Simulation-based Modeling Frameworks for Networked Multi-processor System-on-Chip

Shankar Mahadevan

Kongens Lyngby 2006
IMM-PHD-2006-157

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Abstract

This thesis deals with modeling aspects of multi-processor system-on-chip (MpSoC) design affected by the on-chip interconnect, also called the Network-on-Chip (NoC), at various levels of abstraction. To begin with, we undertook a comprehensive survey of research and design practices of networked MpSoC. The survey presents the challenges of modeling and performance analysis of the hardware and the software components used in such devices. These challenges are further exasperated in a mixed abstraction workspace, which is typical of complex MpSoC design environment.

We provide two simulation-based frameworks: namely ARTS and RIPE, that allows to model hardware (computation time, power consumption, network latency, caching effect, etc.) and software (application partition and mapping, operating system scheduling, interrupt handling, etc.) aspects from system-level to cycle-true abstraction. Thereby, we can realistically model the application executing on the architecture. This includes *e.g.* accurate modeling of synchronization, cache refills, context switching effects, so on, which are critically dependent on the architecture and the performance of the NoC. The foundation of the ARTS model is abstract tasks, while the foundation of the RIPE model is cycle-count. For ARTS, using different case-studies with over one hundred tasks (five applications) from the mobile multimedia domain, we show the potential of the framework under real-time constraints. For RIPE, first using six applications we derive the requirements to model the application and the architecture properties independent of the NoC, and then use these applications to successfully validate the approach against a reference cycle-true system.

The presence of a standard socket at the intellectual property (IP) and the NoC interface in both the ARTS and the RIPE frameworks allows easy incorporation of IP cores from either frameworks, into a new instance of the design. This could pave the way for seamless design evaluation from system-level to cycle-true abstraction in future component-based MpSoC design practice.

Preface

This thesis was prepared at the institute of Informatics Mathematical Modelling, in partial fulfillment of the requirements for acquiring the Ph.D. degree in Computer Science and Engineering department at the Technical University of Denmark. The Ph.D. was supervised by Associate Professor Jens Sparsø and Professor Jan Madsen.

The thesis stems out of the "On-Chip Interconnect Networks" project started in September 2002. The original Ph.D. study plan proposed an evaluation of reconfigurable networks for multi-processor systems-on-chip (MPSoC) with focus on low-power solutions. During the course of the study, it was found that understanding the application and the architectural properties of the MPSoC was the first crucial step towards this goal. The investigation of these properties was found to be a challenge in its own right. In this thesis, the solutions pursued to meet these challenges are presented for perusal towards the Ph.D. degree requirements. The outcome of this thesis are the ARTS and the RIPE frameworks, which can now allow a realistic investigation of the goals stated in the original study plan.

The thesis consists of a collection of seven research papers written during the period 2003–2005, and published elsewhere.

Lyngby, March 2006

Shankar Mahadevan

Manuscript Collection

The following list of manuscripts contribute directly to the body of this thesis.

#1: Tobais Bjerregaard, and Shankar Mahadevan. "A Survey of Research and Practices of Network-on-Chip." *To appear in the Journal of ACM Computing Surveys.* ACM, 2006.

#2: Jan Madsen, Shankar Mahadevan, Kashif Virk and Mercury Gonzalez. "Network-on-Chip Modeling for System-Level Multiprocessor Simulation." *In Proceedings of the 24th Real-Time Systems Symposium (RTSS), Cancun Mexico.* IEEE, Dec. 2003: 265-274.

#3: Jan Madsen, Shankar Mahadevan, and Kashif Virk. "Network-Centric System-Level Model for Multiprocessor System-on-Chip Simulation." *Interconnect-Centric Design for Advanced SoC and NoC. Eds. Nurmi J., Tenhunen H., Isoaho J., and Jantsch A. Dordrecht, The Netherlands.* Kluwer Publications, 2004: 341-365.

#4: Shankar Mahadevan, Michael Storgaard, Jan Madsen, and Kashif Virk. "ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality" *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), Atlanta USA.* IEEE, Sept. 2005: 480-483.

#5: Shankar Mahadevan, Federico Angiolini, Michael Storgaard, Rasmus G. Olsen, Jens Sparsø and Jan Madsen. "A Network Traffic Generator Model for Fast Network-on-Chip Simulation." *In Proceedings of Design, Automation and Testing in Europe Conference (DATE), Munich Germany.* IEEE, Mar. 2005: 780-785.

#6: Federico Angiolini, Shankar Mahadevan, Jan Madsen, Luca Benini and Jens Sparsø. “Realistically Rendering SoC Traffic Patterns with Interrupt Awareness.” *IFIP Very Large Scale Integration Systems and their Designs Conference (VLSI-SoC), Perth Australia.* IEEE, Oct. 2005: 211-216.

#7: Shankar Mahadevan, Federico Angiolini, Jens Sparsø, Luca Benini and Jan Madsen. “A Reactive IP Emulator for Multi-Processor System-on-Chip Exploration.” *Submitted for Journal Publication.*

The following maniscripts where also published during the course of this PhD, but are not part of this thesis.

- Tobias Bjerregaard, Shankar Mahadevan, and Jens Sparsø. ”A Channel Library for Asynchronous Circuit Design Supporting Mixed-Mode Modeling.” *In Proceedings of the 14th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Isle of Santorini Greece.* Springer Publications, 2004: 301-310.
- Tobias Bjerregaard, Shankar Mahadevan, Rasmus G. Olsen, and Jens Sparsø. “An OCP Compliant Network Adapter for GALS-based SoC Design Using the MANGO Network-on-Chip.” *Proceedings of the International Symposium on System-on-Chip (ISSoC), Tempere Finland.* IEEE 2005: 171-174.

Acknowledgements

> It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness. . . .
> - Charles Dickens, *A Tale of Two Cities.* London 1859.

In the journey towards my Ph.D. degree, culminating in this thesis, many people have shared their wisdom and warned me about pitfalls. My fellow Ph.D. student and friend, Tobias Bjerregaard for many intense and fruitful discussion. Thanks Tobias for introducing me to the electronic music scene in Copenhagen. This thesis would not have been possible without expert guidance and navigation by my supervisors, Associate Professor Jens Sparsø and Professor Jan Madsen. I am grateful to them for allowing me to follows the path charted in this thesis. Thanks also goes to Kashif Virk for his patience in answering my many questions. In Bologna, I am very grateful for the academic stimulus and the camaraderie of Federico Angiolini and the rest of the gang. Thanks Federico for introducing me to the best-of-the-best pizza and pasta places in Bologna. Thank you Prof. Luca Benini for many discussions, but mostly for allowing me to come to Italy and escape the danish weather. Two Masters' students, Michael Storgaard and Rasmus Olsen, who partook in the implementation activities. Thanks Michael for introducing me to <deque> in C/C++. Maria Jensen for keeping track of my Ph.D. accounts and patience. Per Friis for twice rescuing my hard disk. For funding my research, I am grateful to Nokia Denmark, SoC-MobiNET, Thomas B. Thrige Foundation and ARTIST.

Last but not the least, my parents and brother for their love - despite seeing me only for a few weeks in the past three years!

Shankar Mahadevan
Lyngby, March 2006.

Contents

Abstract **i**

Preface **iii**

Manuscript Collection **v**

Acknowledgements **vii**

I Preamble 1

1 Introduction **3**

1.1 Gist of the Published Work 5

1.2 Discussion 7

1.3 Outline of the Thesis 12

2 Concluding Remarks **13**

2.1 Contribution of this thesis 13

2.2 Suggested Future Direction 14

2.3 Summary and Conclusion 16

II Body **17**

3 Overview of Networked MPSoC **19**

4 The ARTS Modeling Environment **75**

4.1 Network-Centric System-Level Model for Multiprocessor System-on-Chip Simulation 76

4.2 ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality 100

5 The RIPE Modeling Environment **105**

III Appendix **129**

6 Network-on-Chip Modeling for System-Level Multiprocessor Simulation **131**

7 A Network Traffic Generator Model for Fast Network-on-Chip Simulation **143**

8 Realistically Rendering SoC Traffic Patterns with Interrupt Awareness **151**

Part I

Preamble

Chapter 1

Introduction

Integrated circuit (IC) design is driven by the target application domain, the architectural choices and the performance trade-offs. Generally, the applications dictates the architecture and the performance requirements. The architecture is the composition of hardware and software, while performance is speed, power, mobility, etc. The flow from specification to a deployable IC is influenced by the availability and ease of integration of the hardware and the software components. Investigating the performance of the IC, deviced by integration of these components can be a challenge due to many factors. First, the components have to be designed with a level of accuracy to give confidence in the eventual result. Second, due to correlations between the behaviors of the components, it is difficult to postulate how the optimization performed during design of individual components percolates to the entire IC.

The detail to which extent the IC components are modeled and simulated has direct impact on the accuracy and the time for understanding its performance. The closer the design description is to the eventual IC, the higher is the confidence in its performance. For example, a post-layout simulation accounts for all variables, i.e. wire and gate delays, suggesting a high degree of accuracy of the design. However, a large investment in man-hours is required for modeling and simulation at this level of detail. Given the shrinking time-to-market constraints, this investment would not be possible for many of the complex designs of the future.

A typical approach to IC design starts by taking an existing design methodology and apply it to the application and architecture in question. As is observed in [13], while this approach may indeed work for traditional "well-behaved" applications and architectures, the attempt is more likely to fail for more complex applications and architectures that can be expected in the forthcoming years. This is because of the increase in transistor density and the growing gap in using them productively in a timely fashion. This has given rise to a new IC design paradigm namely: *networked multi-processor system-on-chip (MpSoC).* We explain this new terminology as follows:

networked: This refers to the interconnect fabric used to bind the architectural components. As has been motivated in [2, 6], the future of IC design will be limited not by computation, but by communication. Hence a multi-hop, concurrent and distributed interconnect model, the so called Network-on-Chip (NoC) has emerged as candidate solution. We comprehensively address the issues related to NoC in Chapter 3.

multi-processor: This refers to the class of components termed intellectual property (IP); such as the computation and the memory units, that comprise the architecture. It includes the hardware (ASIC, FPGA, ASIPs, general purpose processors (GPP)) and the software layers (operating system and application) stacked on top of the hardware (where applicable). Over the last two decades, it is not as much their design, but the way these components are modeled and used that has changed. The emphasis is on re-use; wherein, the interface of these component are now well defined sockets [18]. Further, traditionally they were generally available only as RTL entities, while now they are described in a range of abstractions from un-timed functional to transaction to cycle-true and including RTL. Thereby, expanding their availability for performance evaluation at different stages of the design.

system-on-chip: This refers to the deployment of entire systems on a single chip in a predictable and timely fashion. Generally, it can be viewed as concurrent activity on two axis: horizontal, where hardware component are assembled (processors, ASIC, etc connected via the interconnect) and vertical, where the software components are compiled (application software, device drivers and operating system (OS)).

The basic premisses of the networked MpSoC design paradigm is component-based design practice with emphasis on the separation of computation and communication concerns. This premisses, has created a gap between the existing design and modeling framework which emphasis top-down step-wise design refinement, and the required frameworks that can undertake a mixed abstraction design exploration. The goal of the new frameworks *must* be to provide modeling primitives that can realistically capture the application behaviour and the architectural properties including the assessment of the impact of interconnect performance. For example, in a networked MpSoC, context switching and cache refills will be critically affected by the network latency, and thus impact the processor's ability to execute the application.

In this thesis, we identify the MpSoC properties affected by the interconnect, and suggest ways to model them at various levels of abstraction. To assess the impact of different applications and architectural changes on the performance of an instance of a networked MpSoC design, we provide two simulation-based modeling environments: ARTS (at system-level), and RIPE (closer to cycle-true abstraction). As will be detailed in the body of the thesis, the foundation of the ARTS framework are abstract tasks, while the foundation of the RIPE framework is cycle-count. In both cases, the execution of the application is abstracted away into "time-slices", albeit at different granularity i.e. at functional-block level in ARTS and at instruction level in RIPE. Using experiments and by validation with other reference systems, we show the potential of our modeling environments to handle many classes of applications seen in real-life. These applications are from different domains, showing real-time constraints requirements, employing different synchronization schemes, and containing multiple threads susceptible to interrupts and OS-dependent context switching. The investigation of such a broad class of application could produce general guidelines and recommendations to address many issues in the design of MpSoC systems.

The thesis is organized as a collection of published or submitted manuscripts. In the reminder of this chapter, we attempt to identify a common theme through these manuscripts. To do this, we first provide the gist of the concepts and techniques detailed in the manuscripts. This is followed by a discussion on the scope of this body of work, where we also fill some gaps in the evolution of the work. Finally we present an outline of the thesis and some notes for the reader to keep in mind during the reading of the remainder of the thesis.

1.1 Gist of the Published Work

In this section, we present the gist of the published papers that is part of this thesis. In this process we also categorize the work. Broadly, the papers can be collected into three groups (seven papers) as follows:

I. **A Survey of Networked MpSoC**

#1: **A Survey of Research and Practices of Network-on-Chip** (Accepted Journal Publication)

This work highlights many of the challenges in designing and modeling networked MpSoC. Specifically for this thesis, the motivation and reference to a large amount of related work can be found in this paper. Overall, NoC can be application-specific or a generic interconnect which can accommodate several applications. Generally, one can avoid over-design of the NoC architecture by studying the traffic requirements for a given problem. The traffic types (latency critical, individual or burst transactions) generated by the system can vary greatly depending on the application characteristics and architectural choices. Primarily one can conclude that these traffic types are the property of the hardware and the software layers stacked on top of the IP core.

II. **The ARTS Modeling Environment**

#2: **Network-on-Chip Modeling for System-Level Multiprocessor Simulation** (Conference Publication)

#3: **Network-Centric System-Level Model for MpSoC Simulation** (Book Chapter)

#4: **ARTS: A System-Level Framework for Modeling MpSoC Components and Analysis of their Causality** (Conference Publication)

This work highlights the requirements to model the application and the architecture at the system-level while giving a central role to the effects of the NoC. Overall, the ARTS framework described here is designed to meet the need for early exploration and understanding of architectural choices and application mapping in MpSoC designs. It is unlike some of the previous work at system-level exploration, wherein the frameworks are limited to exploration of causality between few classes of processors, memory or interconnect. The ARTS framework is not developed with any specific problem in mind, but is modularized and extendable in terms of modeling the different hardware and software layers observed in MpSoC systems. Further, it allows mixed (in terms of abstraction) instantiation for complex problems. From this thesis perspective, the modeling of the NoC in a detailed system-level framework as ARTS, allows us to assess the impacts of OS dynamics, selection of the hardware components, and mapping of the software tasks, on the system performance early in the design phase. A case-study with applications (MP3 decoder, GSM encoder/decoder, MPEG encoder/decoder) from the real-time multimedia application domain consisting of 114 tasks on a 6-processor platform for a hand-held terminal shows the co-exploration capabilities of ARTS. The case study highlights the impact of changing the underlying processing element (between ASIC, FPGA and general purpose processor), communication fabric (bus, mesh and torus) and OS scheduling policy on the processor utilization, the communication contention and the memory usage.

III. **The RIPE Modeling Environment**

#5: **A Network Traffic Generator Model for Fast Network-on-Chip Simulation** (Conference Publication)

#6: **Realistically Rendering SoC Traffic Patterns with Interrupt Awareness** (Conference Publication)

#7: **A Reactive IP Emulator for Multiprocessor System-on-Chip Exploration** (Submitted for Journal Publication)

This work highlights the requirements to model the application and the architecture in an environment closer to cycle-true abstraction. The reactive IP emulator (RIPE) described here can model computation behavior independent of the NoC properties, yet be reactive to changes in NoC architecture. Thereby, it effectively decoupled the simulation of the IP cores from the NoC. Originally deviced to merely mimic processor's behavior for NoC exploration, the *reactiveness* properties identified for emulation has opened opportunities for alternate uses and are explored in a case study documented in the above papers. The hardware and software properties captured in this framework are derived from execution of complex real-life application templates showcasing semaphore-based synchronization, OS scheduling based on time-slicing (multi-tasking), pipeline multimedia data processing, and I/O operations. Further we have validated the approach with a reference cycle-true framework and have determined that great accuracy (over 99%) and notable speedup can be achieved with our RIPE framework.

As will be outlined later, this grouping of the papers not only serves the purpose of categorizing the work covered in this thesis, but also as chapters of this thesis.

1.2 Discussion

The categorization of the work presented above, may at first glance appear to have a seemingly diverse focus. Therefore, in this section we attempt to identify a common theme across the work.

Abstractions	Foundation	Framework	Papers
System-level View	*Tasks*	ARTS	Paper #2 Paper #3 Paper #4
Programmer's View with/without timing	*Memory Map*		
Cycle Accurate	*Clock Cycles*	RIPE	Paper #5 Paper #6 Paper #7

Table 1.1: Abstractions of the Networked MpSoC Addressed in the Thesis.

1.2.1 Modeling Scope

The MpSoC design-related problems can be explored either in the analytical or the simulation domain. The scope, i.e., the problem representation and analysis style, of the ARTS and the RIPE modeling framework, falls into the simulation domain. Analytical approaches to solving MpSoC problems also exists and are well documented in [22, 21, 12, 16, 24]. However, as is also observed in [28], the performance of complex systems such as NoC is not easily expressed analytically. The simulation-based approach on the other hand addresses only the average-case behaviour. We have developed the ARTS and the RIPE framework with the view that one can easily formulate the problem and compare the results across different platforms and implementations. The frameworks are not developed to address any specific design problem, but to provide a necessary set of primitives to model all the required hardware and software components to instantiate the given design problem and evaluate it effectively in different abstractions. In order to take advantage of analytical approaches such as guarantees on best-case and/or worst-case behaviour, we propose a hybrid simulation/analytical approach as is done in [14] and [3]. Here, a limited part of the system (shared resource constraints in [14] and performance analysis in [3]) is described formally within a larger simulation-based setup. Such a design exploration approach can also be accomplished in our frameworks.

1.2.2 Modeling Abstractions

The MpSoC design-related problems can also be analyzed at many abstraction levels, with varying detail of the MpSoC layers (i.e. application, operating system and hardware). Table 1.1, adapted from [5], shows a subdivisioning of various abstractions employed during the MpSoC design. These can be used system-wide, meaning any component be it the NoC or the IP cores can be described at any level of abstraction and then be integrated with other components

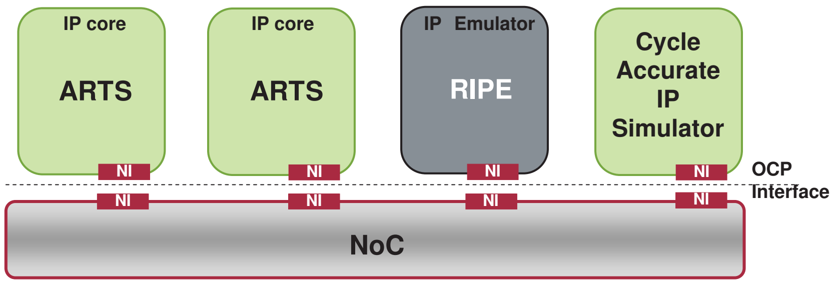


Figure 1.1: System-wide Abstraction for Modeling MpSoC Components.

via suitable interfaces for performance analysis. Such a system using components from ARTS, RIPE and cycle accurate (CA) framework is illustrated in Figure 1.1. Here, the components use standard sockets at the network interface (NI), which in the case of our frameworks is compatible with open core protocol (OCP) [17].

In the ***system-level view*** (SV), instead of the actual functionality, the execution time of the task representing the functionality is used to model the application's behavior. In this case, the interdependencies between the tasks translates into communication carried over the NoC. Taskgraphs are a well-known way to represent and structure such coarse-grain application behavior at this abstraction. To associate architecture properties into the application behavior, the task properties (execution time, memory requirement, power consumption, etc) are characterized on various IP cores. However, the impact of cache behavior, consequences of data dependencies, contention over shared resources, and so on, are difficult to predict at this abstraction, and hence, a degree of tolerance is introduced while assessing these properties. This observation leads to a spectrum of behaviors from *best-case* to *worst-case* scenarios.

Keeping this mind, a range of frameworks have been proposed in the literature [9, 1, 10, 13, 27]. They investigate the impact of OS scheduling, and limitations posed by the processor and the interconnect architectures such as memory and bandwidth, for a given application domain. Our ARTS model is inspired by the desire to undertake similar investigation. However, as is distinguished in the papers, we also attempt to modularize the framework to include a range of IP cores e.g. ASIC, GPP and FPGA, and a range of OS scheduling policies such as earliest-deadline-first (EDF) and rate-monotonic (RM), with support for preemption. Via the framework's comprehensive support for both hardware and software layers, i.e. application, OS and the platform architecture, the designers can investigate problems both in the general and the real-time application domains.

To do this investigation, the ARTS framework utilizes three basic blocks: the allocator, the scheduler and the synchronizer. The *allocator* controls the ownership of resources: be it execution engine of the processor, or the routers/links of the NoC. The *scheduler* controls the order in which the task execute on the resource: be it application task on the PE or communication tasks in the NoC, and the *synchronization* controls the interdependencies: be it precedence constraint in application tasks or priortization of communication tasks. The ARTS modeling primitive is based on the principle of composition outlined in [26]. As a way of preserving composition, the above described blocks handle its relevant data independently of the other. The communication between the application task and the RTOS blocks is handled by message exchanges. This way the MpSoC designer can easily combine alternate allocation, scheduling and synchronization policies without cumbersome recoding of the entire RTOS or compiling of the framework. This is the motivation for selecting composition based modeling. Additionally, we have found common characteristics to model both a diverse range of IP and interconnect behaviors using these three blocks.

The potential of the ARTS modeling framework has been demonstrated via case studies of a mobile multimedia terminal where the advantages of introducing NoC has to be traded-off against performance parameters such as memory, power and program completion time. In some cases even correct operation of the system cannot be guaranteed. For example, we show (in Paper #3) that even a small MpSoC system with three processors connected via a torus NoC (using wormhole routing protocol) could potentially cause system-deadlock due to OS preemption of the communicating tasks.

In the ***programmer's view*** (PV) of system design, parts of the architecture is exposed to the application, thereby introducing a degree of accuracy in the modeling and performance evaluation. As is discussed in [5], in the untimed PV the absolute behavior is not guaranteed, but the degree of accuracy can be postulated based on the description of the IP model such as pessimistic, optimistic, random, typical or a combination of models in these circumstances. Communication is point-to-point and based on a common, highly efficient transport mechanism. In the timed PV the request and response are completed in a single transaction and time is indicated as 'time-passed' rather that event-per-clock-tick. This view is analogous to a range of models also described under transaction-level models (TLM) [4, 7, 11, 19, 20, 23, 25].

By sacrificing simulation speed, the models at this level extract additional accuracy for performance evaluation. The goal of such analysis is the same as for SV i.e. investigating and extracting as much performance as possible out of given processor and interconnect for a given application. Parts of both the ARTS and RIPE frameworks straddle this level of abstraction. In the ARTS, the communication interdependencies are triggered by writing to specific address in the IP cores. In the RIPE, except for a few special purpose registers, the complete program, data and register files are addressable. Overall, in either frameworks, the presence of OCP [17] inherently allows to access the public memory of the IP cores.

