.

[9] *Weibull Distribution*, `http://en.wikipedia.org/wiki/Weibull_distribution`.

[10] *Weighted Fairness Scheduling*, `http://en.wikipedia.org/wiki/Weighted_fair_queuing`.

[11] *Maximum Throughput Scheduling*, `http://en.wikipedia.org/wiki/Maximum_throughput_scheduling`.

[12] Federico Angiolini, *Interconnection systems for highly integrated computation devices*, PhD Thesis at University of Bologna (2007).

[13] David Atienza, Federico Angiolini, Srinivasan Murali, Antonio Pullini, Luca Benini, and Giovanni De Micheli, *Network-on-chip design and synthesis outlook*, Integration, The VLSI Journal **41** (2008), no. 3, 340–359.

[14] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri, *Mparm: Exploring the multi-processor SoC design space with SystemC*, The Journal of VLSI Signal Processing **41** (2005), no. 2, 169–182.

[15] Luca Benini and Giovanni De Micheli, *Networks on chip: a new SoC paradigm*, IEEE Computer **35** (2002), no. 1, 70–78.

[16] Luca Benini and Giovanni De Micheli (eds.), *Networks on chips: Technology and tools*, Morgan Kaufmann Publishers, San Francisco, CA, USA, 2006.

[17] Davide Bertozzi and Luca Benini, *Xpipes: A network-on-chip architecture for gigascale systems-on-chip*, IEEE Circuits and Systems Magazine **4** (2004), no. 2, 18–31.

[18] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna Tamhankar, Stergios Stergiou, Luca Benini, and Giovanni De Micheli, *Noc synthesis flow for customized domain specific multiprocessor systems-on-chip*, IEEE Trans. Parallel Distrib. Syst. **16** (2005), no. 2, 113–129.

[19] Dongyan Chen, Sachin Garg, and Kishor S. Trivedi, *Supporting voip traffic in ieee 802.11 wlan with enhanced medium access control (mac) for quality of service*, Tech. report, Avaya Labs Research, 2002.

[20] Wen-Tsuen Chen, Jaw-Liang Lo, and Hung-Chang Hsiao, *Multiple traffic scheduling for enhanced general packet radio service*, Vehicular Technology Conference, 2001. VTC 2001 Fall. IEEE VTS 54th, vol. 2, 2001, pp. 817–819.

[21] On-Line Applications Research Corporation, *Real-Time Executive for Multiprocessor Systems*, `http://www.rtems.com`.

[22] Matteo Dall'Osso, Gianluca Biccari, Giovannini Luca, Davide Bertozzi, and Luca Benini, *Xpipes: a latency insensitive parameterized network-on-chip architecture for multiprocessor socs*, Proceedings of the 21st International Conference on Computer Design, 2003, pp. 536–539.

[23] William J. Dally and Brian Towles, *Route packets, not wires: on-chip inteconnection networks*, DAC '01: Proceedings of the 38th annual Design Automation Conference, ACM, 2001, pp. 684–689.

[24] Design and Reuse, *Comparison of NOCs and Busses*, `http://www.design-reuse.com/articles/10496/a-comparison-of-network-on-chip-and-busses.html`.

[25] Mark Garrett and Walter Willinger, *Analysis, modeling and generation of self-similar vbr video traffic*, ACM SIGCOMM Computer Communication Review, 1994, pp. 269–280.

[26] David Geer, *Industry trends: Chip makers turn to multicore processors*, Computer **38** (2005), no. 5, 11–13.

[27] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago Gonzalez Pestana, Andrei Radulescu, and Edwin Rijpkema, *A design flow for application-specific networks on chip with guaranteed performance to accelerate soc design and verification*, Design, Automation and Test in Europe, 2005, pp. 1182–1187.

[28] Kees Goossens, John Dielissen, and Andrei Radulescu, *Aethereal network on chip: Concepts, architectures, and implementations*, IEEE Design and Test of Computers **22** (2005), no. 5, 414–421.

[29] Antoine Jalabert, Srinivasan Murali, Luca Benini, and Giovanni De Micheli, *Xpipes compiler: A tool for instantiating application specific networks on chip*, Proceedings of Design, Automation and Test in Europe Conference and Exhibition, 2004, pp. 1–6.

[30] Advanced Risc Machines Ltd., *Arm7 data sheet*, Tech. report, Advanced Risc Machines Ltd., December 1994.

[31] Shankar Mahadevan, Federico Angiolini, Jens Spars, Luca Benini, and Jan Madsen, *A reactive and cycle-true ip emulator for mpsoc exploration*, IEEE Transactions

on Computer-Aided Design of Integrated Circuits and Systems **27** (2008), no. 1, 109–122.

[32] Robert Mullins, *On-Chip Network Resources*, `http://www.cl.cam.ac.uk/~rdm34/onChipNetBib/noc.html`.