The RIPE framework was originally devised to optimize the interconnect performance at the cycle-true abstraction. To do this it has to be *reactive* to the NoC architectural changes. For example, network latency could have different outcomes on the system performance in cases where synchronization occurs over the interconnect and OS-dependent context-switching is involved. The RIPE can be programmed to account for the impact of communication latency on the application execution. Via a simple non-pipelined instruction set architecture, implementing basic flow-control instructions, it can be configured to initiate a range of communication transactions (single read/write, burst read/write, interrupts) separated by idle waits. Thereby, it can mimic the externally observable behaviour of an IP core executing an application for the rest of the MpSoC. By introducing a programmable paradigm, the RIPE can be used in association with manually written programs to generate traffic patterns typical of IPs still in the design phase, helping in the tuning of the communication performance or understanding the causality relationship with other IPs in the MpSoC. This choice allows us to describe reactiveness characteristics of a wide range of IP cores at different levels of abstraction. Additionally, this choice allows future deployment as a hardware device in test chips containing interconnect prototypes. Through case studies based on real-life applications, such as multimedia data processing, input/output operation, and OS-aware multi-tasking, we have demonstrated that the RIPE can handle and emulate a wide class of application behaviours independent of interconnect aspects.

In the ***cycle accurate*** (CA) view of the system design, nearly all aspects of the architecture are described. The pipelined behavior, the address and data encoding/decoding and every other atomic (non-interruptible) action sequence can be tracked at every clock cycle at this abstraction. The work presented in [8, 15] models this abstraction. Such models provide a high degree of accuracy for investigating both the interconnect and the processor performance. This affords us the mechanism to validate the proposed frameworks (ARTS and RIPE). As is outlined in papers in Group III, this thesis covers the work done to validate the RIPE against the MPARM proposed in [8]. The validation of ARTS framework is left as future work.

From the above discussion, we can visualize a common theme, stretching from work related to ARTS to work related to RIPE. The commonalty between the two frameworks is that, their respective modeling primitives attempt to capture the interaction among the same three entities i.e. the application, the OS and the architecture. The difference is that they do so at different abstractions. As eluded to before, the presence of OCP at the interface of both the ARTS and RIPE allows easy mixing of modules from one framework with other (Figure 1.1). This would allow mixed abstraction design exploration. Though not addressed in this thesis, a comprehensive framework that can operate at any mode of abstraction is foreseeable. Instantiation of mixed-abstraction design is already possible using the components from the ARTS and the RIPE frameworks, which are the focus of this thesis.

1.3 Outline of the Thesis

The thesis is organized in three parts: Preamble, Body, and Appendix. The current chapter (Chapter 1) and the following chapter acts as a preamble for the rest of the thesis. As has been demonstrated in this chapter, the preamble part sets the scene and draw a common theme for the main body of the thesis which is a composition of various peer-reviewed published papers. Chapter 2 summarizes the contribution of the paper, and presents concluding remarks and hints at future direction.

The body of the thesis has three chapters. In Chapter 3, we present the paper (Paper #1) that provides an overview of issues relating to the NoC aspects and its impact on MpSoC design and performance. This is followed by two papers (Paper #3 and #4), which comprise Chapter 4 and detail the work related to the ARTS framework. In Chapter 5 via the Paper #7, we detail the work related to the RIPE framework.

Note that we have selectively combined the papers listed in Section 1.1. Papers #2, #5 and #6 are not part of the main body of the thesis but can be found in the Appendix part. The reason is as follows. Paper #2 is limited version of Paper #3, while Paper #5 and #6 are precursors to the Paper #7. Papers #2, #5 and #6 can be found in Appendix 6, 7 and 8 respectively. This is to ensure a consistent reading of the thesis, and to avoid revisiting similar concepts spread across different papers.

The various papers comprising the main body of the thesis have been published over different stages of the development of the frameworks. Consequently, a note on the nomenclature is suitable. With regards to the ARTS framework, in Paper #2 and #3, it is referred to as ‘abstract system-level model’ or ‘system-level RTOS modeling framework’. With regards to the RIPE framework, in Paper #5 and #6, it is referred to as simply ‘traffic generator’ or ‘reactive traffic generator’. The nomenclatures reflects the state of the framework at the time of publication.

Chapter 2

Concluding Remarks

2.1 Contribution of this thesis

Here, we outline the specific ideas, concepts and techniques that have been contributed by the author of this thesis. We refer to abstractions outlined in Table 1.1 (in Section 1.2.2) to structure the research work.

i. A structured overview of the networked MpSoC research has been presented. There are many challenges and opportunities identified in this overview, ranging from the design of individual NoC components, such as routers and links, to higher-level architectural concerns. An outline of modeling and design issues related to NoC in the wider MpSoC is also presented.

ii. At the system-level, the identification of modeling primitives to capture the causality between the hardware and the software components, when taking the behaviour of the NoC into account, has been the main contribution. The motivation here is to understand the cross-layer dependencies of the architecture, the OS, the device drivers and the application layers. The causality is understood by modeling and implementing the NoC topology and protocol aspects through the basic blocks of the ARTS model namely: the allocator, the synchronizer and the schedular. Requirements and implementation of modeling primitives capturing memory dynamics for abstract task execution and communication was also undertaken. We have successfully modeled bus, mesh and torus architectures and then performed a co-exploration to demonstrate the impact of these architectures on the system performance under real-time constraints. The trade-off metrics that were monitored include processor utilization, memory usage and communication contention.

iii. Near the cycle-true abstraction, the contribution of the thesis can be listed as follows.

- We have identified, the so called *reactive* behaviour essential to undertake exploration of alternate NoC architectures and features under realistic application behaviour. The idea is to abstract away the computation time while maintaining data and interrupt dependent communication sensitivity in the application behaviours. The reactive behaviours include complex synchronization schemes (as is observed in multimedia data processing) and OS interaction (as in multi-tasking and input/output operations).
- We have developed a simple instruction set architecture based model namely, the reactive IP emulator (RIPE), to mimic the IP core's reactiveness at its interface with the NoC. This model has three basic flow-control instructions (IF, JUMP and Set Register) which, we have found to be sufficient to model the wide class of reactive behaviour mentioned above. Additional instructions support the range of communication transactions, and parameterized computation time (via idle waits or cycle-count).
- We have successfully validated our RIPE approach with a cycle-true reference system via executing templates of applications possessing these reactiveness properties in a multithreaded environment.
- Finally, we have developed a case study to show the potential of such abstraction of computation time (into cycle-count) in a design space exploration for reducing communication latency and therefore execution time.

2.2 Suggested Future Direction

In Paper #7 we have validated RIPE framework against a cycle-true reference system. In the near term future, the validation of the ARTS framework against RIPE or a cycle-true framework is desirable. This step would allow for a seamless component-based design flow from abstract to cycle-true environment.

In the long term, the complexity of the MpSoC architecture and applications can only be expected to grow. Due to modularity, the challenge in designing individual components would diminish, however the challenge of integrating and understanding the impact a collection of these components into a MpSoC will grow. Overall frameworks that support mixed abstraction study in a predictable and scalable fashion is required.

Given the experience during this thesis work, considerable research potential in following two fields have been identified:

- Mechanisms and interfaces to complement the simulation-based frameworks with some analytical models would enhance the solution space covered during the MpSoC design space exploration.
- A flexible techniques to partition and apply parts of an application in abstract "task" form and other parts in different (possibly C/C++ code or cycle-count) form would be very useful during the study of a mixed abstraction design.

Realization of these goals is not easy by any means. As eluded to in Section 1.2.1, work presented in [14] and [3] is already addressing preliminary concerns in mixed simulation/analytical frameworks. For mixed abstraction instantiation, considerable understanding of the application behaviour and structure (e.g. functional blocks, OS access, etc) and underlying architecture (cache configuration, synchronization means, etc) is needed. The literature in Chapter 3 mentions many efforts to address this issue.

The practical uses of instantiating designs in any and mixed levels of abstraction are many. First, for design from start, it can take advantage of availability (in terms of the same entity described in multiple abstraction) and selection of IP cores for performance evaluation at different stages of the design abstraction. With insight and moderation, this will allow investigation of a greater number of design instances much earlier in the design phase. For simpler MpSoC design problems, one could even envision developing a automated computer-aided tool for taking the design problem from specification to candidate solution, in a fast and rigorous manner. Second, for design re-use, it can allows us to access the impact of replacement of select parts of design without excessive modeling and time spent on integration and debugging. However, until mechanisms to accomplish this type of easy mixing of abstraction with detailed description of both hardware and software components are available, the separation of the IP and the NoC related concerns, as is prescribed in our work can assist the networked MpSoC designer to optimize the individual components or the system as a whole.

2.3 Summary and Conclusion

The contribution of this thesis are two simulation-based frameworks, ARTS and RIPE, that cover a range of abstractions in modeling networked MpSoC. Crucially, via these frameworks we have attempted to fill the gaps between the existing design and modeling frameworks, and the required framework for realistically capturing hardware and software behaviours. Unlike typical MpSoC frameworks, which operate in one abstraction, these two frameworks can operate in a mixed abstraction environment. Additionally, they capture many details of a true MpSoC device, specifically relating to the application behavior in the presence of interconnect and, when taking into account the IPs' hardware characteristics and OS properties.

In the ARTS framework, we have focused on understanding the impact of NoC in conjunction with IP selection, application mapping and OS dynamics on system performance (memory peaks, PE utilization, etc). Initial results show the potential of the framework in providing a flexible and fast way to instantiate these different components. Via case studies we have attempted to investigate a couple of design problem associated with mobile multimedia terminal.

In the RIPE framework, we have provided an accurate IP emulation device for performance evaluation NoC and prototyping IPs under design. A thorough validation of the framework under diverse conditions in terms of context switching, synchronization and architecture instances has proven the applicability of the design methodology.

Overall, the body of work presented in this thesis, can address a class of problems associated with network MpSoC in a mixed abstraction environment, such as: impact of NoC topology and protocol on the application flow, impact of OS scheduling on NoC traffic density, etc. The two frameworks, presented here allow extensive design space exploration capabilities in their respective abstraction. More importantly, their concepts and the implementation could allow the understanding of the percolation of design decisions made at higher abstraction, to lower levels of abstraction in a predictable and timely fashion.

Part II

Body

Chapter 3

Overview of Networked MPSoC

This chapter consists of the following papers.

#1. Tobais Bjerregaard, and Shankar Mahadevan. "A Survey of Research and Practices of Network-on-Chip." *To appear in the Journal of ACM Computing Surveys.* ACM, 2006.

A Survey of Research and Practices of Network-on-Chip

TOBIAS BJERREGAARD
and
SHANKAR MAHADEVAN
Technical University of Denmark

The scaling of microchip technologies has enabled large scale systems-on-chip (SoC). Network-on-chip (NoC) research addresses global communication in SoC, involving: (i) a move from computation-centric to communication-centric design and (ii) the implementation of scalable communication structures. This survey presents a perspective on existing NoC research. We define the following abstractions: system, network adapter, network and link; to explain and structure the fundamental concepts. First, research relating to the actual network design is reviewed. Then system level design and modeling are discussed. We also evaluate performance analysis techniques. The research shows that NoC constitutes a unification of current trends of intra-chip communication, rather than an explicit new alternative.

Categories and Subject Descriptors: A.1 [**Introductory and Survey**]: ; B.4.3 [**Input/Output and Data-Communications**]: Interconnections; B.7.1 [**Integrated Circuits**]: Types and Design Styles; C.5.4 [**Computer System Implementation**]: VLSI Systems; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.0 [**General**]: —*System Architectures*

General Terms: Design
Additional Key Words and Phrases: chip-area networks, communication-centric design, GALS, GSI design, interconnects, network-on-chip, NoC, OCP, on-chip communication, SoC, sockets, system-on-chip, ULSI design

1. INTRODUCTION

Chip design has four distinct aspects: computation, memory, communication and I/O. As processing power has increased and data intensive applications have emerged, the challenge of the communication aspect in single-chip systems, Systems-on-Chip (SoC), has had increasing attention. This survey treats a prominent concept for communication in SoC known as Network-on-Chip (NoC). As will become clear in the following, NoC does not constitute an explicit new alternative for intra-chip communication, but is rather a concept which presents a unification of on-chip communication solutions.

This paper is a joint first author effort, authors in alfabetical order.
S. Mahadevan was funded partly by SoC-MOBINET (IST-2000-30094) and Nokia.
Authors' address: Technical University of Denmark, Informatics and Mathematical Modelling, Richard Petersens Plads, Building 321, DK-2800 Lyngby, Denmark; email:{tob,sm}@imm.dtu.dk
This work is under review with ACM Computing Surveys.

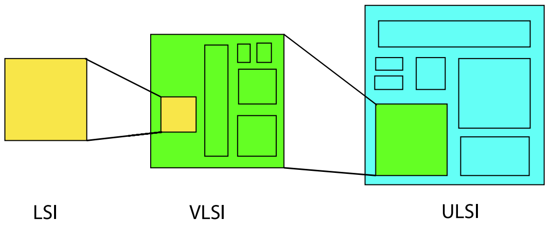


Fig. 1. When a technology matures, it leads to a paradigm shift in system scope. Shown here is the chip scope in LSI, VLSI and ULSI, the sequence of technologies leading to the enabling of SoC designs.

In this section we shall first briefly review the history of microchip technology that has led to a call for NoC based designs. With our minds on intra-chip communication, we will then look at a number of key issues of large-scale chip design, and finally show how the NoC concept provides a viable solution space to the problems presently faced by chip designers.

1.1 Intra-SoC Communication

The scaling of microchip technologies has lead to a doubling of available processing resources on a single chip every second year. Even though this is projected to slow down to a doubling every three years in the next few years for fixed chip sizes [ITRS 2003], the exponential trend is still in force. Though the evolution is continuous, the system level focus, or system scope, moves in steps. When a technology matures for a given implementation style, it leads to a paradigm shift. Examples of such shifts are moving from room- to rack-level systems (LSI - 1970s) and later from rack- to board-level systems (VLSI - 1980s). Recent technological advances allowing multi million transistor chips (currently well beyond 100M) have lead to a similar paradigm shift from board- to chip-level systems (ULSI - 1990s). The scope of a single chip has changed accordingly, as illustrated in Figure 1. In LSI systems a chip was a component of a system module (e.g. a bitslice in a bitslice processor), in VLSI systems a chip was a system level module (e.g. a processor or a memory), and in ULSI systems a chip constitutes an entire system (hence the term System-on-Chip or SoC). SoC opens up to the feasibility of a wide range of applications making use of massive parallel processing and tightly interdependent processes, some adhering to real-time requirements, bringing into focus new complex aspects of the underlying communication structure. Many of these aspects are addressed by NoC.

There are multiple ways to approach an understanding of NoC. Readers well versed in macro network theory may approach the concept by adapting proven techniques from multicomputer networks. Much work done in this area during the 80s and 90s can readily be built upon. Layered communication abstraction models, and decoupling of computation and communication are relevant issues. There are however, a number of basic differences between on- and off-chip communication. These generally reflect the difference in the cost ratio between wiring and processing resources.

Historically, computation has been expensive and communication cheap. With scaling microchip technologies this changed. Computation is becoming ever cheaper, while communication encounters fundamental physical limitations such as time-of-flight of electrical signals, power-use in driving long wires/cables, etc. In comparison with off-chip, on-chip communication is significantly cheaper. There is room for lots of wires on a chip. Thus the shift to single-chip systems has relaxed system communication problems. However on-chip wires do not scale in the same manner as does transistors, and as we shall see in the following, the cost gap between computation and communication is widening. Meanwhile the differences between on- and off-chip wires make the direct scaling down of traditional multicomputer-networks sub-optimal for on-chip use.

In this survey we attempt to incorporate the whole range of design abstractions while relating to the current trends of intra-chip communication. With the *Giga Transistor Chip* era close at hand, the solution space of intra-chip communication is far from trivial. Below we have summarized a number of relevant key issues. Though not new, we find it worthwhile to go through them, as the NoC concept presents a possible unification of solutions for these. In Section 3 and 4, we will look into the details of research being done in relation to these issues, and their relevance for NoC.

—**Electrical wires.** Even though on-chip wires are cheap in comparison with off-chip wires, on-chip communication is becoming still more costly, in terms of both power and speed. As fabrication technologies scale down, wire resistance per mm is increasing while wire capacitance does not change much, the major part of the wire capacitance being due to edge capacitance [Ho et al. 2001]. For CMOS, the approximate point at which wire delays begin to dominate gate delays, was the 0.25μm generation for aluminum, and 0.18μm for copper interconnects, as first projected in [SIA 1997]. Shrinking metal pitches, in order to maintain sufficient routing densities, is appropriate at the local level where wire lengths also decrease with scaling. But global wire lengths do not decrease, and as local processing cycle times decrease, the time spend on global communication, relative to the time spend on local processing, increases drastically. Thus in future deep submicron (DSM) designs the interconnect effect will definitely dominate performance [Sylvester and Keutzer 2000]. Figure 2 taken from the International Technology Roadmap for Semiconductors [ITRS 2001] shows the projected relative delay for local wires, global wires and logic gates of the near future. Another issue of pressing importance concerns signal integrity. In DSM technologies, the wire models are unreliable, due to issues like fabrication uncertainties, crosstalk, noise sensitivity etc. These issues are especially applicable to long wires.
Due to these effects of scaling, it has become necessary to differentiate between local and global communication, and as transistors shrink the gap is increasing. The need for global communication schemes supporting single-chip systems has emerged.

—**System synchronization.** As chip technologies scale and chip speeds increase, it is becoming harder to achieve global synchronization. The drawbacks of the predominant design style of digital integrated circuits, strict global synchrony, are growing relative to the advantages. The clocktree needed to implement a globally synchronized clock is demanding increasing portions of the power and area budget, and even so the clock skew is claiming an ever larger relative part of the total cycle time available [Oklobdzija and Sparsø 2002][Oberg 2003]. This has triggered work on skew tolerant circuit design [Nedovic et al. 2003], which deals with clockskew by relaxing the need for timing margins, and on the use of optical waveguides for on-chip clock distribution [Piguet et al. 2004], the main purpose being to minimize power usage. Still power hungry skew adjustment techniques such as phase locked loops (PLL) and delay locked loops (DLL), traditionally used for chip-to-chip synchronization, are finding their way into single-chip systems [Kurd et al. 2001][Xanthopoulos et al. 2001].

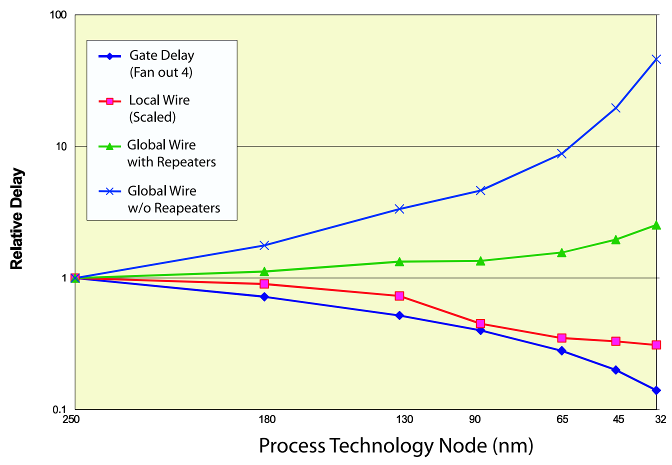


Fig. 2. Projected relative delay for local and global wires and for logic gates of near future technologies [ITRS 2001].

As a reaction to the inherent limitations of global synchrony, alternative concepts such as GALS (Globally Asynchronous Locally Synchronous systems) are being introduced. A GALS chip is made up of patches of locally synchronous islands which communicate asynchronously [Chapiro 1984][Meincke et al. 1999][Muttersbach et al. 2000]. There are two main advantageous aspects of this method. One is the reducing of the synchronization problem to a number of smaller subproblems. The other relates to the integration of different IP (Intellectual Property) cores, easing the building of larger systems from individual blocks with different timing characteristics.

—**Design productivity.** The exploding amount of processing resources available in chip design together with a requirement for shortened design cycles have pushed the productivity requirements on chip designers. Between 1997 and 2002 the market demand reduced the typical design cycle by 50%. As a result of increased chip sizes, shrinking geometries and the availability of more metal layers, the design complexity increased 50 times in the same period [OCPIP 2003a]. To keep up with these requirements, IP reuse is pertinent. A new paradigm for design methodology is needed, which allows the design effort to scale linearly with system complexity.

Abstraction at register transfer level (RTL) was introduced with the ASIC design flow during the 90s, allowing synthesized standard cell design. This made it possible to design large chips within short design cycles, and synthesized RTL design is at present the defacto standard for making large chips quickly. But the availability of on-chip resources is outgrowing the productivity potential of even the ASIC design style. In order to utilize the exponential growth in number of transistors on each chip, even higher levels of abstraction must be applied. This can be done by introducing higher level communication abstractions, making for a layered design methodology enabling a partitioning of the design effort into minimally interdependent subtasks. Support for this at the hardware level includes standard communication sockets, allowing IP cores from different vendors to be plugged effortlessly together. Also, the development of design techniques to further increase the productivity of designers, is important. New electronic design automation

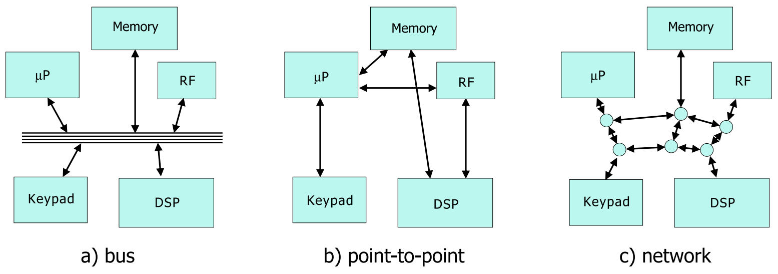


Fig. 3. Examples of communication structures in Systems-on-Chip. a) traditional bus-based communication, b) dedicated point-to-point links, c) a chip area network.

(EDA) tools are necessary for supporting a design flow which make efficient use of such communication abstraction and design automation techniques, and which make for seamless iterations across all abstraction levels. Pertaining to this, the complex, dynamic interdependency of data – arising when using a shared media for data traffic – threatens to foil the efforts of obtaining minimal interdependence between IP cores. Without special quality-of-service (QoS) support, the performance of data communication may become unwarrantly arbitrary [Goossens et al. 2005].
To ensure the effective exploitation of technology scaling, intelligent use of the available chip design resources is necessary, at the physical as well as at the logical design level. Enabling means are the development of effective and structured design methods and EDA tools.

As seen above, the major driving factors for the development of global communication schemes are the ever increasing density of on-chip resources, and the drive to utilize these resources with a minimum of effort, as well as the need to counteract physical effects of DSM technologies. The trend is towards a subdivision of processing resources into manageable pieces. This helps reduce design cycle time since the entire chip design process can be divided into minimally interdependent subproblems. Such a modular structure also allows the use of modular verification methodologies, i.e. verification at low abstraction level of cores (and communication network) individually, and at high abstraction level of the system as a whole. Working at a high abstraction level allows a great degree of freedom from lower level issues. It also lends towards a differentiation of local and global communication. As inter-core communication is becoming the performance bottleneck in many multicore applications, the shift in design focus is from a traditional processing-centric to a communication-centric one. One top level aspect of this involves the possibility to save on global communication resources at the application level by introducing communication aware optimization algorithms in compilers [Guo et al. 2000]. System level effects of technology scaling are further discussed in [Catthoor et al. 2004].

A standardized global communication scheme, together with standard communication sockets for IP cores, would make Lego-brick-like plug-and-play design styles possible, allowing good use of the available resources and fast product design cycles.

1.2 NoC in SoC

Figure 3 show some examples of basic communication structures in a sample SoC, e.g. a mobile phone. Since the introduction of the SoC concept in the 90s, the solutions for

Table I. Bus-versus-network arguments (adapted from [Guerrier and Greiner 2000]).

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

SoC communication structures have generally been characterized by custom designed ad hoc mixes of buses and point-to-point links [Lahiri et al. 2001]. The bus builds on well understood concepts and is easy to model. In a highly interconnected multicore system however, it can quickly become a communication bottleneck. As more units are added to it, the power usage per communication event grows as well, due to more attached units leading to higher capacitive load. For multi-master busses, the problem of arbitration is also not trivial. Table I summarizes the pros and cons of buses and networks. A crossbar overcomes some of the limitations of the buses. However, it is not ultimately scalable and as such an intermediate solution. Dedicated point-to-point links are optimal in terms of bandwidth availability, latency and power usage, as they are designed especially for the given purpose. Also, they are simple to design and verify, and easy to model. But the number of links needed increases exponentially as the number of cores increases. Thus an area and possibly a routing problem develops.

From the point of view of design-effort one may argue that in small systems of less than 20 cores a custom designed communication structure is viable. But as the systems grow and the design cycle time requirements decrease, the need for more generalized solutions becomes pressing. For maximum flexibility and scalability, it is generally accepted that a move towards a shared, segmented global communication structure is needed. This notion translates into a data-routing network consisting of communication links and routing nodes, being implemented on the chip. In contrast to traditional SoC communication methods outlined above, such a distributed communication media scales well with chip size and complexity. Additional advantages include increased aggregated performance by exploiting parallel operation.

From a technological perspective, a similar solution is reached: in DSM chips, long wires must be segmented in order to avoid signal degradation, and busses are implemented as multiplexed structures in order to reduce power and increase responsiveness. Hierarchical bus structures are also common, as a means to adhere to the given communication requirements. The next natural step is to increase throughput by pipelining these structures. Wires become pipelines and bus-bridges become routing nodes. Expanding on a structure using these elements, one gets a simple network.

A common concept for segmented SoC communication structures is based on networks. This is what is known as Network-on-Chip (NoC) [Agarwal 1999][Guerrier and Greiner 2000][Dally and Towles 2001][Benini and Micheli 2002][Jantsch and Tenhunen 2003]. As seen above, the distinction between different communication solutions is fading. NoC is seen to be a unifying concept rather than an explicit new alternative. In the research community, there are two widely held perceptions of NoC: (i) NoC as a subset of SoC, and (ii) NoC as an extension of SoC. In the first view, NoC is defined strictly as the data-forwarding communication fabric, i.e. the network and methods and modules used in accessing the network. In the second view NoC is defined more broadly, also to encompass issues dealing with the application, system architecture, and its impact on communication or vice versa.

1.3 Outline

The purpose of this survey is to clarify the NoC concept and to map the scientific efforts made into the area of NoC research. We will identify general trends, and explain a range of issues which are important for state-of-the-art global chip-level communication. In doing so we primarily take the first view of NoC, i.e. it being a subset of SoC, to focus and structure the diverse discussion. From our perspective, the view of NoC as an extension of SoC muddles the discussion with topics common to any large-scale IC design effort such as: partitioning and mapping application, co-design, compiler choice, etc.

The rest of the survey is organized as follows. In Section 2 we will discuss the basics of NoC. We will give a simple NoC example, address some relevant system level architectural issues, and relate the basic building blocks of NoC to abstract network layers, and to research areas. In Section 3 we will go into more details of existing NoC research. This section is partitioned according to the research areas defined in Section 2. In Section 4 we discuss high abstraction level issues such as design space exploration and modeling. These are issues often applicable to NoC only in the view of it being an extension of SoC, but we treat specifically issues of relevance to NoC-based designs and not to large scale IC designs in general. In Section 5 performance analysis is addressed. Section 6 presents a set of case studies, describing a number of specific NoC implementations, and Section 7 summarizes the survey.

2. NOC BASICS

In this section the basics of NoC are uncovered. First a component based view will be presented, introducing the basic building blocks of a typical NoC. Then we shall look at system level architectural issues relevant to NoC-based SoC designs. After this, a layered abstraction based view will be presented, looking at network abstraction models, in particular OSI, and the adaption of such for NoC. Using the foundations established in this section, we will go into further details of specific NoC research in Section 3.

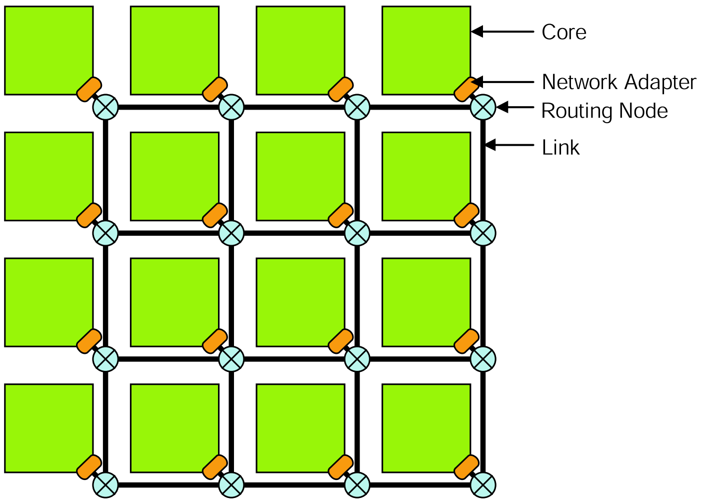


Fig. 4. Topological illustration of a 4-by-4 grid structured NoC, indicating the fundamental components.

2.1 A Simple NoC Example

Figure 4 shows a sample NoC structured as a 4-by-4 grid, which provides global chip-level communication. Instead of busses and dedicated point-to-point links, a more general scheme is adapted, employing a grid of *routing nodes* spread out across the chip, connected by communication *links*. For now we will adapt a simplified perspective in which the NoC contains the following fundamental components:

—**Network Adapters** implement the interface by which *cores* (IP blocks) connect to the NoC. Their function is to decouple computation (the cores) from communication (the network).

—**Routing Nodes** route the data according to chosen protocols. They implement the routing strategy.

—**Links** connect the nodes, providing the raw bandwidth. They may consist of one or more logical or physical channels.

Figure 4 covers only the topological aspects of the NoC. The NoC in the figure could thus employ packet or circuit switching or something entirely different, and be implemented using asynchronous, synchronous or other logic. In Section 3 we will go into details of specific issues with an impact on the network performance.

2.2 Architectural Issues

The diversity of communication in the network is affected by architectural issues such as system composition and clustering. These are general properties of SoC, but since they have direct influence on the design of the system level communication infrastructure we find it worthwhile to go through them here.

Figure 5 illustrates how system composition may be categorized along the axes of *homogenity* and *granularity* of system cores. The figure also clarifies a basic difference between NoC and networks for more traditional parallel computers; the latter have generally

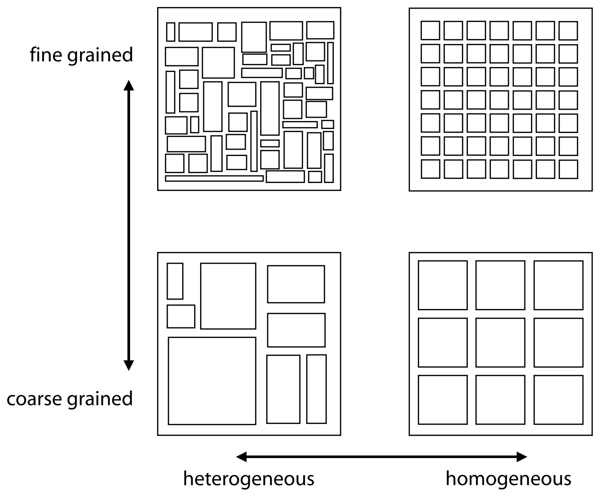


Fig. 5. System composition may be categorized along the axes of homogenity and granularity of system components.

been homogeneous and coarse grained, where as NoC-based systems implement a much higher degree of variety in composition, and in traffic diversity.

Clustering deals with the localization of portions of the system. Such localization may be logical or physical. Logical clustering can be a valuable programming tool. It can be supported by the implementation of hardware primitives in the network, e.g. flexible addressing schemes or virtual connections. Physical clustering, based on pre-existing knowledge of traffic patterns in the system, can be used to minimize global communication, thereby minimizing the total cost of communicating, power- and performance-wise.

Generally speaking, *reconfigurability* deals with the ability to allocate available resources for specific purposes. In relation to NoC-based systems, reconfigurability concerns how the NoC, a flexible communication structure, can be used to make the system reconfigurable from an application point of view. A configuration can be established e.g. by programming connections into the NoC. This resembles the reconfigurability of an FPGA, though NoC-based reconfigurability is most often of coarser granularity. In NoC, the reconfigurable resources are the routing nodes and links rather than wires.

Much research work has been done on architecturally oriented projects, in relation to NoC-based systems. Such work is important in that it provides motivation for the introduction of NoC. The main issue in architectural decisions is the balancing of flexibility, performance and hardware costs of the system as a whole. As the underlying technology advances, the trade-off spectrum is continually shifted, and the viability of the NoC concept has opened up to a communication-centric solution space, which is what current system level research explores.

At one corner of the architecural space outlined in Figure 5, is the Pleiades architecture [Zhang et al. 2000] and its instantiation the Maia processor. A microprocessor is combined with a relatively fine grained heterogeneous collection of ALUs, memories, FPGAs, etc. An interconnection network allows arbitrary communication between modules of the system. The network is hierarchical and employs clustering in order to provide the

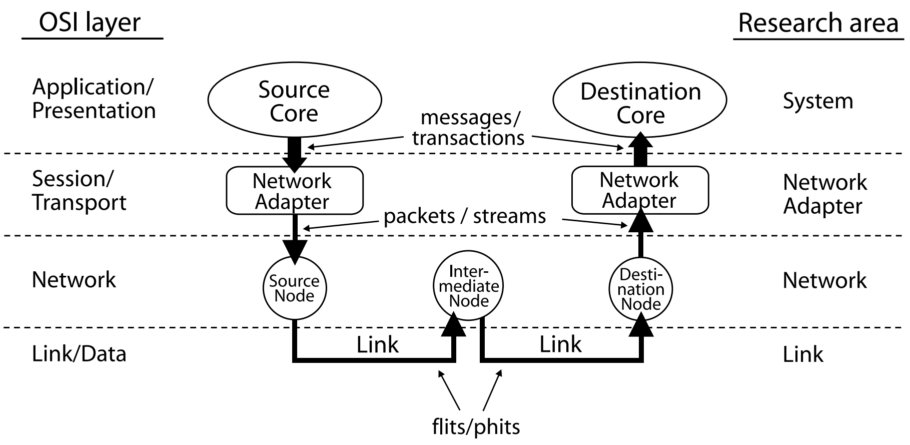


Fig. 6. The flow of data from source to sink, through the NoC components, with an indication of the types of datagrams and research area.

required communication flexibility while maintaining good energy-efficiency.

At the opposite corner are a number of works, implementing homogeneous, coarse grained multiprocessors. In the Smart Memories [Mai et al. 2000] a hierarchical network is used, with physical clustering of four processors. The flexibility of the local cluster network is used as a means for reconfigurability, and the effectiveness of the platform is demonstrated by mimicking two machines on far ends of the architectural spectrum, the Imagine streaming processor and Hydra multiprocessor, with modest performance degradation. The global NoC is not described however. In the RAW architecture [Taylor et al. 2002] on the other hand, the NoC which interconnects the processor tiles is described in detail. It consists of a static network, in which the communication is preprogrammed cycle by cycle, and a dynamic network. The reason for implementing two physically separate networks is to accommodate different types of traffic in general purpose systems (see Section 4.3 concerning traffic characterization). The Eclipse [Forsell 2002] is another similarly distributed multiprocessor architecture, in which the interconnection network plays an important role. Here, the NoC is a key element in supporting a sofisticated parallel programming model.

2.3 Network Abstraction

The term NoC is used in research today in a very broad sense ranging from gate-level physical implementation, across system layout aspects and applications, to design methodologies and tools. A major reason for the wide-spread adaptation of network terminology lies in the readily available and widely accepted abstraction models for networked communication. The OSI model of layered network communication can easily be adapted for NoC usage, as done in [Benini and Micheli 2001] and [Arteris 2005]. In the following we will look at network abstraction, and make some definitions to be used later in the survey.

To better understand the approaches of different groups involved in NoC, we have partitioned the spectrum of NoC research into four areas: 1) System, 2) Network Adapter, 3) Network and 4) Link research. Figure 6 shows the flow of data through the network, indicating the relation between these research areas, the fundamental components of NoC and the OSI layers. Also indicated is the basic datagram terminology.

The *System* encompasses applications (processes) and architecture (cores and network). At this level, most of the network implementation details may still be hidden. Much research done at this level is applicable to large scale SoC design in general. The *Network Adapter* (NA) decouples the cores from the network. It handles the end-to-end flow control, encapsulating the *messages* or *transactions* generated by the cores for the routing strategy of the Network. These are broken into *packets* which contain information about their destination, or connection-oriented *streams* which do not, but have had a path setup prior to transmission. The NA is the first level which is 'network aware'. The *Network* consists of the routing nodes, links, etc, defining the topology and implementing the protocol and the node-to-node flow control. The lowest level is the *Link* level. At this level, the basic datagram are *flits* (**fl**ow control un**its**), node level atomic units from which packets and streams are made up. Some researchers operate with yet another subdivision, namely *phits* (**ph**ysical un**its**), which are the minimum size datagram that can be transmitted in one link transaction. Most commonly flits and phits are equivalent, though in a network employing highly serialized links, each flit could be made up of a sequence of phits. Link level research deals mostly with encoding and synchronization issues. The presented datagram terminology seems to be generally accepted, though no standard exists.

In a NoC, the layers are generally more closely bound than in a macro network. Issues arising often have a more physically related flavor, even at the higher abstraction levels. OSI specifies a protocol stack for multicomputer networks. Its aim is to shield higher levels of the network from issues of lower levels, in order to allow communication between independently developed systems, e.g. of different manufacturers, and to allow on-going expansion of systems. In comparison with macro networks, NoC benefits from the system composition being completely static. The network can be designed based on knowledge of the cores to be connected, and possibly also on knowledge of the characteristics of the traffic to be handled, as demonstrated in e.g. [Bolotin et al. 2004] and [Goossens et al. 2005]. Awareness of lower levels can be beneficial, as it can lead to higher performance. The OSI layers, which are defined mainly on a basis of pure abstraction of communication protocols, thus cannot be directly translated into the research areas defined here. With this in mind, the relation established in Figure 6 is to be taken as a conceptual guideline.

3. NOC RESEARCH

In this section we provide a review of the approaches of various research groups. Figure 7 illustrates a simplified classification of this research. The text is structured based on the layers defined in Section 2.3. Since we consider NoC as a subset of SoC, system level research is dealt with separately in Section 4.

3.1 Network Adapter

The purpose of the Network Adapter (NA) is to interface the core to the network, and make communication services transparently available with a minimum of effort from the core. At this point, the boundary between computation and communication is specified.

As illustrated in Figure 8, the NA component implements a Core Interface (CI) at the core side and a Network Interface (NI) at the network side. The function of the NA is to provide high-level communication services to the core by utilizing primitive services provided by the network hardware. Thus the NA *decouples* the core from the network,

NoC Research
- System Level
  - Design Methodology and Abstraction: co-exploration, modeling.
  - Architecture Domain: system composition, clustering, reconfigurability.
  - Traffic Characterization: latency-critical, data-streams and best-effort.
- Network Adapters
  - Functionality: encapsulation, service managment.
  - Sockets: plug and play, IP reuse.
- Network
  - Topology: regular vs. irregular topologies, switch layout.
  - Protocol: routing, switching, control schemes.
  - Flow control: deadlock avoidance, virtual channels, buffering.
  - Quality-of-Service: service classification and negotiation.
  - Features: error-correction, broadcast/multicast/narrowcast, virtual wires.
- Link Level
  - synchronization, reliability, encoding.

Fig. 7. NoC Research Area Classification. This classification, which also forms the structure of this section, is meant as a guideline to evaluate NoC research, and not as a technical categorization.

implementing the network end-to-end flow control, facilitating a layered system design approach. The level of decoupling may vary. A high level of decoupling allows for easy reuse of cores. This makes possible a utilization of the exploding resources available to chip designers, and greater design productivity is achieved. On the other hand, a lower level of decoupling (a more network aware core) has the potential to make more optimal use of the network resources.

In this section, we first address the use of standard sockets. We then discuss the abstract functionality of the NA. Finally, we talk about some actual NA implementations, which also address issues related to timing and synchronization.

3.1.1 *Sockets.* The CI of the NA may be implemented to adhere to a SoC socket standard. The purpose of a socket is to orthogonalize computation and communication. Ideally a socket should be completely NoC implementation agnostic. This will facilitate the greatest degree of reusability, because the core adheres to the specification of the socket alone, independently of the underlying network hardware. One commonly used socket is the *Open Core Protocol* (OCP) [OCPIP 2003b][Haverinen et al. 2002]. The OCP specification defines a flexible family of core-centric protocols for use as native core interface in on-chip systems. The three primary properties envisioned in OCP include: (i) architecture independent design reuse, (ii) feature specific socket implementation, and (iii) simplification of system verification and testing. OCP addresses not only data-flow signaling, but also uses related to errors, interrupts, flags and software flow control, control and status, and test. Another previously proposed standard is the *Virtual Component Interface* (VCI) [VSI Alliance 2000] used in the SPIN [Guerrier and Greiner 2000] and Proteo [Siguenza-Tortosa et al. 2004] NoCs. In [Radulescu et al. 2004] support for the Advanced eXtensible Interface (AXI) [ARM 2004] and Device Transaction Level (DTL) [Philips Semiconductors 2002] protocols was also implemented in an NA design.

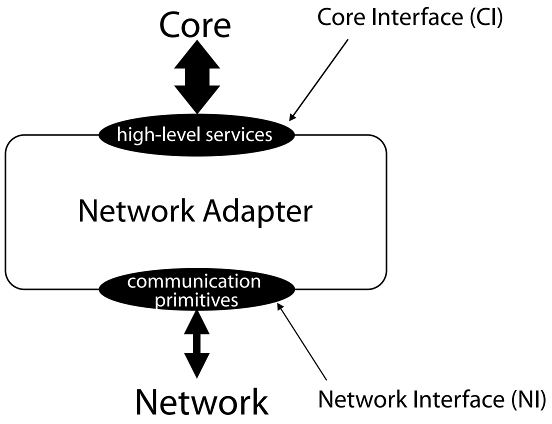


Fig. 8. The Network Adapter (NA) implements two interfaces, the Core Interface (CI) and the Network Interface (NI). The CI provides high-level communication services to the core, based on primitive services provided by the network through the NI.

3.1.2 *NA Services.* Basically, the NA provides *encapsulation* of the traffic for the underlying communication media and *management* of services provided by the network. Encapsulation involves handling of end-to-end flow control in the network. This may include global addressing and routing tasks, re-order buffering and data acknowledgement, buffer management to prevent network congestion, e.g. based on credits, packet creation in a packet-switched network, etc.

Cores will content for network resources. These may be provided in terms of service quantification as e.g. bandwidth and/or latency guarantees (see also Sections 3.2.4 and 5). Service management concerns setting up circuits in a circuit-switched network, book keeping tasks such as keeping track of connections, and matching responses to requests. Another task of the NA could be to negotiate the service needs between the core and the network.

3.1.3 *NA Implementations.* Many researchers have realized that the NA holds the key to unlocking the potential of a NoC. A clear understanding of its role is essential to successful NoC design. Muttersbach, Villiger and Fichtner [Muttersbach et al. 2000] address synchronization issues, proposing a design of an asynchronous wrapper for use in a practical GALS design. Here the synchronous modules are equipped with asynchronous wrappers which adapt their interfaces to the self-timed environment. The packetization occurs within the synchronous module. The wrappers are assembled from a concise library of predesigned technology-independent elements and provide high speed data transfer. Another mixed asynchronous/synchronous NA architecture is proposed in [Bjerregaard et al. 2005]. Here, a synchronous OCP interface connects to an asynchronous, message-passing NoC. Packetization is performed in the synchronous domain, while sequencing of flits is done in the asynchronous domain. This makes the sequencing independent of the speed of the OCP interface, while still taking advantage of synthesized synchronous design, for maintaining a flexible packet format. Thus the NA leverages the advantages particular to either circuit design style. In [Radulescu et al. 2004] a complete NA design for the ÆTHEREAL NoC is presented, which also offers a shared-memory abstraction to the cores. It provides

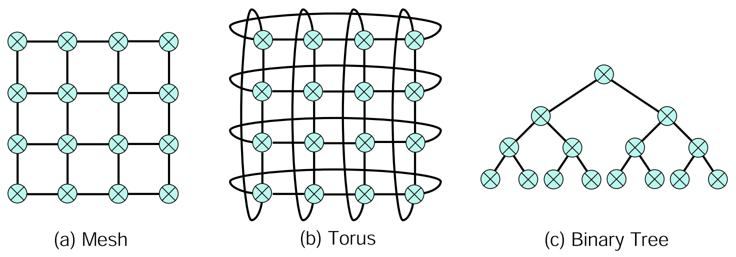


Fig. 9. Regular forms of topologies scale predictably with regards to area and power. Examples are (a) 4-ary 2-cube mesh, (b) 4-ary 2-cube torus and (c) binary (2-ary) tree.

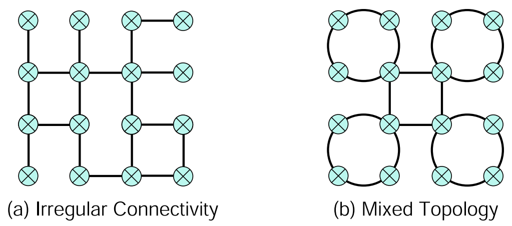


Fig. 10. Irregular forms of topologies are derived by altering the connectivity of a regular structure such as shown in (a) where certain links from a mesh have been removed, or by mixing different topologies such as in (b) where a ring co-exists with a mesh.

compatibility to existing on-chip protocols such as AXI, DTL and OCP, and allows easy extension to other future protocols as well.

However, the cost of using standard sockets is not trivial. As demonstrated in the HERMES NoC [Ost et al. 2005], the introduction of OCP makes the transactions upto 50% slower compared to the native core interface. An interesting design trade-off issue is the partitioning of the NA functions between software (possibly in the core) and hardware (most often in the NA). In [Bhojwani and Mahapatra 2003] a comparison of software and hardware implementations of the packetization task was undertaken, the software taking 47 cycles to complete, while the hardware version taking only 2 cycles. In [Radulescu et al. 2004] a hardware implementation of the entire NA introduces a latency overhead of between 4 and 10 cycles, pipelined to maximize throughput. The NA in [Bjerregaard et al. 2005] takes advantage of the low forward latency of clockless circuit techniques, introducing an end-to-end latency overhead of only 3 to 5 cycles for writes and 6 to 8 cycles for reads, which include data return.