[33] Srinivasan Murali, *Methodologies for reliable and efficient design of networks on chip*, PhD Thesis at Stanford University (2005).

[34] OCP, *Open core protocol specification, release 2.0.*, 2003, `http://www.ocpip.org`.

[35] Stergios Stergiou, Federico Angiolini, Davide Bertozzi, and Giovanni De Micheli, *xpipes lite: A synthesis oriented design flow for networks on chips*, Design, Automation and Test in Europe, 2005.

[36] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson, *Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level*, IEEE/ACM Transactions on Networking **5** (1997), 71–86.

Micro-Benchmarks - Source A

The presented micro-benchmarks are developed at the University of Bologna and are executed on the MPARM platform to generate reference traces.

A.1 asm-matrixind

```
#define LINES 8
#define ITERATIONS 100
#define MAT_SIZE (LINES*LINES*sizeof(int));
#define PRIVATE (PRIVATE_BASE + 0x00020000)
typedef int matrix[LINES][LINES];

void main()
{
NODE_NUMBER = get_proc_id();
NNODES = get_proc_num();
int dummy, cycles, i, j, k, row, column;
char *dummy_mat_private = (char *)PRIVATE;
char *dummy_mat_input = (char *)PRIVATE + MAT_SIZE;
char *dummy_mat_output = (char *)PRIVATE + 2 * MAT_SIZE;
matrix *mat_private = (matrix *)dummy_mat_private;
matrix *mat_input = (matrix *)dummy_mat_input;
matrix *mat_output = (matrix *)dummy_mat_output;
end_boot();

for (row = 0; row ¡ LINES; row ++)
for (column = 0; column ¡ LINES; column ++)
{(*mat_private)[row][column] = row + column;
(*mat_input)[row][column] = 1; }

for (cycles = 0; cycles ¡ ITERATIONS; cycles ++)
{for (i = 0; i ¡ LINES; i ++)
for (k = 0; k ¡ LINES; k ++)
{(*mat_output)[i][k] = 0;
for (j = 0; j ¡ LINES; j ++)
(*mat_output)[i][k] += (*mat_input)[i][j] * (*mat_private)[j][k]; }}
stop_simulation();
}}
```

A.2 asm-matrixdep

```
#define LINES 8
#define ITERATIONS 100
#define MAT_SIZE (LINES * LINES * sizeof(int));
#define PRIVATE (PRIVATE_BASE + 0x00020000)
typedef int matrix[LINES][LINES];

void main()
{
NODE_NUMBER = get_proc_id();
NNODES = get_proc_num();
int cycles, i, j, k, row, column;
char *dummy_mat_private = (char *)PRIVATE;
char *dummy_mat_input = (char *)PRIVATE + MAT_SIZE;
char *dummy_mat_output = (char *)PRIVATE + 2 * MAT_SIZE;
char *dummy_shared_input = (char *)SHARED_BASE + (NODE_NUMBER - 1) *
MAT_SIZE;
char *dummy_shared_output = (char *)SHARED_BASE + NODE_NUMBER *
MAT_SIZE;
matrix *mat_private = (matrix *)dummy_mat_private;
matrix *mat_input = (matrix *)dummy_mat_input;
matrix *mat_output = (matrix *)dummy_mat_output;
matrix *shared_input = (matrix *)dummy_shared_input;
matrix *shared_output = (matrix *)dummy_shared_output;
end_boot();

for (i = 0; i ¡ NNODES; i ++)
semaphores[i] = 1;

for (i = NNODES; i ¡ 2 * NNODES; i ++)
semaphores[i] = 0;

for (row = 0; row ¡ LINES; row ++)
for (column = 0; column ¡ LINES; column ++)
(*mat_private)[row][column] = row + column;

if (NODE_NUMBER == 1)
{
for (row = 0; row ¡ LINES; row ++)
for (column = 0; column ¡ LINES; column ++)
(*shared_input)[row][column] = 1;
}
for (cycles = 0; cycles ¡ ITERATIONS; cycles ++)
{
for (row = 0; row ¡ LINES; row ++)
for (column = 0; column ¡ LINES; column ++)
(*mat_input)[row][column] = (*shared_input)[row][column];

if (NODE_NUMBER != 1)
semaphores[NODE_NUMBER + NNODES - 2] = 0;

for (i = 0; i ¡ LINES; i ++)
for (k = 0; k ¡ LINES; k ++)
{
(*mat_output)[i][k] = 0;
for (j = 0; j ¡ LINES; j ++)
(*mat_output)[i][k] += (*mat_input)[i][j] * (*mat_private)[j][k]; }

for (row = 0; row ¡ LINES; row ++)
for (column = 0; column ¡ LINES; column ++)
(*shared_output)[row][column] = (*mat_output)[row][column];

if (NODE_NUMBER != (NNODES))
semaphores[NODE_NUMBER] = 0;
}
stop_simulation();
}
```