3.2 Network Level

The job of the network is to deliver messages from their source to their designated destination. This is done by providing the hardware support for basic communication primitives. A well-built network, as noted by Dally and Towles [Dally and Towles 2001], should appear as a logical wire to its clients. An on-chip network is defined mainly by its topology and the protocol implemented by it. Topology concerns the layout and connectivity of the nodes and links on the chip. Protocol dictates how these nodes and links are used.

3.2.1 *Topology.* One simple way to distinguish different regular topologies is in terms of *k-ary n-cube* (grid-type), where *k* is the degree of each dimension (i.e. number of nodes) and *n* is the dimensions (Figure 9), first described by Dally [Dally 1990] for multicomputer networks. The *k-ary tree* and the *k-ary n-dimensional* fat tree are two alternate regular forms of networks explored for NoC. The network area and power consumption scales predictably for increasing size of regular forms of topology. Most NoCs implement regular forms of network topology, that can be laid out on a chip surface (a 2-dimensional plane) e.g. k-ary 2-cube, commonly known as grid-based topologies. The Octagon NoC demonstrated in [Karim et al. 2001][Karim et al. 2002] is an example of a novel regular NoC topology. Its basic configuration is a ring of 8 nodes connected by 12 bi-directional links, which provides two-hop communication between any pair of nodes in the ring, and a simple, shortest-path routing algorithm. Such rings are then connected edge to edge, to form a larger, scalable network. For more complex structures such as trees, finding the optimal layout is a challenge in its own right.

Besides the form, the nature of links adds an additional aspect to the topology. In k-ary 2-cube networks, popular NoC topologies based on the nature of link are: the mesh which uses bidirectional links, and torus using unidirectional links. For a torus, a folding can be employed to reduce long wires. In the NOSTRUM NoC presented in [Millberg et al. 2004] a folded torus is discarded in favor of a mesh, with the argument that it has longer delays between routing nodes. Figure 9 shows examples of regular forms of topology. Generally, mesh topology makes better use of links (utilization) while tree-based topologies are useful for exploiting locality of traffic.

Irregular forms of topologies are derived by mixing different forms, in a hierarchical, hybrid or asymmetric fashion, as seen in Figure 10. Irregular forms of topologies scale non-linearly with regards to area and power. These are usually based on the concept of clustering. A small private/local network often implemented as a bus, [Mai et al. 2000] and [Wielage and Goossens 2002], for local communication with k-ary 2-cube global communication is a favored solution. In [Pande et al. 2005], the impact of clustering on five NoC topologies is presented. It shows 20% to 40% reduction in bit-energy for the same amount of throughput, due to traffic localization.

With regards to the presence of a local traffic source or sink connected to the node, *direct networks* are those that have at least one core attached to each node, *indirect networks* on the other hand have a subset of nodes not connected to any core, performing only network operations; as is generally seen in tree-based topology where cores are connected at the leaf nodes. The examples of indirect tree-based networks are fat-tree in SPIN [Guerrier and Greiner 2000] and butterfly in [Pande et al. 2003]. The fat-tree used in SPIN is proven in [Leiserson 1985] to be most hardware efficient compared to any other network.

For alternate classifications of topology the reader is referred to [Aggarwal and Franklin 2002], [Jantsch 2003] and [Culler et al. 1998]. Culler in [Culler et al. 1998] combines protocol and geometry, to bring out a new type of classification which is defined as topology.

With regards to the routing nodes, a layout trade-off is the *thin switch* vs *square switch* presented by Kumar et al [Kumar et al. 2002]. Figure 11 illustrates the difference between these two layout concepts. A thin switch is distributed around the cores and wires are routed across them. A square switch is placed on the crossings of dedicated wiring channels between the cores. It was found that the square switch is better for performance and bandwidth while the thin switch requires relatively low area. The area overhead required to

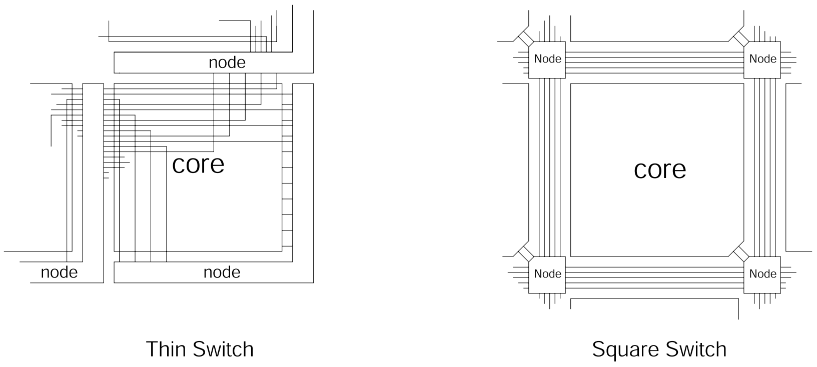


Fig. 11. Two layout concepts. The thin switch is distributed around the cores and wires are routed across it. The square switch is placed on the crossings in dedicated channels between the cores.

implement a NoC is in any case expected to be modest. The processing logic of the router, for a packet switched network, is estimated to be approximately between 2.0% [Pande et al. 2003] to 6.6% [Dally and Towles 2001] of the total chip area. In addition to this, the wiring uses a portion of the upper two wiring layers.

3.2.2 *Protocol.* The protocol concerns the strategy of moving data through the NoC. We define switching as the mere transport of data, while routing is the intelligence behind, i.e. it determines the path of the data transport. This is in accordance with Culler et al [Culler et al. 1998]. In the following these and other aspects of protocol commonly addressed in NoC research, are discussed:

—**Circuit vs packet switching:** Circuit switching involves the circuit from source to destination being setup and reserved until the transport of data is complete. Packet switched traffic on the other hand is forwarded on a per-hop basis, each packet containing routing information as well as data.

—**Connection-oriented vs connection-less:** Connection-oriented mechanisms involve a dedicated (logical) connection path being established prior to data transport. The connection is then terminated upon completion of communication. In connection-less mechanisms the communication occurs in a dynamic manner, with no prior arrangement between the sender and the receiver. Thus circuit switched communication is always connection-oriented, whereas packet switched communication may be either connection-oriented or connection-less.

—**Deterministic vs adaptive routing:** In a deterministic routing strategy, the traversal path is determined by its source and destination alone. Popular deterministic routing schemes for NoC are source routing and X-Y routing (2-d dimension order routing). In source routing, the source core specifies the route to the destination. In X-Y routing the packet follow the rows first then along the columns toward destination or vice versa. In an adaptive routing strategy the routing path is decided on a per-hop basis. Adaptive schemes involve dynamic arbitration mechanisms, e.g. based on local link congestion. This results in more complex node implementations, but offers benefits like dynamic load balancing.

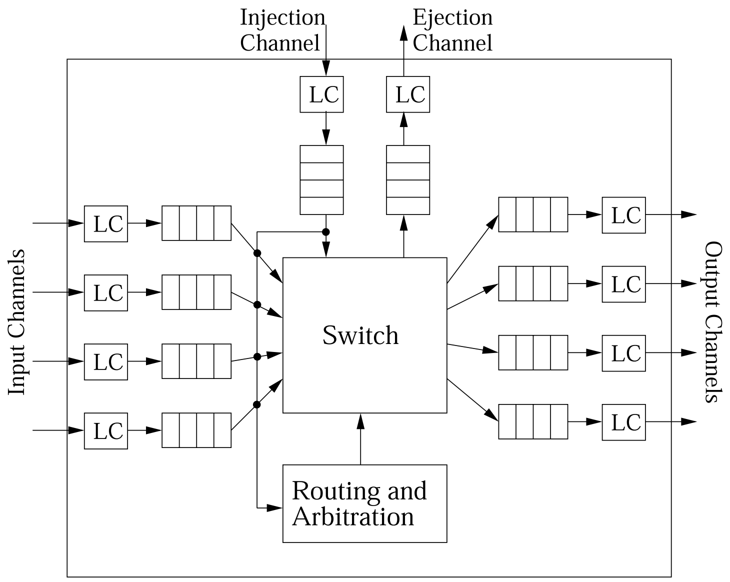


Fig. 12. Generic router model. LC = link controller ([Duato et al. 2003], fig. 2.1)

—**Minimal vs non-minimal routing:** A routing algorithm is minimal if it always chooses among shortest paths toward the destination; otherwise it is non-minimal.

—**Delay vs loss:** In the delay model datagrams are never dropped. This means that the worst that can happen is the data being delayed. In the loss model datagrams can be dropped. In this case the data needs to be retransmitted. The loss model introduces some overhead in that the state of the transmission, successful or failed, must somehow be communicated back to the source. There are however some advantages involved in dropping datagrams, e.g. as a means of resolving network congestion.

—**Central vs distributed control:** In centralized control mechanisms, routing decisions are made globally, e.g. bus arbitration. In distributed control, most common for segmented interconnection networks, the routing decisions are made locally.

The protocol defines the use of the available resources, and thus the node implementation reflects design choices based on the above listed terms. In Figure 12, taken from [Duato et al. 2003], Duato et al. have clearly identified the major components of any routing node i.e.: buffers, switch, routing and arbitration unit and link controller. The switch connects the input buffers to the output buffers, while the routing and arbitration unit implements the algorithm that dictates these connections. In a centrally controlled system, the routing control would be common for all nodes, and a strategy might be chosen which guarantees no traffic contention. Thus no arbitration unit would be necessary. Such a scheme can be employed in a NoC in which all nodes have a common sense of time, as done in [Millberg et al. 2004]. Here the NOSTRUM NoC implements an explicit time division multiplexing mechanism which the authors call *Temporally Disjoint Networks* (TDN). Packets cannot collide, if they are in different TDNs. This is similar to the slot allocation mechanism in the ÆTHEREAL NoC [Goossens et al. 2005].

The optimal design of the switching fabric itself relates to the services offered by the router. In [Kim et al. 2005] a crossbar switch is proposed, which offers adaptive bandwidth control. This is facilitated by adding an additional bus, allowing the crossbar to be bypassed during periods of congestion. Thus, the switch is shown to improve the throughput and latency of the router by up to 27% and 41% respectively, at a modest area and power overhead of 21% and 15% respectively. In [Bjerregaard and Sparsø 2005a] on the other hand, a non-blocking switch is proposed, which allows for hard performance guarantees, when switching connections within the router (more details in Section 3.2.4). By utilizing the knowledge that only a limited number of flits can enter the router through each input port, the switch can be made to scale linearly rather than exponentially, with the number of connections on each port. In [Leroy et al. 2005] a switch similarly provides guaranteed services. This switch however switches individual wires on each port, rather than virtual connections.

A quantitative comparison of connection-oriented and connection-less scheme for MPEG-2 Video Decoder is presented in [Harmanci et al. 2005]. The connection-oriented scheme is based on ÆTHEREAL, while the connection-less scheme is based on DiffServ - a priority based packet scheduling NoC. The conclusions of tests, conducted in the presence of background traffic noise, show that (i) the individual end-to-end delay is lower in connection-less than in connection-oriented one, due to better adaptation of the first approach to variable bit-rates of the MPEG video flows, and (ii) the connection-less schemes present a higher stability towards a wrong decision in the type of service to be assigned to a flow.

Concerning the merits of adaptive routing, versus deterministic, there are different opinions. In [Neeb et al. 2005] a comparison of deterministic (dimension-order) and adaptive (negative first and planar-adaptive) routing, applied to mesh, torus and cube networks, was made. For chips performing interleaving in high throughput channel decoder wireless applications, dimension-order routing scheme was found to be inferior compared to adaptive schemes, when using lower dimension NoCs topologies. However, it was shown to be the best-choice, due to low area and high thoughput characteristics, for higher dimension NoC topologies. The impact on area and throughput, of input and output buffer queues in the router, was also discussed. In [de Mello et al. 2004], the performance of minimal routing protocols in the HERMES [Moraes et al. 2004] NoC were investigated: one deterministic protocol (XY-routing) and three partially adaptive protocols (west-first, north-last and negative-first routing). While the adaptive protocols can potentially speed up the delivery of individual packets, it was shown that the deterministic protocol was superior to the adaptive ones from a global point. The reason is that adaptive protocols tend to concentrate the traffic in the center of the network, resulting in increased congestion here.

The wide majority of NoC research is based on packet switching networks. In addition, most are delay-based since the overhead of keeping account of packets being transmitted, and of retransmitting dropped packets is high. Most often connection-less routing is employed for best-effort (BE) traffic (Section 4.3), while connection-oriented routing is used to provide service guarantees (Section 3.2.4). In SoCBUS [Sathe et al. 2003] a different approach is taken, in that a connection-oriented strategy is used to provide BE traffic routing. Very simple routers establish short lived connections, setup using BE routed control packets, which provide a very high throughput of 1.2GHz in a 0.18μm CMOS process. Drawbacks are the time spent during the setup phase, which requires a path acknowledge,

Table II. Protocol-Datagram Table.

Protocol	Router		Stalling
	Latency	Storage	
store-and-forward	packet	packet	at two nodes and the link between them
wormhole	header	header	at all nodes and links spanned by the packet
virtual cut-through	header	packet	at the local node

and the fact that only a single connection can be active on each link at any given time. A similarly connection-oriented NoC is aSoC [Liang et al. 2000], which implements a small *reconfigurable communication processor* in each node. This processor has interconnect memory that programs the crossbar for data transfer from different sources across the node on each communication cycle.

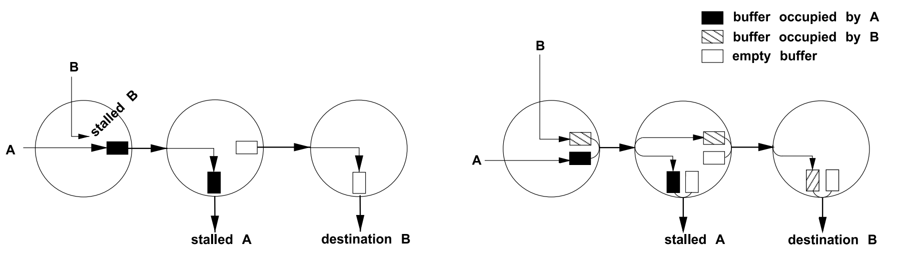
The most common forwarding strategies are store-and-forward, wormhole and virtual cut-through. These will be explained below. Table II summarizes the latency penalty and storage cost in each node for each of these schemes.

Store-and-forward. Store-and-forward routing is a packet switched protocol, in which the node stores the complete packet and forwards it based on the information within its header. Thus the packet may stall if the router in the forwarding path does not have sufficient buffer space. The CLICHE [Kumar et al. 2002] is an example of a store-and-forward NoC.

Wormhole. Wormhole routing combines packet switching with the data streaming quality of circuit switching, to attain a minimal packet latency. The node looks at the header of the packet to determine its next hop and immediately forwards it. The subsequent flits are forwarded as they arrive. This causes the packet to *worm* its way through the network, possibly spanning a number of nodes, hence the name. The latency within the router is not that of the whole packet. A stalling packet however has the unpleasantly expensive side effect of occupying all the links that the worm spans. In Section 3.2.3 we shall see how virtual channels can relieve this side effect at a marginal cost. In [Al-Tawil et al. 1997] a well structured survey of wormhole routing techniques is provided and a comparison between a number of schemes is made.

Virtual cut-through. Virtual cut-through routing has a forwarding mechanism similar to that of wormhole routing. But before forwarding the first flit of the packet, the node waits for a guarantee that the next node in the path will accept the entire packet. Thus if the packet stalls, it aggregates in the current node without blocking any links.

While macro networks usually employ store-and-forward routing, the prevailing scheme for NoC is wormhole routing. Advantages are low latency and the avoidance of area costly buffering queues. A special case of employing single flit packets is explored in [Dally and Towles 2001]. Here the data and header bits of the packets are transmitted separately and in parallel across a link, and the data path is quite wide (256 bits). Each flit is thus a packet in its own right, holding information about its destination. Hence, unlike wormhole routing, the stream of flits may be interlaced with other streams and stalling is restricted to the local node. Still single flit latency is achieved. The cost is a higher header-to-payload ratio, resulting in larger bandwidth overhead.



Incomming data stream B is stalled by stream A   Virtual channels allow stream B to pass stalled stream A

Fig. 13. Using virtual channels, independently buffered logical channels sharing a physical link, to prevent stalls in the network. Streams on different VCs can pass each other, while streams sharing buffer queues may stall.

3.2.3 *Flow Control.* Peh and Dally have defined flow control as the mechanism that determines the packet movement along the network path [Peh and Dally 2001]. Thus it encompasses both global and local issues. Flow control mainly addresses the issue of ensuring correct operation of the network. In addition, it can be extended to include also issues on utilizing network resources optimally and providing predictable performance of communication services. Flow control primitives thus also form the basis of differentiated communication services. This will be discussed further in Section 3.2.4

In the following, we first discuss the concept of virtual channels and their use in flow control. We then discuss a number of works in the area, and finally we address buffering issues.

Virtual channels (VCs): VCs is the sharing of a physical channel by several logically separate channels with individual and independent buffer queues. Generally, between 2 and 16 VCs per physical channel have been proposed for NoC. Their implementation results in an area and possibly also power, and latency overhead, due to the cost of control and buffer implementation. There are however a number of advantageous uses. Among these are:

—**Avoiding deadlocks.** Since VCs are not mutually dependent on each other, adding VCs to links, and choosing the routing scheme properly, one may break cycles in the resource dependency graph (see below) [Dally and Seitz 1987].

—**Optimizing wire utilization.** In future technologies, wire costs are projected to dominate over transistor costs [ITRS 2003]. Letting several logical channels share the physical wires, the wire utilization can be greatly increased. Advantages include reduced leakage power and wire routing congestion.

—**Improving performance.** VCs can generally be used to relax the inter-resource dependencies in the network, thus minimizing the frequency of stalls. In [Dally 1992] it is shown that dividing a fixed buffer size across a number of VCs improve the network performance at high loads. In [Duato and Pinkston 2001] the use of VCs to implement adaptive routing protocols is presented. [Vaidya et al. 2001] and [Cole et al. 2001] discusses the impact and benefit of supporting VCs.

—**Providing diffentiated services.** Quality-of-service (QoS, see Section 3.2.4) can be used as a tool to optimize application performance. VCs can be used to implement such services, by allowing high priority data streams to over take those of lower priority [Felicijan and Furber 2004][Rostislav et al. 2005][Beigne et al. 2005], or by providing guaranteed service levels on dedicated connections [Bjerregaard and Sparsø 2005a].

To ensure correct operation, the flow control of the network must first and foremost avoid deadlock and livelock. Deadlock occurs when network resources (e.g. link bandwidth or buffer space) are suspended waiting for each other to be released, i.e. where one path is blocked leading to other being blocked in a cyclic fashion [Dally and Seitz 1987]. It can be avoided by breaking cyclic dependencies in the resource dependency graph. Figure 13 illustrates how VCs can be used to prevent stalls due to dependencies on shared network resources. It is seen how in a network without VCs, stream B is stalled by stream A. In a network with VCs however, stream B is assigned to a different VC with a separate buffer queue. Thus even though stream A is stalled stream B is enabled to pass.

Livelock occurs when resources constantly change state waiting for other to finish. Livelock is less common but may be expected in networks where packets are reinjected into the network, or where back-stepping is allowed, e.g. during non-minimal adaptive routing.

Methods to avoid deadlock and livelock can be applied either locally at the nodes with support from service primitives e.g. implemented in hardware, or globally by ensuring logical separation of data streams by applying end-to-end control mechanisms. While local control is most widespread, the latter was done in [Millberg et al. 2004] using the concept of Temporally Disjoint Networks which was described in Section 3.2.2. As mentioned above, dimension-ordered routing is a popular choice for NoC, because it provides freedom from deadlock, without the need to introduce VCs. The *turn model* [Glass and Ni 1994] also does this, but allows more flexibility in routing. A related approach is the *odd-even turn model* [Chiu 2000] for designing partially adaptive deadlock-free routing algorithms. Unlike the turn model, which relies on prohibiting certain turns in order to achieve freedom from deadlock, this model restricts the *locations* where some types of turns can be taken. As a result, the degree of routing adaptiveness provided is more even for different source-destination pairs. The ANoC [Beigne et al. 2005] implements this routing scheme.

The work of José Duato has addressed the mathematical foundations of routing algorithms. His main interests have been in the area of adaptive routing algorithms for multicomputer networks. Most of the concepts are directly applicable to NoC. In [Duato 1993] the theoretical foundation for deadlock-free adaptive routing in wormhole networks is given. This builds on early work by Dally, which showed that avoiding cyclic dependencies in the channel dependency graph of a network, deadlock-free operation is assured. Duato expands the theory to allow adaptive routing, and furthermore shows that the absence of cyclic dependencies is too restrictive. It is enough to require the existence of a channel subset which defines a connected routing subfunction with no cycles in its *extended* channel dependency graph. The extended channel dependency graph is defined in [Duato 1993] as a graph for which the arcs are not only pairs of channels for which there is a direct dependency, but also pairs of channels for which there is an indirect dependency. In [Duato 1995] and [Duato 1996] this theory is refined and extended to cover also cut-through and store-and-forward routing. In [Duato and Pinkston 2001] a general theory is presented, which glues together several of the previously proposed theories into a single theoretical framework.

In [Dally and Aoki 1993] Dally has investigated a hybrid of adaptive and deterministic routing algorithms using VCs. Packets are routed adaptively until a certain number of hops have been made in a direction away from the destination. There after, the packets are routed deterministically, in order to be able to guarantee deadlock-free operation. Thus the benefits of adaptive routing schemes are approached, while keeping the simplicity and predictability of deterministic schemes.

Other research has addressed flow control approaches purely for improving performance. In [Peh and Dally 1999] and [Kim et al. 2005] look-ahead arbitration schemes are used to allocate link and buffer access ahead of data arrival, thus reducing the end-to-end latency. This results in increased bandwidth utilization as well. Peh and Dally use virtual channels, and their approach is compared with simple virtual-channel flow control, as described in [Dally 1992]. It shows an improvement in latency of about 15%, across the entire spectrum of background traffic load, and network saturation occurs at a load 20% higher. Kim et al do not use virtual channels. Their approach is shown to improve latency considerably (by 42%) when network load is low (10%) with much less improvement (13%) when network load is high (50%). In [Mullins and Moore 2004] a virtual-channel router architecture for NoC is presented, which optimizes routing latency by hiding control overheads, in a single cycle implementation.

Buffering: Buffers are an integral part of any network router. In by far the most NoC architectures, buffers account for the main part of the router area. As such, it is a major concern to minimize the amount of buffering necessary, under given performance requirements. There are two main aspects of buffers: (i) their size and (ii) their location within the router. In [Kumar et al. 2002] it is shown that increasing the buffer size is not a solution towards avoiding congestion. At best, it delays the onset of congestion, since the throughput is not increased. The performance improved marginally in relation to the power and area overhead. On the other hand, buffers are useful to absorb bursty traffic, thus leveling the bursts.

Tamir and Frazier [Tamir and Frazier 1988] have provided an comprehensive overview of advantages and disadvantages of different buffer configurations (size and location) and additionally proposed a buffering strategy called dynamically allocated multi-queue (DAMQ) buffer. In the argument of input vs. output buffers, for equal performance the queue length in a system with output port buffering is always found to be shorter than the queue length in an equivalent system with input port buffering. This is so, since in a routing node with input buffers, a packet is blocked if it is queued behind a packet whose output port is busy (head-of-the-line-blocking). With regards to centralized buffer pools shared between multiple input and output ports vs distributed dedicated FIFOs, the centralized buffer implementations are found to be expensive in area due to overhead in control implementation, and become bottlenecks during periods of congestion. The DAMQ buffering scheme allows independent access to the packets destined for each output port, while applying its free space to any incoming packet. DAMQ shows better performance than FIFO or statically-allocated shared buffer space per input-output port due to better utilization of the available buffer space especially for non-uniform traffic. In [Rijpkema et al. 2001] a somewhat similar concept called virtual output queuing is explored. It combines moderate cost with high performance at the output queues. Here independent queues are designated to the output channels thus enhancing the link utilization by bypassing blocked packets.

In [Hu and Marculescu 2004a] the authors present an algorithm which sizes the (input) buffers in a mesh-type NoC, on basis of the traffic characteristics of a given application. In all evaluated benchmarks, it was shown how such intelligent buffer allocation resulted in about 85% savings in buffering resources, in comparison with uniform buffer sizes, without any reduction in performance.

3.2.4 *Quality of Service (QoS).* QoS is defined as service quantification that is provided by the network to the demanding core. Thus it involves two aspects: (i) defining the services represented by a certain quantification and (ii) negotiating the services. The services could be low latency, high through-put, low power, bounds on jitter, etc. Negotiating implies balancing the service demands of the core with the services available from the network.

In [Jantsch and Tenhunen 2003](pgs: 61-82), Goossens et al characterize the nature of QoS in relation to NoC. They identify two basic QoS classes, best-effort services (BE) which offer no commitment, and guaranteed services (GS) which do. They also present different levels of commitment, and discuss their effect on predictability of the communication behavior: 1) *correctness* of the result, 2) *completion* of the transaction, 3) *bounds* on the performance. In [Rijpkema et al. 2001] argumentation for the necessity of a combination of BE and GS in NoC is provided. Basically, GS incur predictability, a quality which is often desirable e.g. in real-time systems, while BE improves the average resource utilization [Jantsch and Tenhunen 2003](pgs: 61-82)[Goossens et al. 2002][Rijpkema et al. 2003]. More details of the advantages of GS from a design flow and system verification perspective are given in [Goossens et al. 2005], in which a framework for the development of NoC-based SoC, using the ÆTHEREAL NoC, is described.

Strictly speaking, BE refers to communication for which no commitment can be given, whatsoever. In most NoC related works however, BE covers the traffic for which only correctness and completion are guaranteed while GS is traffic for which additional guarantees are given, i.e. on the performance of a transaction. In macro networks, service guarantees are often of a statistical nature. In tightly bound systems such as SoC, hard guarantees are often preferred. GS allows analytical system verification, and hence a true decoupling of sub-systems. In order to give hard guarantees, GS communication must be logically independent of other traffic in the system. This requires connection-oriented routing. Connections are instantiated as virtual circuits which use logically independent resources, thus avoiding contention. The virtual circuits can be implemented by either virtual channels, time-slots, parallel switch fabric, etc. As the complexity of the system increases and as GS requirements grow, so does the number of virtual circuits and resources (buffers, arbitration logic, etc) needed to sustain them.

While hard service guarantees provide an ultimate level of predictability, soft (statistical) GS or GS/BE hybrids have also been the focus of some research. In [Bolotin et al. 2004], [Felicijan and Furber 2004], [Beigne et al. 2005] and [Rostislav et al. 2005] NoCs providing prioritized BE traffic classes are presented. SoCBUS [Sathe et al. 2003] provides hard, short-lived GS connections, however since these are setup using BE routed packets, and torn down once used, this can also be categorized as soft GS.

ÆTHEREAL [Goossens et al. 2005], NOSTRUM [Millberg et al. 2004], MANGO [Bjerregaard and Sparsø 2005a], SONICS [Weber et al. 2005], aSOC [Liang et al. 2004], and also the NoCs presented in [Liu et al. 2004], in [Leroy et al. 2005], and the static NoC used in the RAW multiprocessor architecture [Taylor et al. 2002], are examples of NoCs implementing hard GS. While most NoCs that implement hard GS use variants of time division multiplexing (TDM) to implement connection-oriented packet routing, thus guaranteeing bandwidth on connections, the clockless NoC MANGO uses sequences of virtual channels to establish virtual end-to-end connections. Hence limitations of TDM, such as bandwidth and latency guarantees being inversely proportional, can be overcome by appropriate scheduling. In [Bjerregaard and Sparsø 2005b] a scheme for guaranteeing latency, independently of bandwidth, is presented. In [Leroy et al. 2005] an approach for allocating individual wires on the link for different connections, is proposed. The authors call this *spatial division multiplexing*, as opposed to TDM.

For readers interested in exploitation of GS (in terms of throughput) virtual circuits during idle times, in [Andreasson and Kumar 2005] and [Andreasson and Kumar 2004] the concept of slack-time aware routing is introduced. A producer manages injection of BE packets during the slacks in time-slots reserved for GS packets, thereby mixing GS and BE traffic at the source which is unlike in other scheme discussed so far where it is done in the routers. In [Andreasson and Kumar 2005] the impact of variation of output buffer on BE latency is investigated, while in [Andreasson and Kumar 2004] the change of injection control mechanism for fixed buffer size is documented. QoS can also be handled by controlling the injection of packets into a BE network. In [Tortosa and Nurmi 2004] scheduling schemes for packet injection in a NoC with a ring topology were investigated. While a basic scheduling, which always farvors traffic already in the ring, provided the highest total bandwidth, weighted scheduling schemes were much more fair in their serving of different cores in the system.

In addition to the above, QoS may also cover special services such as:

—**Broadcast, multicast, narrowcast:** These features allow simultaneous communication from one source to all, i.e. broadcast, or select destinations, as is shown in ÆTHEREAL [Jantsch and Tenhunen 2003](pgs: 61-82), where a master can perform read or write operations on an address-space distributed among many slaves. In a connection oriented environment, the master request is channeled to a single slave for execution in narrowcast, while the master request is replicated for execution at all slaves in multicast. APIs are available within the NA to realize these types of transactions [Radulescu et al. 2004]. An alternate mulitcast implementation is discussed in [Millberg et al. 2004], where a virtual circuit meanders through all the destinations.

—**Virtual wires:** This refers to the use of network message-passing services to emulate direct pin-to-pin connection. In [Bjerregaard et al. 2005] such techniques are used to support a flexible interrupt scheme, in which the interrupt of a slave core can be programmed to trigger any master attached to the network, by sending a trigger packet.

—**Complex operations:** Complex functionality such as test-and-set, issued by a single command across the network, can be used to provide support for e.g. semaphores.

3.3 Link Level

Link level research regards the node-to-node links. These links consist of one or more channels, which can be either virtual or physical. In this section, we present a number of areas of interest for link level research: synchronization, implementation, reliability and encoding.

3.3.1 *Synchronization.* For link level synchronization in a multi clock domain SoC, Chelcea and Nowick have presented a mixed-time FIFO design [Chelcea and Nowick 2001]. The FIFO employs a ring of storage elements in which tokens are used to indicate full or empty state. This simplifies detection of the state of the FIFO (full or empty) and thus makes synchronization robust. In addition, the definitions of full and empty are extended so that full means 0 or 1 cell being unused, while empty means only 0 or 1 cells being used. This helps in hiding the synchronization delay introduced between the state detection and the input/output handshaking. The FIFO design introduced can be made arbitrarily robust, with regards to metastability, as settling time and latency can be traded-off.

With the emerging of the GALS concept of *globally asynchronous locally synchronous* systems [Chapiro 1984][Meincke et al. 1999], implementing links using asynchronous circuit techniques [Sparsø and Furber 2001][Hauck 1995] is an obvious possibility. A major advantage of asynchronous design styles, relevant for NoC, is the fact that apart from leakage, no power is consumed when the links are idle. Thus, the design style addresses also the problematic issue of increasing power usage by large chips. Another advantage is the potentially very low forward latency, in uncongested data paths, leading to direct performance benefits. Examples of NoCs based on asynchronous circuit techniques are CHAIN [Bainbridge and Furber 2002][Amde et al. 2005], MANGO [Bjerregaard and Sparsø 2005a], ANoC [Beigne et al. 2005], and QNoC [Rostislav et al. 2005]. Asynchronous logic incorporates some area and dynamic power overhead compared with synchronous logic, due to local handshake control. The 1-of-4 encodings discussed in Section 3.3.4 below, generalized to 1-of-N, is much used in asynchronous links [Bainbridge and Furber 2001].

On the other hand, resynchronization of an incoming asynchronous transmission is also not trivial. It costs both time and power, and bit errors may be introduced. In [Dobkin et al. 2004], resynchronization techniques are described, and a method for achieving high throughput across an asynchronous to synchronous boundary is proposed. The work is based on the use of stoppable clocks, a scheme in which the clock of a core is stopped while receiving data on an asynchronous input port. Limitations to this technique are discussed, and the proposed method involves only the clock on the input register being controlled. In [Ginosaur 2003] a number of synchronization techniques are reviewed, and the pitfalls of the topic are addressed.

The trade-offs in the choice of synchronization scheme in a globally asynchronous or multiclocked system, is sensitive to the latency requirements of the system, the expected network load during normal usage, the node complexity, etc.

3.3.2 *Implementation issues.* As chip technologies scale into the DSM domain, the effect of wires on link delays and power consumption increase. Aspects and effects on wires of technology scaling are presented in [Ho et al. 2001], [Lee 1998], [Havemann and Hutchby 2001] and [Sylvester and Keutzer 2000]. In [Liu et al. 2004] these issues are covered specifically from a NoC point-of-view, projecting the operating frequency and size of IP cores in NoC-based SoC designs for future CMOS technologies, down to 0.05 μm. In the following, we will discuss a number of physical level issues relevant to the implementation of on-chip links.

Wire segmentation. At the physical level, the challenge lies in designing fast, reliable and low power point-to-point interconnects, ranging across long distances. Since the delay of long on-chip wires is characterized by distributed RC charging, it has been standard procedure for some time to apply segmentation of long wires by inserting *repeater* buffers at regular intervals, in order to keep the delay linearly dependent on the length of the wire. In [Dobbelaere et al. 1995] an alternative type of repeater is proposed. Rather than splitting and inserting a buffer in the path of the wire, it is based on a scheme of sensing and pulling the wire using a keeper device attached to the wire. The method is shown to improve the delay of global wires by up to 2 times compared with conventional repeaters.

Pipelining. Partitioning long interconnects into pipeline stages, as an alternative to wire

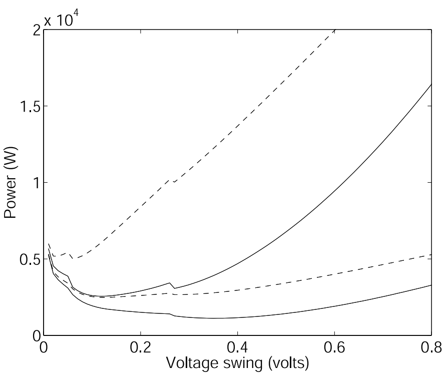


Fig. 14. Total power versus voltage swing for long (5-10mm) on-chip interconnect. Solid line case 1: power supply generated off-chip by high efficiency DC-DC converter. Dashed line case 2: power supply generated internally on-chip. Upper curves for data activity of 0.25, lower curves 0.05 [Svensson 2001] fig 2.

segmentation, is an effective way of increasing throughput. The flow control handshake loop is shorter in a pipelined link, making the critical loop faster. This is at the expense of latency of the link and circuit area, since pipeline stages are more complex than repeater buffers. But the forward latency in an asynchronous pipeline handshake cycle can be minimized to a few gate delays, so as wire effects begin to dominate performance in DSM technologies, the overhead of pipelining as opposed to buffering will dwindle. In [Singh and Nowick 2000] several high-throughput clockless pipeline designs were implemented using dynamic logic. Completion detection was employed at each stage to generate acknowledge signals, which were then used to control the precharging and evaluation of the dynamic nodes. The result was a very high throughput of up to 1.2GDI/s for single rail designs, in a 0.6μm CMOS technology. In [Mizuno et al. 2001] a hybrid of wire segmentation and pipelining was shown, in that a channel was made with segmentation buffers implemented as latches. A congestion signal traveling backwards through the channel compresses the data in the channel, storing it in the latches, until the congestion is resolved. Thus a back pressure flow control scheme was employed, without the cost of full pipeline elements.

Low swing drivers. In an RC charging system, the power consumption is proportional to the voltage shift squared. One way of lowering the power consumption for long on-chip interconnects is by applying low-swing signaling techniques, which are also widely used for off-chip communication lines. Such are presented and analyzed in [Zhang et al. 1999]. Basically the power usage is lowered at the cost of the noise margin. However, a differential transmission line (2 wires), on which the voltage swing is half that of a given single-ended transmission line, has differential mode noise characteristics comparable to the single-ended version. This is so, because the voltage difference between the two wires is the same as that between the single-ended wire and a mid-point between supply and ground. As an approximation, it uses only half the power however, since the two wires working at half the swing each consume one-fourth the power. The common mode noise immunity of the differential version is also greatly improved, and it is thus less sensitive to crosstalk and ground bounces, important sources of noise in on-chip environments as

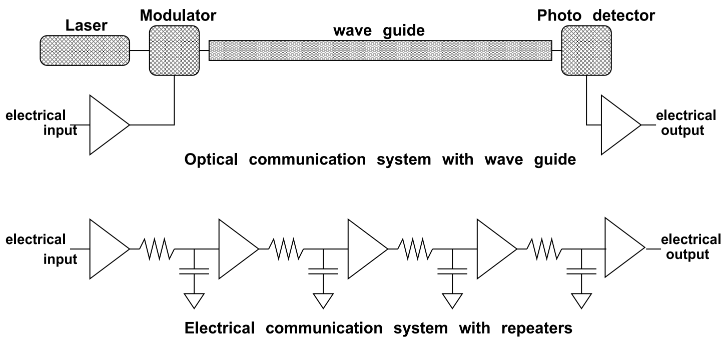


Fig. 15. Model of electrical and optical signaling systems for on-chip communication, showing the basic differences.

discussed in the reliability section below. In [Ho et al. 2003] the design of a low-swing, differential on-chip interconnect for the Smart Memories [Mai et al. 2000] is presented and validated with a test chip.

In [Svensson 2001] Svensson demonstrated how an optimum voltage swing for minimum power consumption in on- and off-chip interconnects can be found for a given data activity rate. The work takes into account dynamic and static power consumption of driving the wire, as well as in the receiver, which needs to amplify the signal back to full logic level. Calculations are presented for a 0.18μm CMOS technology. Figure 14 displays the power consumption versus voltage swing for a global on-chip wire of 5-10mm, a power supply of 1.3V, and a clock frequency of 1GHz. For a data activity rate of 0.25 (random data) it is seen that there is a minimum at 0.12V. This minimum occurs for a two stage receiver amplifier and corresponds to a power saving of 17x. Using a single stage amplifier in the receiver, there is a minimum at 0.26V, corresponding to a power saving of 14x.

Future issues. In [Heiliger et al. 1997] the use of microstrip transmission lines as waveguides for sub-mm wave on-chip interconnects is analyzed. It is shown that using SiO_2 as dielectric exhibit prohibitively large attenuation. However the use of bisbenzocyclobutene-polymer offers favorable line-parameters, with an almost dispersion free behavior at moderate attenuation (=< 1 dB/mm at 100GHz). In [Kapur and Saraswat 2003] a comparison between electrical and optical interconnects for on-chip signaling and clock distribution is presented. Figure 15 shows the models used in evaluating optical and electrical communication. The delay vs. power and delay vs. interconnect length tradeoffs are analyzed for the two types of signaling. As seen in Figure 16, it is shown that the critical length above which the optical system is faster than the electrical is approximately 3-5 mm, projected for a 50nm CMOS fabrication technology with copper wiring. The work also shows that for long interconnects (defined as 10mm and above) the optical communication has a great potential for low power operation. Thus it is projected to be of great use in future clock distribution and global signaling.

3.3.3 *Reliability.* Designing global interconnects in DSM technologies, a number of communication reliability issues become relevant. Noise sources which can have an influence on this are mainly crosstalk, power supply noise such as ground bounce, electromagnetic interference (EMI), and intersymbol interference.

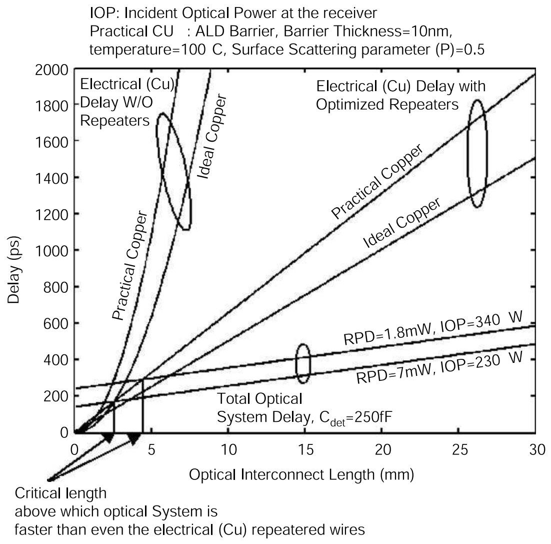


Fig. 16. Delay comparison of optical and electrical interconnect (with and without repeaters) in a projected 50 nm technology [Kapur and Saraswat 2003] fig 13.

Crosstalk is becoming a serious issue due to decreasing supply voltage, increasing wire to wire capacitance, increasing wire inductance (e.g. in power supply lines), and increasing rise times of signaling wires. The wire length at which the peak crosstalk voltage is 10% of the supply voltage, decreases drastically with technology scaling [Jantsch and Tenhunen 2003](chapter 6), and since the length of global interconnects does not scale with technology scaling, this issue is especially relevant to the implementation of NoC links. Power supply noise is worsened by the inductance in the package bonding wires, and the insufficient capacitance in the on-chip power grid. The effect of EMI is worsening as the electric charges moved by each operation in the circuit is getting smaller, making it more susceptible to external influence. Intersymbol interference, i.e. the interference of one symbol on the following symbol on the same wire, is increasing as circuit speeds go up.

In [Jantsch and Tenhunen 2003](chapter 6), the Bertozzi and Benini present and analyze a number of error detecting/correcting encoding schemes in relation to NoC link implementation. Error recovery is a very important issue, since an error in i.e. the header of a packet, may lead to deadlock in the NoC, blocking the operation of the entire chip. This is also recognized in [Zimmer and Jantsch 2003] in which a fault model notation is proposed which can represent multi-wire and multi-cycle faults. This is interesting due to the fact that crosstalk in DSM busses can cause errors across a range of adjacent bits. It is shown that by splitting a wide bus into separate error detection bundles, and interleaving these, the error rate after using single-error correcting and double-error detecting codes can be reduced by several orders of a magnitude. This is because these error-correction schemes function properly when only respectively one or two errors occur in each bundle. When the bundles are interleaved, the probability of multiple errors within the same bundle is greatly reduced.

In [Gaughan et al. 1996] Gaughan et al deal with dynamically occurring errors in networks with faulty links. Their focus is on routing algorithms that can accommodate such errors, assuming that support for the detection of the errors is implemented. For wormhole routing, they present a scheme in which a data transmission is terminated upon detection of an error. A *kill* flit is transmitted backwards, deleting the worm and telling the sender to retransmit it. This naturally presents an overhead, and is not generally representative for excising NoC implementations. It can however prove necessary in mission critical systems. The paper provides formal mathematical proofs of deadlock-freedom.

Another issue with new CMOS technologies is the fact that the delay distribution – due to process variations – flattens with each new generation. While typical delay improves, worst-case delay barely changes. This presents a major problem in todays design methodologies, as these are mostly based on worst-case assumptions. Self-calibrating methods, as used in [Worm et al. 2005], are a way of dealing with unreliability issues of this character. The paper presents a self-calibrating link, and the problem of adaptively controlling its voltage and frequency. The object is to maintain acceptable design trade-offs between power consumption, performance and reliability, when designing on-chip communication systems using DSM technologies.

Redundant transmission of messages in the network is also a way of dealing with fabrication faults. In [Pirretti et al. 2004], two different flooding algorithms and a random walk algorithm are compared. It is shown that the flooding algorithms have an exceedingly large communication overhead, while the random walk offers reduced overhead while maintaining useful levels of fault tolerance.

With the aim of improving fabrication yield, Dally and Towles propose extra wires between nodes, so that defective wires found during post production tests or during self-test at start-up can be bypassed [Dally and Towles 2001]. Another potential advantage of a distributed shared communication structure is the possibility of bypassing entire regions of a chip, if fabrication faults are found.

Dynamic errors are more likely in long wires, and segmenting links into pipeline stages helps keeping the error rate down and the transmission speed up. Since segmentation of the communication infrastructure is one of the core concepts of NoC, it inherently provides solutions to the reliability problems. The segmentation is made possible because NoC-based systems generally imply the use of programming models allowing some degree of latency insensitive communication. Thus it is seen how the issues and solutions at the physical level relate directly to issues and solutions at system level, and vice versa. Another solution towards avoiding dynamic errors is the shielding of signal wires, e.g. by ground wires. This helps to minimize crosstalk from locally interfering wires, at the expense of wiring area.

3.3.4 *Encoding*. Using encoding for on-chip communication has been proposed, the most common objective being to reduce power usage per communicated bit, while maintaining high speed and good noise margin. In [Bogliolo 2001] the proposed encoding techniques are categorized as speed-enhancing or low-power encodings, and it is shown how different schemes in these two categories can be combined to gain the benefits of both. In [Nakamura and Horowitz 1996] a very simple low-weight coding technique was used to reduce dI/dt noise due to simultaneous switching of off-chip I/O drivers. An 8-bit signal was simply converted to a 9-bit signal, the 9th bit indicating whether the other 8 bits should be inverted. The density of 1's was thus reduced, resulting in a reduction of switching noise by 50% and of power consumption by 18%. Similar techniques could prove useful in relation to long on-chip interconnects. The abundant wire resources available on-chip can also be used to implement more complex M-of-N encodings, thus trading wires for power. A widely used technique, especially in asynchronous implementations, is 1-of-4 encoding. This results in a good power/area tradeoff and low encoding/decoding overhead [Bainbridge and Furber 2001][Bainbridge and Furber 2002].

Another area of encoding, also discussed in Section 3.3.3, relates to error management. This involves the detection and correction of errors that may occur in the network. The mechanism may be observed at different layers of the network, and thus be applicable to either phits, flits, packets or messages. With regards to NoC, the interesting issues involve errors in the links connecting the nodes, since long wires of deep submicron technologies may exhibit unreliable behavior (see Section 3.3.3). ×pipes [Osso et al. 2003] implements flit-level CRC mechanism running in parallel with switching (thus masking its delay) to detect errors. Another common technique is parity-checks. The need here is to balance complexity of error-correction circuits to the urgency of such mechanism.

An interesting result is obtained in [Jantsch and Vitkowski 2005], wherein the authors investigate power consumption in the NOSTRUM NoC. Results are based on a 0.18μm implementation, and scaled down to 65 nm. The paper concludes that the major part of the power is spent in the link wires. Power saving encoding however results in reduced performance, and simply scaling the supply voltage to normalize performance – in non-encoded links – actually results in better power figures than any of the encoding schemes investigated. Subsequently, the authors propose the use of end-to-end data protection, through error correction methods, which allows voltage scaling while maintaining the fault probability without lowering the link speed. In effect this results in better power figures. These claims are supported by simulations showing promising results.

In this section we have discussed issues relevant to the lowest level of the NoC, the link level. This concludes the discussion of network design and implementation topics. In the following section we discuss NoC from the view of design approach, modeling, and in relation to SoC.

4. NOC MODELING

NoC, described as a subset of SoC, is an integral part of SoC design methodology and architecture. Given the vast design space and implementation decisions involved in NoC design, modeling and simulation is important to design flow, integration and verification of NoC concepts. In this section, first we discuss issues related to NoC modeling and then we explore design methodology used to study the system-level impact of the NoC. Finally traffic characterization, which bridges system-level dynamics with NoC requirements is discussed.

4.1 Modeling

Modeling the NoC in abstract software models is the first means to approach and understand the required NoC architecture and the traffic within it. Conceptually the purpose of NoC modeling is (i) to explore the vast design and feature space, and (ii) to evaluate trade-offs between power, area, design-time, etc; while adhering to application requirement on one side and technology constraints on the other side. Modeling NoC has three intertwined aspects: modeling environment, abstraction levels, and result analysis. In the modeling environment section, we present three frameworks to describe NoC. Section 4.1.2 discusses work done across different levels of NoC abstraction. The result analysis deals with a wide range of issues and is hence dealt with separately in Section 5.

4.1.1 *Modeling Environment.* The NoC models are either analytical or simulation based and can model communication across abstractions.

In a purely abstract framework, a NoC model using allocators, scheduler, and synchronizer is presented in [Madsen et al. 2003] and [Mahadevan et al. 2005]. The allocator translates the path traversal requirements of the message in terms of its resource requirements such as bandwidth, buffers, etc. It attempts to minimize resource conflicts. The scheduler executes the message transfer according to the particular network service requirements. It attempts to minimize resource occupancy. A synchronizer models the dependencies among communicating messages allowing concurrency. Thus these three components are well suited to describe a wide variety of NoC architecture and can be simulated in a multi-processor real-time environment.

OPNET, a commercial network simulator originally developed for macro-networks, is used as NoC simulator in [Bolotin et al. 2004][Xu et al. 2004][Xu et al. 2005]. OPNET provides a convenient tool for hierarchical modeling of a network, including processes (state machines), network topology description and simulation of different traffic scenarios. However, as noted in [Xu et al. 2004] and [Xu et al. 2005], it needs to be adapted for synchronous environments, requiring explicit design of clocking scheme and a distribution network. [Bolotin et al. 2004] uses OPNET to model a QoS-based NoC architecture and design with irregular network topology.

A VHDL based cycle accurate RTL model for evaluating power and performance of NoC architecture is presented in [Banerjee et al. 2004]. The power and delay are evaluated for fine-grain components of the routers and links using SPICE simulations for a 0.18μm technology and incorporated into the architectural-level blocks. Such modeling enables easy evaluation of dynamic vs leakage power at system-level. As expected, at high injection rate (packets/cycle/node) it was found that dynamic power dominates over leakage power. The Orion power-performance simulator proposed by Wang et al. [Wang et al. 2002], modeled only dynamic power consumption.

Recently, due to increasing size of applications, NoC emulation [Genko et al. 2005] has been proposed as alternative to simulation-based NoC models. It has been shown that FPGA based emulation can take few seconds compared to simulation-based approaches which can take hours to process through many millions of cycles, as would be necessary in any thorough communication co-exploration.

4.1.2 *Noc Modeling at Different Abstraction Levels.* New hardware description languages are emerging, such as SystemC [SystemC 2002], a library of C++, and SystemVerilog [Fitzpatrick 2004], which make simulations at a broad range of abstraction levels readily available, and thus support the full range of abstractions needed in a modular NoC-based SoC design. In [Bjerregaard et al. 2004] mixed-mode asynchronous handshake channels were developed in SystemC, and a mixed abstraction level design flow was used to design two different NoC topologies.

From an architectural point of view, the network topology generally incur the use of a segmented (multi-hop) communication structure, however some researchers working at the highest levels of abstraction, define NoC merely as a multiport blackbox communication structure or core, presenting a number of ports for communication. A message can

Table III. Communication semantics and abstraction for NoC.

Layer	Interface semantics	Communication

Application/ Presentation	IP-to-IP messaging
	`sys.send(struct myData)`
	`sys.receive(struct myData)`
		Message passing

Session/ Transport	IP-to-IP port-oriented messaging
	`nwk.read(messagepointer*, unsigned len)`
	`nwk.write(int addr, msgptr*, unsigned len)`
		Message passing or shared memory

Network	NA-to-NA packet streams
	`ctrl.send(), ctrl.receive()`
	`link.read(bit[] path, bit[] data_packet)`
	`link.write(bit[] path, bit[] data_packet)`
		Message passing or shared memory

Link	Node-to-Node logical links and shared byte streams
	`ctrl.send(), ctrl.receive()`
	`channel.transmit(bit[] link, bit[] data_flit)`
	`channel.receive(bit[] link, bit[] data_flit)`
		Message passing

Physical	Pins and wires `A.drive(0), D.sample(), clk.tick()`	Interconnect

be transmitted from an arbitrary port to any other, allowing maximum flexibility of system communication. At this level, the actual implementation of the NoC is often not considered. Working at this high abstraction level allows a great degree of freedom from lower level issues. Table III adapted from Gerstlauer [Gerstlauer 2003] summarizes, in general, the communication primitives at different levels of abstraction.

At system level, transaction level models (TLM) are typically used for modeling communication behavior. This takes the form of either synchronous or asynchronous *send()/ receive()* message passing semantics, which use unique channels for communication between the source and the destination. One level below this abstraction, for NoCs, additional identifiers such as addressing may be needed to uniquely identify the traversal path or for providing services for end-to-end communication. Control primitives at network and link level, which are representative of actual hardware implementation, model the NoC flow-control mechanisms. In [Gerstlauer 2003], a JPEG encoder and voice encoder/decoder running concurrently were modeled for each and for mixed-levels of abstraction. As expected the results show that the model complexity generally grows exponentially with lower level of abstraction. By extrapolating the result from bus to NoC, interestingly, model complexity at NA level can be found to be higher than at other levels due to slicing of message, connection management, buffer management, and others.

Working between session to network layer, Juurlink and Wijshoff [Juurlink and Wijshoff 1998] have made a comparison of three communication models used in parallel computation; (i) asynchronous communication with fixed message size, (ii) synchronous communication which rewards burst-mode message transfers, and (iii) asynchronous with variable message size communication while also accounting for network load. Cost-benefit analysis shows that, though the software-based messaging layers serve as a very useful function of delinking computation and communication, it creates anywhere between 25% to 200% overhead, as opposed to optimized hardware implementation.

A similar study of parallel computation applications, but with more detailed network model, was undertaken by Vaidya et al. [Vaidya et al. 2001]. Here the network was implemented to use adaptive routing with virtual channels. The applications, running on power-of-two number of processors using grid based network topologies, used shared-memory or message passing for communication thus generating wide range of traffic patterns. They have found that increasing the number of VCs and routing adaptively offer little performance improvement for scalable shared-memory applications. Their observation holds true over a range of systems and problem sizes. The results show that the single most important factor for improving performance in such applications is the router speed, which is likely to provide lasting payoffs. The benefits of a faster router are visible across all applications in a consistent and predictable fashion.

Ahonen et al. [Ahonen et al. 2004] and Lahiri et al. [Lahiri et al. 2001] have associated high level modeling aspects with actual design choices such as: selection of an appropriate topology, selection of communication protocols, specification of architectural parameters (such as bus widths, burst transfer size, priorities, etc), and mapping communications onto the architecture, as requirements to optimize the on-chip communication for application-specific needs. Using a tool called OIDIPUS, Ahonen et al. compare (IP) block placement of twelve processors restricted to a ring-based topology. It is found that OIDIPUS, which uses physical path taken by the communication channels as cost function, generated topologies are only marginally inferior to human design. Without being restricting to any one topology, Lahiri et al. have evaluated traffic characteristics in a static priority based shared bus, hierarchical bus, two-level time division multiplexed access (TDMA), and ring based communication architecture. It was found that no single architecture uniformly outperforms other.

Wieferink [Wieferink et al. 2004] have demonstrated a processor/communication co-exploration methodology which works cross-abstraction and in a co-simulation platform. Here LISA based IP core descriptions have been integrated with SystemC based bus based transaction level models. A wide range of APIs are then provided to allow modeling between LISA and SystemC models, to allow instruction accurate model to co-exist with cycle accurate model, and TLM with RTL models. MPARM [Loghi et al. 2004] is a similar cycle-accurate and SystemC co-exploration platform, used in exploration of AMBA, STBus and ×pipes NoC evaluation.

4.2 Design and Co-Exploration Methodology

The NoC components, as described in Section 2.1, lends itself to flexible NoC designs such as parameterizable singular IP core, or malleable building-blocks, customizable at the network-layer, for design and reuse into application-specific NoC. A SoC design methodology requiring a communication infrastructure, can exploit either characteristics to suit the application's needs. Keeping this in mind, different NoC researchers have uniquely tailored their NoC architectures. Figure 17 shows our assessment of instance-specific capability of these NoC architectures. The two axis are explained as follows.

—**Parametrizability at system-level:** By this, we mean the ease with which a system-level NoC characteristic can be changed at instantiation time. The NoC description may encompass a wide range of parameters, such as: number of slots in the switch, pipeline-stages in the links, number of ports of the network and others. This is very useful for co-exploration directly with IP cores of the SoC.

—**Granularity of NoC:** By granularity, we mean at what level the NoC or NoC components is described. At the coarser end, the NoC may described as a single core, while at

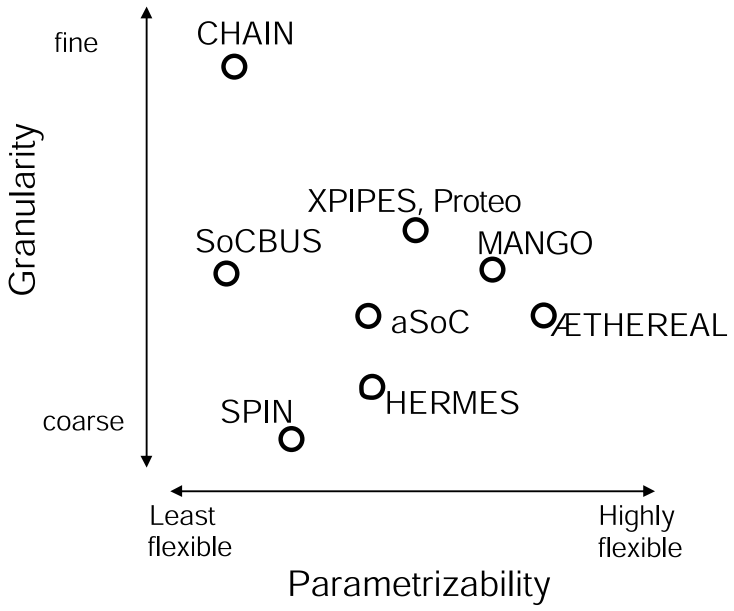


Fig. 17. NoC Instantiation Space

other end of the spectrum, the NoC may be assembled from lower-level blocks.

Consider the example of CHAIN [Bainbridge and Furber 2002]. It provides a library of fine-grained NoC components. Using these components, a NoC designer can use Lego-brick approach to build the desired NoC topology, though as system-level block such a NoC has low flexibility. Thus it may be disadvantageous, when trying to find the optimum SoC communication architecture in a recursive design space exploration process. The ÆTHEREAL [Goossens et al. 2002], SoCBUS [Sathe et al. 2003], and aSoC [Liang et al. 2000] networks describe the NoC as a relatively coarse grain system-level module but with widely different characteristics. The ÆTHEREAL is highly flexible in terms of adjusting the available slots, number of ports, etc which is useful for NoC exploration; where as aSoC and SoCBUS do not expose many parameters for change (though aSoC supports flexible programming of connections after instantiation). The SPIN NoC [Guerrier and Greiner 2000], designed as a single IP core, is least parameterizable with its fixed topology and protocol. Interestingly, the xpipes [Osso et al. 2003] provides not merely a set of relatively fine-grain soft-macros of switches and pipelined links, which the xpipesCompiler [Jalabert et al. 2004] uses to automatically instantiate an application specific network, but also enables optimization of system-level parameters such as removing redundant buffers from output ports of switches, sharing signals common to objects, etc. This lends itself to both high flexibility for co-exploration and easy architectural changes when needed. Similarly, conclusions can be drawn of Proteo [Siguenza-Tortosa et al. 2004], HERMES [Moraes et al. 2004] and MANGO [Bjerregaard and Sparsø 2005a] NoCs. A detailed comparison of different features of most of the above listed NoCs is tabulated in [Moraes et al. 2004].

The impact on SoC design time and co-exploration, of different NoC design styles listed above is considerable. For example in [Jalabert et al. 2004], during design space exploration, to find an optimum NoC for three video applications: video object plane decoder, MPEG4 decoder and multi-window displayer; the xpipesCompiler found that irregular networks with large switches may be more advantageous than regular networks. This is easier to realize in macro-block NoC such as CHAIN or xpipes, than it is in NoC designed as a single (system-level) IP core such as SPIN. The basis for the compiler's decision is the pattern of traffic generated by the application. This is the focus of the next section. For further understanding of trade-offs in using a flexible instantiation-specific NoC can be found in [Pestana et al. 2004], where different NoC topologies and each topology with different router and NA configuration is explored.

4.3 Traffic Characterization

The communication types expected in a NoC range across virtual wires, memory access, audio/video stream, interrupts, and others. Many combinations of topology, protocol, packet sizes and flow control mechanisms exist for the efficient communication of one or more predominant traffic patterns. For example, in [Kumar et al. 2002] packet-switched NoC concepts have been applied to a 2D mesh network topology, whereas in [Guerrier and Greiner 2000] such concepts have been applied to a butterfly fat-tree topology. The design decisions were based on the traffic expected in the respective systems. Characterizing the expected traffic is an important first step towards making sound design decisions.

A NoC must accommodate different types of communication. We have realized that regardless of the system composition, clustering, topology and protocol, the traffic within a system will fall in one of three categories:

(1) **Latency-critical:** Latency-critical traffic is traffic with stringent latency demands such as for critical interrupts, memory access, etc. These often have low payload,

(2) **Data-streams:** Data streaming traffic have high payload and demand QoS in terms of bandwidth. Most often it is large, mostly fixed bandwidth which may be jitter critical. Examples are MPEG data, DMA access, etc.

(3) **Best-effort:** The best-effort traffic, as explained in Section 3.2.4, is traffic with no specific requirements of commitment from the network.

The categorization above is a guideline, rather than a hard specification and is presented as a superset of possible traffic types. Bolotin et al. [Bolotin et al. 2004] provide a more refined traffic categorization, combining the transactions at the network boundary with service requirements, namely: signaling, real-time, read/write (RD/WR) and block-transfer. In relation to the above categorization; signaling is latency-critical, real-time is data-streaming, and RD/WR and block-transfer are both best-effort with distinguishing factor being the message size. Though one or more of the traffic patterns may be predominant in the SoC, it is important to understand that a realistic NoC design should be optimized for a mix of above traffic patterns. The conclusions of a case-study of NoC routing mechanism for three traffic conditions with fixed number of flits per packet as presented in [Ost et al. 2005], can thus be enriched by using non-uniform packet size and relating them to the above traffic categories.

It is important to understand the bandwidth requirements of the listed traffic types for a given application, and accordingly map the IP cores on the choosen NoC topology. Such a study is done in [Murali and Micheli 2004a]. NMAP (now called SUNMAP [Murali and Micheli 2004b]), a fast mapping algorithm that minimizes the average communication delay with minimal-path and split-traffic routing in 2D mesh, is compared with greedy and partial branch-and-bound algorithms. It is shown to produce results of higher merit (reduced packet latency) for DSP benchmarks. Another dimension in the mapping task is that of allocating guaranteed communication resources. In [Goossens et al. 2005] and [Hansson et al. 2005], approaches to this task are explored for the ÆTHEREAL NoC.

Specific to data-stream type traffic described above, Rixner et al. [Rixner et al. 1998] have identified unique qualities relating to the inter-dependencies between the media streams and frequency of such streams in the system. It is called the streaming programming model. The basic premises of such programming is static analysis of the application to optimize the mapping effort, based on prior knowledge of the traffic pattern, so as to minimize communication. The communication architecture tuner (CAT) proposed by Lahiri et al. [Lahiri et al. 2000] is a hardware-based approach that does runtime analysis of traffic and manipulates the underlying NoC protocol. It does this by monitoring the internal state and communication transactions of each core, and then predicts the relative importance of each communication event in terms of its potential impact on different system-level performance metrics such as number of deadline misses, average processing time, etc.

The various blocks of NoC can be tuned for optimum performance with regard to a specific traffic characteristic, or the aim can be more general, towards a one-fits-all network, for greater flexibility and versatility.

5. NETWORK ANALYSIS

The most interesting and universally applicable parameters of NoC are *latency*, *bandwidth*, *jitter*, *power consumption* and *area usage*. Latency, bandwidth and jitter can be classified as performance parameters, while power consumption and area usage are the cost factors. In this section we will discuss the analysis and presentation of results in relation to these parameters.

5.1 Performance Parameters and Benchmarks

Specifying a single of the performance parameters introduced above is not sufficient to confer a properly constrained NoC behavior. At least two must be defined. The following example illustrates this:

Given a network during normal operation, it is assumed that the network is not overloaded. For such a network, all data is guaranteed to reach its destination, when employing a routing scheme in which no data is dropped (see Section 3.2.2, delay routing model). This means that as long as the capacity of the network is not exceeded, any transmission is guaranteed to succeed (any required bandwidth is guaranteed). However, nothing is stated concerning the transmission latency, which may well be very high in a network operated near full capacity. As seen in Figure 18, the exact meaning of which will be explained later, the latency of packets rise in an exponential manner, as the *network load* increases. The exact nature of the network load will be detailed later in this section. It is obvious that such guarantees are not practically usable. We observe that the bandwidth specification is worthless with out a bound on the latency as well. This might also be presented in terms of a maximum time window, within which the specified bandwidth would always be reached, i.e. the jitter of the data stream (the *spread* of the latencies). Jitter is often a more interesting parameter in relation to bandwidth, than latency, as it describes the temporal evenness of the data stream.

Likewise, a guaranteed bound on latency might be irrelevant, if the amount of data that can be transmitted at this latency is extremely small. Thus latency, bandwidth and jitter are closely related. Strictly speaking, one should not be specified without at least one of the others.

At a higher abstraction level, performance parameters used in evaluating multicomputer networks in general have been adopted by NoC researchers. These include *aggregated bandwidth*, *bisection bandwidth*, *link utilization*, *network load*, etc. The aggregate bandwidth is the accumulated bandwidth of all links, and the bisection bandwidth is the minimum collective bandwidth across links that when cut, separate the network into two equal set of nodes. Link utilization is the load on the link, compared with the total bandwidth available. The network load can be measured as a fraction of the *network capacity*, as *normalized bandwidth*. The network capacity is the maximum capacity of the network for a uniform traffic distribution, assuming that the most heavily loaded links are located in the network bisection. These and other aspects of network performance metrics are discussed in detail in Chapter 9 of [Duato et al. 2003].

For highly complex systems, such as full-fledged computer systems including processor(s), memory and peripherals, the individual parameters may say little about the overall functionality and performance of the system. In such cases, it is customary to make use of benchmarks. NoC-based systems represents such complexity, and benchmarks would be natural to use in its evaluating. Presenting performance in the form of benchmark results would help clarify the effect of implemented features, in terms of both performance benefits (latency, jitter and bandwidth) and implementation and operation costs (area usage and power consumption). Benchmarks would thus provide a uniform plane of reference from which to evaluate different NoC architectures. At present, no benchmark system exists explicitly for NoC, but its development is an exciting prospect. In [Vaidya et al. 2001] examples from the NAS benchmarks [Bailey et al. 1994] were used, in particular Class-A NAS-2. This is a set of benchmarks that has been developed for the performance evaluation of highly parallel supercomputers, which mimic the computation and data movement characteristics of large scale computational fluid dynamics applications. It is questionable however, how such parallel computer benchmarks can find use in NoC, as the applications in SoCs are very different. In particular, SoC applications are generally highly heterogeneous, and the traffic patterns therein likewise. Another set of benchmarks, used as basis of NoC evaluation in [Hu and Marculescu 2004a], are the embedded system synthesis benchmark suites [Dick].

5.2 Presenting Results

Generally it is necessary to simplify the multidimensional performance space. One common approach is to adjust a single aspect of the design, while tracking the effect on the performance parameters. An example is tracking the latency of packets, while adjusting the bandwidth capabilities of a certain link within the network, or the amount of background traffic generated by the test environment. In Section 5.2.1 we will give specific examples of simple yet informative ways of communicating results of NoC performance measurements.

Since the NoC is a shared, segmented communication structure, wherein many individual data transfer sessions can take place in parallel, the performance measurements depend not only on the traffic being measured upon, but also on the other traffic in the network, the *background traffic*. The degree of background traffic is often being indicated by the network load, as described above. Though very simple, this definition makes valuable sense in considering a homogeneous, uniformly loaded network. One generally applicable practical method for performance evaluation is thus generating a uniform randomly distributed background traffic so that the network load reaches a specified point. Test packets can then be sent from one node to another, according to the situation that one desires to investigate, and the latencies of these packets can be recorded (see example (i) in Section 5.2.1 below).

Evenly distributed traffic however, may cloud important issues of the network performance. In [Dally and Aoki 1993] the degree of symmetry of the traffic distribution in the network was used to illustrate aspects of different types of routing protocols; adaptive and deterministic. The adaptive protocol resulted in a significant improvement of throughput over the deterministic one, for non-uniform traffic, but had little effect on performance with uniformly distributed traffic. The reason for this is that the effect of adaptive protocols is to even out the load, to avoid *hotspots*, thus making better use of the available network resources. If the bulk load is already evenly distributed, there is no advantage. Also traffic parameters like number of packets and packet size, can have a great influence on performance, e.g. in relation to queueing strategies in nodes.

There are many ways to approach the task of presenting test results. The performance space is a complex, multidimensional one, and there are many pitfalls to be avoided, in order to display intelligible and valuable information about the performance of a network. Often the presented results fail to show the interesting aspects of the network. It is easy to get lost in the multitude of possible combinations of test parameters. This may lead to clouding, or at worst failure to properly communicate, the relevant aspects of the research. Though the basis for performance evaluation may vary greatly, it is important for researchers to be clear about the evaluation conditions, allowing others to readily and intuitively grasp the potential of a newly developed idea, and the value of its usage in NoC.

5.2.1 *Examples.* Below we will give some specific examples that we find clearly communicate the performance of the networks being analyzed. What makes these examples good are their simplicity in providing a clear picture of some very fundamental properties of the involved designs.

(i) Average latency vs. network load. In [Dally and Aoki 1993] this is used to illustrate the effect of different routing schemes. Figure 18 is a sample figure from the article, showing how the average latency of the test data grows exponentially as the background traffic load of the network is increased. In the presented case, the *throughput saturation point*, the point at which the latency curve bends sharply upwards, is shifted right as more complex routing schemes are applied. This corresponds to a better utilization of available routing resources. The article does not address the question of cost factors of the implementation.

(ii) Frequency of occurrence vs. latency of packet. Displaying the average latency of packets in the network may work well for establishing a qualitative notion of network performance. Where more detail is needed, a histogram, or similar graph, showing the distribution of latencies, across the delay spectrum is often used with great effect. This form of presentation is used in [Dally 1992] to illustrate the effect of routing prioritization schemes on the latency distribution. Figure 19, taken from the article, shows the effect of *random scheduling* and *deadline scheduling*. Random scheduling schedules the packets for transmission in a random fashion, while deadline scheduling prioritize packets according to how long they have been waiting (oldest-packet-first). It is seen how the choice

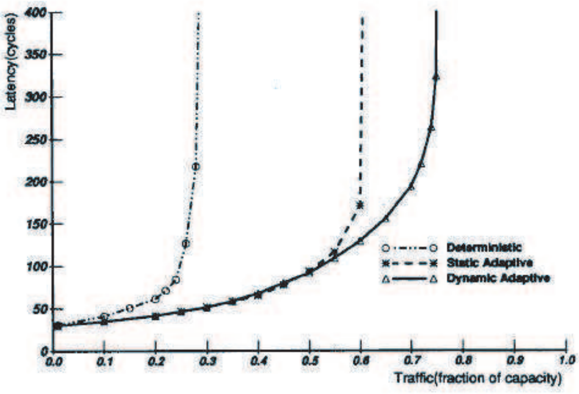


Fig. 18. Latency vs. network load for different routing schemes. The figure shows how the employment of more complex routing schemes move the point at which the network saturates [Dally and Aoki 1993] fig 5.

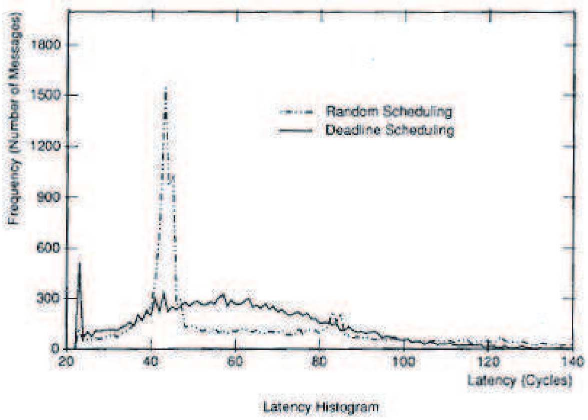


Fig. 19. Number of messages as a function of latency of message (latency distribution), for two scheduling schemes [Dally 1992] fig 17.

of scheduling affect the distribution of latencies of messages. In [Bjerregaard and Sparsø 2005b] such a latency distribution graph is also used, to display how a scheduling scheme provides hard latency bounds, in that the graph is completely empty beyond a certain latency.

(iii) Jitter vs. network load. The jitter of a sequence of packets is important when dimensioning buffers in the network nodes. High jitter (*bursty* traffic) needs large buffers to compensate, in order to avoid congestion resulting in suboptimal utilization of routing resources. This issue is especially relevant in multimedia application systems with large continuous streams of data, such as that presented in [Varatkar and Marculescu 2002]. In this work statistical mathematical methods are used to analyze the traffic distribution. Fig-

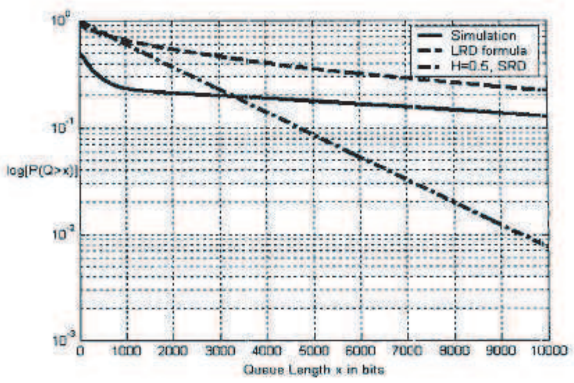


Fig. 20. The probability of queue length exceeding buffer size. The results for two models based on stochastic processes, LRD (Long Range Dependent) and SRD (Short Range Dependent), are plotted along with simulation results for comparison [Varatkar and Marculescu 2002] fig 6.

ure 20, taken from the article, explores the use of two different models based on stochastic processes, for predicting the probability that the queue length needed to avoid congestion exceeds the actual buffer size, in the given situation. The models displayed in the figure are LRD (Long Range Dependent) or *self-similar*, and SRD (Short Range Dependent) or *Markovian* stochastic processes. In the figure, these models are compared with simulation results. The contributions of the paper include showing that LRD processes can be used effectively to model the bursty traffic behavior at chip-level, and the figure shows how indeed the predictions of the LRD model comes closer to the simulation results than those of the SRD model.

5.3 Cost Factors

The cost factors are basically power consumption and area usage. A comparative analysis of cost of NoC is difficult to make. As is the case for performance evaluation, no common ground for comparison exists. This would require different NoC being demonstrated for the same application, which is most often not the case. Hence a somewhat broad discussion of cost in terms of area and power cost is presented in this section.

The power consumption of the communication structure in large single-chip systems is a major concern, especially for mobile applications. As discussed earlier, the power used for global communication does not scale with technology scaling, leading to increased power use by communication relative to power use by processing. In calculating the power consumption of a system, there are two main terms: (i) power per communicated bit and (ii) idle power. Depending on the traffic characteristics in the network, different implementation styles will be more beneficial with regards to power usage. In [Nielsen and Sparsø 2001] a power analysis of different low-power implementations of on-chip communication structures was made. The effects on power consumption of scaling a design were seen and a bus design was compared with torus connected grid design (both synchronous and asynchronous implementations). Asynchronous implementation styles (discussed in Section 3.3.1), are beneficial for low network usage, since they have very limited power consumption when idle, but use more power per communicated bit, due to local control overhead. Technology scaling however leads to increased leakage current, resulting in an increasing static power use in transistors. Thus the benefit of low idle power in asynchronous circuits may dwindle.

From a system-level perspective, knowledge of network traffic can be used to control the power use of the cores. Interest has been in investigating centralized versus distributed power management schemes. Centralized power managers (PM) are a legacy in bus-based systems. Since NoC is most often characterized by distributed routing control, naturally distributed PMs such as those proposed in [Benini and Micheli 2001] and [Simunic and Boyd 2002], would be useful. In both of these studies, conceptually there is a node-centric and network-centric PM. The node-centric PM controls the powering up or down of the core. The network-centric PM is used to for overall load-balancing and to provide some estimations to the node-centric PM of incoming requests, thus masking the core's wake-up cost by precognition of traffic. This type of power management is expected to be favored to reduce power consumption in future NoCs. The results, presented in [Simunic and Boyd 2002] show that with only node PM, the power saving range from factor of 1.5 to 3 compare to no power managers. Combining dynamic voltage scaling with DPM gives overall saving of factor of 3.6. The combined implementation of node and network centric management approaches shows energy savings of a factor of 4.1 with performance penalty reduced by minimum 15% compared to node-only PM. Unlike these dynamic runtime energy monitors, in [Hu and Marculescu 2004b] a system-level energy-aware mapping and scheduling (EAS) algorithm is proposed, which statically schedules both communication transactions and computation tasks. For experiments done on 2D mesh with minimal-path routing, energy savings of 44% are reported, when executing complex multimedia benchmarks.

A design constraint of NoC less applicable to traditional multicomputer networks, lies in the area usage. A NoC is generally required to take up less than 5% of the total chip area. For a 0.13μm SoC with one network node per core, and an average core size of 2x2mm (app. 100 cores on a large chip), this corresponds to 0.2mm^2 per node. One must also remember that the NA will use some area, depending of the complexity of the features that it provides. Trade-off decisions which are applicable to chip design in general and not particular to NoC are beyond the scope of this survey. At the network level, many researchers have concluded that buffering accounts for the major portion of the node area, hence wormhole routing has been a very popular choice in NoCs, see Section 3.2.2. As examples of an area issue related to global wires can be mentioned that introducing *fat wires*, i.e. the usage of wide and tall top level metal wires for global routing, the power figures may improve, at the expense of area [Sylvester and Keutzer 2000].

6. NOC EXAMPLES

In this section we briefly recapitulate on a handful of specific NoC examples, describing the design choices of actual implementations, and accompanying work by the research groups behind. This is by no means to be seen as a complete compilation of existing NoCs, there are many more, rather the purpose of this section is to address a representative set: ÆTHEREAL, NOSTRUM, SPIN, CHAIN, MANGO, and ×PIPES. In [Moraes et al. 2004] a list in tabular form is provided, which effectively characterizes many of the NoCs not covered in the following.

i **ÆTHEREAL:** The ÆTHEREAL, developed at Philips, is a NoC that provides guaranteed throughput (GT) along side best-effort (BE) service [Rijpkema et al. 2001][Goossens et al. 2002][Wielage and Goossens 2002][Dielissen et al. 2003][Jantsch and Tenhunen 2003](pgs: 61-82)[Rijpkema et al. 2003][Radulescu et al. 2004][Goossens et al. 2005]. In the ÆTHEREAL the guaranteed services pervade as a requirement for hardware design and also as a foundation for software programming. Here, the router's primary function is to provide both GT and BE services. All routers in the network have a common sense of time, and the routers forward traffic based on slot allocation. Thus a sequence of slots implement a virtual circuit. GT traffic is connection-oriented, and did in early router instantiations not have headers, as the next hop was determined by a local slot table. In recent versions the slot tables have been removed to save area, and the information is provided in a GT packet header. The allocation of slots can be setup statically, during an initialization phase, or dynamically during runtime. BE traffic makes use of non-reserved slots and of any slots reserved but not used. BE packets are used to program the GT slots of the routers. With regards to buffering, input queuing is implemented using custom-made hardware fifos, to keep the area costs down. The ÆTHEREAL connections support a number of different transaction types, such as read, write, acknowledged write, test and set, and flush, and as such it is similar to existing bus protocols. In addition, it offers a number of connection types: narrowcast, multicast, and simple.

In [Dielissen et al. 2003] an ÆTHEREAL router with 6 bidirectional ports of 32 bits was synthesized in 0.13 μm CMOS technology. The router had custom made BE input queues depth of 24 words per port. The total area was 0.175 mm^2, and the bandwidth was 500 MHz x 32 bits = 16 Gbit/s per port. A network adapter with 4 standard socket interfaces (master and slave – OCP, DTL or AXI based) was also reported with an area of 0.172 mm^2 implemented in the same technology.

In [Goossens et al. 2005] and [Pestana et al. 2004] an automated design flow for instantiation of application specific ÆTHEREAL is described. The flow uses XML to input various parameters such as traffic characteristics, GT and BE requirements, and topology. A case study of MPEG codec SoC is used to validate and verify the optimizations undertaken during the automated flow.

ii **NOSTRUM:** The work of researchers at KTH in Stockholm has evolved from a system-level chip design approach [Kumar et al. 2002][Jantsch and Tenhunen 2003][Zimmer and Jantsch 2003][Millberg et al. 2004]. Their emphasis has been on architecture and platform-based design, targeted towards multiple application domains. They have recognized the increasing complexity of working with high density VLSI technologies and hence highlighted advantages of a grid-based, router-driven communication media for on-chip communication.

Also the implementation of guaranteed services has also been a focus point of this group. In the NOSTRUM NoC guaranteed services are provided by so called *looped containers*. These are implemented by virtual circuits, using an explicit time division multiplexing mechanism which they call *Temporally Disjoint Networks* (TDN) (refer to Sections 3.2.2 and 3.2.3 for more details).

In [Jantsch and Vitkowski 2005], the group addressed encoding issues and showed that lowering the voltage swing, then re-establishing reliability using error correction, actually resulted in better power saving than a number of dedicated power saving algorithms used for comparison.

iii **SPIN:** The SPIN network (*Scalable Programmable Integrated Network*) [Guerrier and Greiner 2000][Andriahantenaina and Greiner 2003] implements a fat-tree topology with two one-way 32bit datapaths at the link layer. The fat-tree is an interesting choice of irregular network, claimed in [Leiserson 1985] to be *nearly the best routing network* for a given amount of hardware. It is proven that for any given amount of hardware, a fat-tree can simulate any other network built from the same amount of hardware, with only a polylogarithmic slowdown in latency. This is in contrast to e.g. two-dimensional arrays or simple trees which exhibit polynomial slowdown when simulating other networks, and as such do not have any advantage over a sequential computer.

In SPIN, packets are sent via the network as a sequence of flits each of size 4 bytes. Wormhole routing is used, with no limit on packet size. The first flit contains the header, with one byte reserved for addressing, and the last byte of the packet contains the payload checksum. There are three types of flits; *first*, *data* and *last*. Link-level flow control is used to identify the flit type and act accordingly upon its content. The additional bytes in the header can be used for packet tagging for special services, and for special routing options. The performance of the network was evaluated primarily based on uniform randomly distributed load (see Section 5). It was noted that random hick-ups can be expected under high load. It was found that the protocol accounts for about 31% of the total throughput, a relatively large overhead. In 2003, a 32-port SPIN network was implemented in a 0.13μm CMOS process, the total area was 4.6 mm^2 (0.144 mm^2 per port), for an accumulated bandwidth of about 100Gbits/s.

iv **CHAIN:** The CHAIN network (*CHip Area INterconnect*) [Bainbridge and Furber 2002], developed at the University of Manchester, is interesting in that it is implemented entirely using asynchronous, or *clockless*, circuit techniques. It makes use of delay insensitive 1-of-4 encoding, and source routes BE packets. An easy adaption along a path consisting of links of different bit widths is supported. CHAIN is targeted for heterogeneous low power systems, in which the network is system specific. It has been implemented in a smart card, which benefits from the low idle power capabilities of asynchronous circuits. Work from the group involved with CHAIN concerns prioritization in asynchronous networks. In [Felicijan et al. 2003] an asynchronous low latency arbiter was presented, and its use in providing differentiated communication services in SoC was discussed, and in [Felicijan and Furber 2004] a router implementing the scheme was described.

v **MANGO:** The MANGO network (*Message-passing Asynchronous Network-on-chip providing Guaranteed services over OCP interfaces*), developed at the Technical University of Denmark, is another clockless NoC, targeted for coarse-grained GALS-type SoC. MANGO provides connection-less BE routing as well as connection-oriented guaranteed services (GS) [Bjerregaard and Sparsø 2005a]. In order to make for a simple design, the routers implement virtual channels (VCs) as separate physical buffers. GS connections are established by allocating a sequence of VCs through the network. While the routers themselves are implemented using area efficient bundled-data circuits, the links implement delay insensitive signal encoding. This makes global timing robust, because no timing assumptions are necessary between routers. A scheduling scheme called ALG (*Asynchronous Latency Guarantees*) [Bjerregaard and Sparsø

2005b], schedules access to the links, allowing latency guarantees to be made, which are not inversely dependent on the bandwidth guarantees, as is the case in TDM-based scheduling schemes. Network adapters provide OCP-based standard socket interfaces, based on the primitive routing services of the network [Bjerregaard et al. 2005]. This includes support for interrupts, based on virtual wires. The adapters also synchronize the clocked OCP interfaces to the clockless network.

vi **×PIPES:** ×pipes [Osso et al. 2003] and the accompanying NetChip compiler (a combination of ×pipesCompiler [Jalabert et al. 2004] and SUNMAP [Murali and Micheli 2004b]) are developed by University of Bologna and Stanford University. ×pipes consists of soft macros of switches and links that can be turned into instance-specific network components at instantiation time. It promotes the idea of pipelined links with a flexible number of stages to increase throughput. A go-back-N retransmission strategy is implemented as part of link-level error control, which reduces switch complexity, though at considerable delay since each flit is not acknowledged until it has been transmitted across the destination switch. The error is indicated by a CRC block running concurrently with switch operation. Thus the ×pipes architecture lends itself to be robust to interconnect errors. Overall, delay for a flit to traverse from across one link and node is 2N+M cycles where N is number of pipeline stages and M is switch stages. The ×pipesCompiler is a tool to automatically instantiate an application specific custom communication infrastructure using ×pipes components. It can tune flit size, degree of redundancy of the CRC error-detection, address space of cores, number of bits used for packet sequence count, maximum number of hops between any two network nodes, number of flit size, etc.

In a top-down design methodology, once the SoC floorplan is decided, the required network architecture is fed into the ×pipesCompiler. Examples of compiler optimization include removing redundant buffers from missing output ports of switches, sharing signals common to objects, etc. Via case studies presented in [Bertozzi et al. 2005], the NetChip compiler has been validated for mesh, torus, hypercube, Clos and butterfly NoC topologies for four video processing applications. Four routing algorithms: dimension-ordered, minimum-path, traffic splitting across minimum-path, and traffic splitting across all paths, is also part of the case study experiments. The floorplan of switches and links of NoC takes the IP block size into consideration. Results are available for average hop delay, area and power for mapping of each of the video application on the topologies. A light-weight implementation, named ×pipes-lite, presented in [Stergiou et al. 2005], is similar in to ×pipes in concept, but is however optimized for link latency, area and power, and provides direct synthesis path from SystemC description.

7. SUMMARY

NoC encompasses a wide spectrum of research, ranging from highly abstract software related issues, across system topology to physical level implementation. In this survey we have given an overview of current activities in the field. We have first stated the motivation for NoC and given an introduction of the basic concepts. In order to avoid the wide range of topics relevant to large scale IC design in general, we have assumed a view of NoC as a subset of SoC.

From a system level perspective, NoC is motivated by the demand for a well structured design approach in large scale SoCs. A modularized design methodology is needed, in order to make efficient use of the ever increasing availability of on-chip resources in terms of number of transistors and routing layers. Like-wise, programming these systems necessitates clear programming models and predictable behavior. NoC has the potential to provide modularity through the use of standard sockets such as OCP, and predictability through the implementation of guaranteed communication services. From a physical level perspective, with scaling of technologies into the DSM region, the increasing impact of wires on performance forces a differentiation between local and global communication. In order for global communication structures to exhibit scalability and high performance, segmentation, wire sharing and distributed control is employed.

In structuring our work, we have adopted a layered approach similar to OSI, and divided NoC research into four areas: *System*, *Network Adapter*, *Network* and *Link* research. In accordance with the view of NoC as a subset of SoC, we dealt first with the latter three areas of research, which relate directly to the NoC implementation.

The network adapter orthogonalizes communication and computation, enabling *communication-centric* design. It is thus the entity which enables a modularized design approach. Its main task is to decouple the core from the network, the purpose being to provide high-level network-agnostic communication services based on the low-level routing primitives provided by the network hardware. In implementing standard sockets, IP reuse becomes feasible, and the network adapter may thus hold the key to commercial success of NoC.

At the network level, issues such as network topology, routing protocols, flow control, and quality-of-service are dominant. With regards to topology, NoC is restricted by a 2D layout. This has made the grid a wide-spread topological choice. We have reviewed the most common routing schemes, *store-and-forward*, *wormhole* and *virtual cut-through* routing, and concluded that wormhole routing is by far the most common choice for NoC designs. The use of *virtual channels* in avoiding deadlocks and providing guaranteed services was illustrated and the motivation for guaranteed services was discussed. The predictability that such services incur facilitates easy system integration and analytical system verification, particularly relevant for real-time systems.

Unlike in macro networks, in NoC network adapter and network functionality is often implemented in hardware rather than in software. This is so, since NoC-based systems are more tightly bound, and simple, fast, power efficient solutions are required.

Link level research is much more hardware oriented. We have covered topics like synchronization, i.e. between clock domains, segmentation and pipelining of links in order to increase bandwidth and counteract physical limitations of DSM technologies, on-chip signaling such as low-swing drivers used to decrease the power usage in links, and future technologies such as on-chip wave guides and optical interconnects. Also we have discussed the reliability of long links, which are susceptible to a number of noise sources: *crosstalk*, *ground bounce*, *EMI* and *inter-symbol interference*. Segmentation helps keep the effect of these at bay, since the shorter a wire is the less influence they will have. Error detection and correction in on-chip interconnects was discussed, but this is not a dominating area of research. Different encoding schemes were discussed in relation to increasing bandwidth as well as reducing power consumption.

NoC facilitates communication-centric design, as opposed to traditional computation-centric design. From a system level perspective, topics relate to the role of NoC in future design flows. Key issues are modeling, design methodology and traffic characterization. The purpose of modeling is to evaluate trade-offs with regard to global traffic, in terms of power, area, design time, etc., while adhering to application requirements. With regard to design methodology, we identify two important characteristics of NoC, by which we classify a number of existing NoC solutions: (i) *parametrizability* of the NoC as a system level block and (ii) *granularity* of the NoC components by which the NoC is assembled. These characteristics greatly influence the nature of the design flow enabled by the particular NoC. As a tool towards identifying general requirements of a NoC, we have identified a set of traffic types, *latency-critical*, *data-streams* and *best-effort* traffic, which span the spectrum of possible traffic in a NoC-based system.

The basic performance parameters of NoC are latency, bandwidth and jitter. The basic cost factors are power consumption and area usage. At a higher level of abstraction, terms like aggregate bandwidth, bisection bandwidth, link utilization and network load can be used. These originate in multicomputer network theory and relate to data movement in general. Stepping up yet another abstraction level, benchmarks can be used for performance analysis. Currently no benchmarks exist specifically for NoC, but the use of benchmarks for parallel computers, as well as embedded systems benchmarks, has been reported.

Six case studies are conducted, explaining the design choices of the *ÆTHEREAL*, *NOSTRUM*, *SPIN*, *CHAIN*, *MANGO* and ×*PIPES* NoC implementations. CHAIN and ×PIPES target a platform-based design methodology, in which a heterogeneous network can be instantiated for a particular application. ÆTHEREAL, NOSTRUM and MANGO implement more complex features such as guaranteed services, and target a methodology which draws closer to backbone-based design. SPIN differs from the others in that it implements a fat-tree, rather than a grid-type topology. CHAIN and MANGO also differ in that they are implemented entirely using clockless circuit techniques, and as such inherently support globally asynchronous and locally synchronous (GALS) systems.

Continued technology scaling enables large scale SoC. NoCs facilitate a modular, scalable design approach that overcomes both system and physical level issues. The main job of the NoC designer of the future will be to dimension and structure the network, according to the communication needs of the SoC. At present, an interesting challenge lies in specifying ways to define these needs.

8. ACKNOWLEDGEMENTS

We would like to thank professors Jens Sparsø and Jan Madsen of the Department for Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU) for their tireless effort in helping us review, iterate and structure this survey. Also our grateful thanks to professor Axel Jantsch (KTH - Stockholm, Sweden) and Andrei Radulescu (Phillips - Eindhoven, Netherlands) for their valuable review of the survey as it was closing in on its final form, and to Mihai Budiu (Carnegie Melon - Pittsburgh, USA) for comments and suggestions. Finally, the extensive comments of the anonymous reviewers have helped in taking the survey to its final form.

REFERENCES

AGARWAL, A. 1999. The Oxygen project - Raw computation. Scientific American, (August), 44–47.

AGGARWAL, A. AND FRANKLIN, M. 2002. Hierarchical Interconnects for On-chip Clustering. In Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS) (April 2002). IEEE Computer Society, 602–609.

AHONEN, T., SIGÜENZA-TORTOSA, D. A., BIN, H., AND NURMI, J. 2004. Topology optimization for application-specific networks-on-chip. In International Workshop on System Level Interconnect Prediction (SLIP) (February 2004). ACM, 53–60.

AL-TAWIL, K. M., ABD-EL-BARR, M., AND ASHRAF, F. 1997. A survey and comparison of wormhole routing techniques in a mesh networks. IEEE Network 11, 38–45.

AMDE, M., FELICIJAN, T., EDWARDS, A. E. D., AND LAVAGNO, L. 2005. Asynchronous on-chip networks. IEE Proceedings of Computers and Digital Techniques 152, 273–283.

ANDREASSON, D. AND KUMAR, S. 2004. On improving best-effort throughput by better utilization of guaranteed-throughput channels in an on-chip communication system. In Proceeding of 22th IEEE Norchip Conference (Nov 2004). IEEE.

ANDREASSON, D. AND KUMAR, S. 2005. Slack-time aware routing in NoC systems. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 2353–2356.

ANDRIAHANTENAINA, A. AND GREINER, A. 2003. Micro-network for SoC : Implementation of a 32-port spin network. In Proceedings of Design, Automation and Test in Europe Conference and Exhibition (2003). IEEE, 1128–1129.

ARM. 2004. AMBA Advanced eXtensible Interface (AXI) Protocol Specification, Version 1.0. http://www.arm.com.

ARTERIS. 2005. A comparison of network-on-chip and busses. White paper downloadable from http://www.arteris.com/noc_whitepaper.pdf.

BAILEY, D., BARSZCZ, E., BARTON, J., BROWNING, D., CARTER, R., DAGUM, L., FATOOHI, R., FINEBERG, S., FREDERICKSON, P., LASINSKI, T., SCHREIBER, R., SIMON, H., VENKATAKRISHNAN, V., AND WEERATUNGA, S. 1994. RNR technical report RNR-94-007. Technical report, NASA Ames Research Center.

BAINBRIDGE, J. AND FURBER, S. 2002. CHAIN: A delay-insensitive chip area interconnect. IEEE Micro 22, 5 (October), 16–23.

BAINBRIDGE, W. AND FURBER, S. 2001. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems (ASYNC) (March 2001). 118 – 126.

BANERJEE, N., VELLANKI, P., AND CHATHA, K. S. 2004. A power and performance model for network-on-chip architectures. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 1250–1255.

BEIGNE, E., CLERMIDY, F., VIVET, P., CLOUARD, A., AND RENAUDIN, M. 2005. An asynchronous NOC architecture providing low latency service and its multi-level design framework. In Proceedings of the 11th International Symposium on Asynchronous Circuits and Systems (ASYNC) (2005). IEEE, 54–63.

BENINI, L. AND MICHELI, G. D. 2001. Powering network-on-chips. In The 14th International Symposium on System Synthesis (ISSS) (October 2001). IEEE, 33–38.

BENINI, L. AND MICHELI, G. D. 2002. Networks on chips: A new SoC paradigm. IEEE Computer 35, 1 (January), 70–78.

BERTOZZI, D., JALABERT, A., MURALI, S., TAMHANKAR, R., STERGIOU, S., BENINI, L., AND DE MICHELI, G. 2005. NoC synthesis flow for customized domain specific multiprocessor Systems-on-Chip. In Transactions on Parallel and Distributed Systems (February 2005). IEEE, 113–129.

BHOJWANI, P. AND MAHAPATRA, R. 2003. Interfacing cores with on-chip packet-switched networks. In Proceedings of the Sixteenth International Conference on VLSI Design. (2003). 382–387.

BJERREGAARD, T., MAHADEVAN, S., OLSEN, R. G., AND SPARSØ, J. 2005. An OCP compliant network adapter for gals-based soc design using the MANGO network-on-chip. In Proceedings of International Symposium on System-on-Chip (ISSoC) (2005). IEEE.

BJERREGAARD, T., MAHADEVAN, S., AND SPARSØ, J. 2004. A channel library for asynchronous circuit design supporting mixed-mode modeling. In Proceedings of the Fourteenth International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS) (2004). Springer, 301–310.

BJERREGAARD, T. AND SPARSØ, J. 2005a. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (March 2005). IEEE, 1226–1231.

BJERREGAARD, T. AND SPARSØ, J. 2005b. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In Proceedings of the 11th International Symposium on Advanced Research in Asynchronous Circuits and Systems (March 2005). IEEE, 34–43.

BOGLIOLO, A. 2001. Encodings for high-performance energy-efficient signaling. In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED) (August 2001). 170–175.

BOLOTIN, E., CIDON, I., GINOSAUR, R., AND KOLODNY, A. 2004. QNoC: QoS architecture and design process for network-on-chip. vol. 50 (2004). Elsevier North-Holland, Inc., 105–128.

CATTHOOR, F., CUOMO, A., MARTIN, G., GROENEVELD, P., RUDY, L., MAEX, K., DE STEEG, P. V., AND WILSON, R. 2004. How can system level design solve the interconnect technology scaling problem. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 332–337.

CHAPIRO, D. 1984. Globally-Asynchronous Locally-Synchronous Systems. Ph. D. thesis, Stanford University. Report No. STAN-CS-84-1026.

CHELCEA, T. AND NOWICK, S. M. 2001. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In Proceedings of the 38th Design Automation Conference (DAC) (June 2001). IEEE, 21–26.

CHIU, G.-M. 2000. The odd-even turn model for adaptive routing. IEEE Transactions on Parallel and Distributed Systems 11, 729 –738.

COLE, R. J., MAGGS, B. M., AND SITARAMAN, R. K. 2001. On the benefit of supporting virtual channels in wormhole routers. Journal of Computer and System Sciences 62, 152–177.

CULLER, D. E., SINGH, J. P., AND GUPTA, A. 1998. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann. 1st Edition.

DALLY, W. J. 1990. Performance analysis of k-ary n-cube interconnection networks. IEEE Transactions on Computer 39, 6 (June), 775 – 785.

DALLY, W. J. 1992. Virtual-channel fbw control. IEEE Transactions on Parallel and Distributed Systems 3, 2 (March), 194 – 205.

DALLY, W. J. AND AOKI, H. 1993. Deadlock-free adaptive routing in multicomputer networks using virtual channels. IEEE Transactions on Parallel and Distributed Systems 4, 4 (April), 466 – 475.

DALLY, W. J. AND SEITZ, C. L. 1987. Deadlock-free message routing in multiprocessor interconnection networks. IEEE Transactions on Computers C-36, 5 (May), 547–553.

DALLY, W. J. AND TOWLES, B. 2001. Route packets, not wires: On-chip interconnection networks. In Proceedings of the 38th Design Automation Conference (DAC) (June 2001). IEEE, 684–689.

DE MELLO, A. V., OST, L. C., MORAES, F. G., AND CALAZANS, N. L. V. 2004. Evaluation of routing algorithms on mesh based nocs. Technical report (May), Faculdade de Informatica PUCRS - Brazil.

DICK, R. Embedded system synthesis benchmarks suite. http://www.ece.northwestern.edu/ dickrp/e3s/.

DIELISSEN, J., RADULESCU, A., GOOSSENS, K., AND RIJPKEMA, E. 2003. Concepts and implementation of the phillips network-on-chip. In Proceedings of the IP based SOC (IPSOC) (November 2003). IFIP.

DOBBELAERE, I., HOROWITZ, M., AND GAMAL, A. E. 1995. Regenerative feedback repeaters for programmable interconnections. IEEE Journal of Solid-State Circuits 30, 11 (November), 1246–1253.

DOBKIN, R., GINOSAUR, R., AND SOTIRIOU, C. P. 2004. Data synchronization issues in GALS SoCs. In Proceedings of the 10th IEEE International Symposium on Asynchronous Circuits and Systems (2004). IEEE, 170–179.

DUATO, J. 1993. A new theory of deadlock-free adaptive routing in wormhole networks. IEEE Transactions on Parallel and Distributed Systems 4, 12 (December), 1320–1331.

DUATO, J. 1995. A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. IEEE Transactions on Parallel and Distributed Systems 6, 10 (October), 1055–1067.

DUATO, J. 1996. A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks. IEEE Transactions on Parallel and Distributed Systems 7, 8 (August), 841–854.

DUATO, J. AND PINKSTON, T. M. 2001. A general theory for deadlock-free adaptive routing using a mixed set of resources. IEEE Transactions on Parallel and Distributed Systems 12, 12 (December), 1219–1235.

DUATO, J., YALAMANCHILI, S., AND NI, L. 2003. Interconnection Networks: An Engineering Approach. Morgan Kaufmann.

FELICIJAN, T., BAINBRIDGE, J., AND FURBER, S. 2003. An asynchronous low latency arbiter for quality of service (QoS) applications. In Proceedings of the 15th International Conference on Microelectronics (ICM) (December 2003). IEEE, 123–126.

FELICIJAN, T. AND FURBER, S. B. 2004. An asynchronous on-chip network router with quality-of-service (QoS) support. In Proceedings IEEE International SOC Conference (2004). IEEE, 274–277.

FITZPATRICK, T. 2004. System verilog for VHDL users. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE Computer Society, 21334.

FORSELL, M. 2002. A scalable high-performance computing solution for networks on chips. IEEE Micro 22, 5, 46–55.

GAUGHAN, P. T., DAO, B. V., YALAMANCHILI, S., AND SCHIMMEL, D. E. 1996. Distributed, deadlock-free routing in faulty, pipelined, direct interconnection networks. IEEE Transactions on Computers 45, 6 (June), 651–665.

GENKO, N., ATIENZA, D., DE MICHELI, G., BENINI, L., MENDIAS, J., HERMIDA, R., AND CATTHOOR, F. 2005. A novel approach for network on chip emulation. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 2365–2368.

GERSTLAUER, A. 2003. Communication abstractions for system-level design and synthesis. Technical Report TR-03-30 (October), Center for Embedded Computer Systems, University of California, Irvine, CA 92697-3425, USA.

GINOSAUR, R. 2003. Fourteen ways to fool your synchrononizer. In Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems (2003). IEEE, 89–96.

GLASS, C. J. AND NI, L. M. 1994. The turn model for adaptive routing. Journal of the Association for Computing Machinery 41, 874–902.

GOOSSENS, K., DIELISSEN, J., GANGWAL, O. P., PESTANA, S. G., RADULESCU, A., AND RIJPKEMA, E. 2005. A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (March 2005). IEEE, 1182–1187.

GOOSSENS, K., DIELISSEN, J., AND RADULESCU, A. 2005. Æthereal network on chip: Concepts, architectures and implementations. IEEE Design & Test of Computers 22, 5, 414–421.

GOOSSENS, K., MEERBERGEN, J. V., PEETERS, A., AND WIELAGE, P. 2002. Networks on silicon: Combining best-effort and guaranteed services. In Proceedings of the Design, Automation and Test in Europe Conference (DATE) (2002). IEEE, 196–200.

GUERRIER, P. AND GREINER, A. 2000. A generic architecture for on-chip packet-switched interconnections. In Proceedings of the Design Automation and Test in Europe (DATE) (March 2000). IEEE, 250–256.

GUO, M., NAKATA, I., AND YAMASHITA, Y. 2000. Contention-free communication scheduling for array redistribution. Parallel Computing 26, 1325–1343.

HANSSON, A., GOOSSENS, K., AND RADULESCU, A. 2005. A unified approach to constrained mapping and routing on networks-on-chip architectures. In CODES/ISSS (2005). ACM/IEEE, 75–80.

HARMANCI, M., ESCUDERO, N., LEBLEBICI, Y., AND IENNE, P. 2005. Quantitative modelling and comparison of communication schemes to guarantee quality-of-service in networks-on-chip. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 1782–1785.

HAUCK, S. 1995. Asynchronous design methodologies: an overview. Proceedings of the IEEE 83, 1 (January), 69–93.

HAVEMANN, R. H. AND HUTCHBY, J. A. 2001. High-performance interconnects: An integration overview. Proceedings of the IEEE 89, 5 (May), 586 – 601.

HAVERINEN, A., LECLERCQ, M., WEYRICH, N., AND WINGARD, D. 2002. SystemC based SoC communication modeling for the OCP protocol. White paper downloadable from http://www.ocpip.org.

HEILIGER, H.-M., NAGEL, M., ROSKOS, H. G., AND KURZ, H. 1997. Thin-film microstrip lines for mm and sub-mm-wave on-chip interconnects. In IEEE MTT-S International Microwave Symposium Digest, vol. 2 (June 1997). 421–424.

HO, R., MAI, K., AND HOROWITZ, M. 2003. Efficient on-chip global interconnects. In Symposium on VLSI Circuits. Digest of Technical Papers. (June 2003). IEEE, 271–274.

HO, R., MAI, K. W., AND HOROWITZ, M. A. 2001. The future of wires. Proceedings of the IEEE 89, 4 (April), 490 – 504.

Hu, J. and Marculescu, R. 2004a. Application-specific buffer space allocation for networks-on-chip router design. In ICCAD (2004). IEEE Computer Society / ACM, 354–361.

Hu, J. and Marculescu, R. 2004b. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 10234–10240.

ITRS. 2001. International technology roadmap for semiconductors (ITRS) 2001. Technical report, International Technology Roadmap for Semiconductors.

ITRS. 2003. International technology roadmap for semiconductors (ITRS) 2003. Technical report, International Technology Roadmap for Semiconductors.

Jalabert, A., Murali, S., Benini, L., and Micheli, G. D. 2004. xpipesCompiler: A tool for instantiating application specific networks-on-chip. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 884–889.

Jantsch, A. 2003. Communication performance in networks-on-chip. Slides downloadable from http://www.ele.kth.se/ axel/presentations/2003/Stringent.pdf.

Jantsch, A. and Tenhunen, H. 2003. Networks on Chip. Kluwer Academic Publishers.

Jantsch, A. and Vitkowski, R. L. A. 2005. Power analysis of link level and end-to-end data protection in networks-on-chip. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 1770–1773.

Juurlink, B. H. H. and Wijshoff, H. A. G. 1998. A quantitative comparison of parrallel computation models. ACM Transactions on Computer Systems 16, 3 (August), 271–318.

Kapur, P. and Saraswat, K. C. 2003. Optical interconnects for future high performance intergrated circuits. Physica E 16, Issue 3-4, 620–627.

Karim, F., Nguyen, A., and Dey, S. 2002. An interconnect architecture for networking systems on chips. IEEE Micro 22, 36–45.

Karim, F., Nguyen, A., Dey, S., and Rao, R. 2001. On-chip communication architecture for OC-768 network processors. In Proceedings of the 38th Design Automation Conference (DAC) (June 2001). ACM, 678–683.

Kim, D., Lee, K., joong Lee, S., and Yoo, H.-J. 2005. A reconfigurable crossbar switch with adaptive bandwidth control for networks-on-chip. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 2369–2372.

Kim, K., Lee, S.-J., Lee, K., and Yoo, H.-J. 2005. An arbitration look-ahead scheme for reducing end-to-end latency in networks-on-chip. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 2357–2360.

Kumar, S., Jantsch, A., Soininen, J.-P., Forsell, M., Millberg, M., Oberg, J., Tiensyrjä, K., and Hemani, A. 2002. A network-on-chip architecture and design methodology. In Proceedings of the Computer Society Annual Symposium on VLSI (ISVLSI) (April 2002). IEEE Computer Society, 117–124.

Kurd, N., Barkatullah, J., Dizon, R., Fletcher, T., and Madland, P. 2001. Multi-GHz clocking scheme for Intel pentium 4 microprocessor. In Digest of Technical Papers. International Solid-State Circuits Conference (ISSCC) (February 2001). IEEE, 404–405.

Lahiri, K., Raghunathan, A., and Dey, S. 2001. Evaluation of the traffic-performance characteristics of system-on-chip communication architectures. In Proceedings of the 14th International Conference on VLSI Design (2001). IEEE, 29–35.

Lahiri, K., Raghunathan, A., Lakshminarayana, G., and Dey, S. 2000. Communication architecture tuners: A methodology for the design of high-performance communication architectures for system-on-chips. In Proceedings of the Design Automation Conference, DAC (2000). IEEE, 513–518.

Lee, K. 1998. On-chip interconnects - gigahertz and beyond. Solid State Technology 41, 9 (September), 85–89.

Leiserson, C. E. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing. IEEE transactions on Computers c-34, 10, 892–901.

Leroy, A., Marchal, P., Shickova, A., Catthoor, F., Robert, F., and Verkest, D. 2005. Spatial division multiplexing: a novel approach for guaranteed throughput on nocs. In CODES/ISSS (2005). ACM/IEEE, 81–86.

Liang, J., Laffely, A., Srinivasan, S., and Tessier, R. 2004. An architecture and compiler for scalable on-chip communication. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12, 7, 711–726.

LIANG, J., SWAMINATHAN, S., AND TESSIER, R. 2000. ASOC: A scalable, single-chip communications architecture. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques 2000 (October 2000). 37–46.

LIU, J., ZHENG, L.-R., AND TENHUNEN, H. 2004. Interconnect intellectual property for network-on-chip (NoC). Journal of Systems Architecture 50, 65–79.

LOGHI, M., ANGIOLINI, F., BERTOZZI, D., BENINI, L., AND ZAFALON, R. 2004. Analyzing on-chip communication in a MPSoC environment. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 752–757.

MADSEN, J., MAHADEVAN, S., VIRK, K., AND GONZALEZ, M. 2003. Network-on-chip modeling for system-level multiprocessor simulation. In Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS) (Dec 2003). IEEE, 82–92.

MAHADEVAN, S., STORGAARD, M., MADSEN, J., AND VIRK, K. 2005. ARTS: A system-level framework for modeling MPSoC components and analysis of their causality. In The 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) (September 2005). IEEE Computer Society.

MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In Proceedings of 27th International Symposium on Computer Architecture (June 2000). 161–171.

MEINCKE, T., HEMANI, A., KUMAR, S., ELLERVEE, P., OBERG, J., OLSSON, T., NILSSON, P., LINDQVIST, D., AND TENHUNEN, H. 1999. Globally asynchronous locally synchronous architecture for large high-performance ASICs. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), vol. 2 (June 1999). 512 –515.

MILLBERG, M., NILSSON, E., THID, R., AND JANTSCH, A. 2004. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network-on-chip. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 890–895.

MIZUNO, M., DALLY, W. J., AND ONISHI, H. 2001. Elastic interconnects: Repeater-inserted long wiring capable of compressing and decompressign data. In Proceedings of the International Solid-State Circuits Conference (2001). IEEE, 346–347, 464.

MORAES, F., CALAZANS, N., MELLO, A., MÖLLER, L., AND OST, L. 2004. HERMES: An infrastructure for low area overhead packet-switching networks on chip. vol. 38 (2004). Elsevier, 69–93.

MULLINS, R. AND MOORE, A. W. S. 2004. Low-latency virtual-channel routers for on-chip networks. In Proceedings of the 31st Annual International Symposium on Computer Architecture (2004). IEEE, 188–197.

MURALI, S. AND MICHELI, G. D. 2004a. Bandwidth-constrained mapping of cores onto noc architectures. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 20896–20902.

MURALI, S. AND MICHELI, G. D. 2004b. SUNMAP: A tool for automatic topology selection and generation for NoCs. In In Proceedings of the 41st Design Automation Conference (DAC) (June 2004). IEEE, 914–919.

MUTTERSBACH, J., VILLIGER, T., AND FICHTNER, W. 2000. Practical design of globally-asynchronous locally-synchronous systems. In Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC) (April 2000). IEEE Computer society, 52–59.

NAKAMURA, K. AND HOROWITZ, M. A. 1996. A 50% noise reduction interface using low-weight coding. In Symposium on VLSI Circuits Digest of Technical Papers (June 1996). IEEE, 144–145.

NEDOVIC, N., OKLOBDZIJA, V. G., AND WALKER, W. W. 2003. A clock skew absorbing flp-fbp. In Proceedings of the International Solid-State Circuits Conference (2003). IEEE, 342–497.

NEEB, C., THUL, M., WEHN, N., NEEB, C., THUL, M., AND WEHN, N. 2005. Network-on-chip-centric approach to interleaving in high throughput channel decoders. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 1766–1769.

NIELSEN, S. F. AND SPARSØ, J. 2001. Analysis of low-power SoC interconnection networks. In Proceedings of Nordchip 2001 (2001). 77–86.

OBERG, J. 2003. Clocking Strategies for Networks-on-Chip. 153–172. Kluwer Academic Publishers.

OCPIP. 2003a. The importance of sockets in SoC design. White paper downloadable from http://www.ocpip.org.

OCPIP. 2003b. Open Core Protocol (OCP) Specification, Release 2.0. http://www.ocpip.org.

OKLOBDZIJA, V. G. AND SPARSØ, J. 2002. Future directions in clocking multi-GHz systems. In Proceedings of the 2002 International Symposium on Low Power Electronics and Design, 2002 (ISLPED '02) (August 2002). ACM, 219.

OSSO, M. D., BICCARI, G., GIOVANNINI, L., BERTOZZI, D., AND BENINI, L. 2003. ×pipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In Proceedings of 21st International Conference on Computer Design (ICCD) (2003). IEEE Computer Society, 536–539.

OST, L., MELLO, A., PALMA, J., MORAES, F., AND CALAZANS, N. 2005. MAIA - a framework for networks on chip generation and verification. In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC) (January 2005). IEEE.

PANDE, P., GRECU, C., JONES, M., IVANOV, A., AND SALEH, R. 2005. Effect of traffic localization on energy dissipation in NoC-based interconnect. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 1774–1777.

PANDE, P. P., GRECU, C., IVANOV, A., AND SALEH, R. 2003. Design of a switch for network-on-chip applications. IEEE International Symposium on Circuits and Systems (ISCAS) 5, 217–220.

PEH, L.-S. AND DALLY, W. J. 1999. Flit-reservation fbw control. In Proceedings of the 6th International Symposium on High-Performance Computer Architecutre (HPCA) (1999). IEEE Computer Society, 73–84.

PEH, L.-S. AND DALLY, W. J. 2001. A delay model for router microarchitectures. IEEE Micro 21, 26–34.

PESTANA, S., RIJPKEMA, E., RADULESCU, A., GOOSSENS, K., AND GANGWAL, O. 2004. Cost-performance trade-offs in networks on chip: a simulation-based approach. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 764–769.

PHILIPS SEMICONDUCTORS. 2002. Device Transaction Level (DTL) Protocol Specification, Version 2.2.

PIGUET, C., JACQUES, HEER, C., O'CONNOR, I., AND SCHLICHTMANN, U. 2004. Extremely low-power logic. In Proceedings of Design, Automation and Testing in Europe Conference (DATE), C. Piguet, Ed. (2004). IEEE, 1530–1591.

PIRRETTI, M., LINK, G., BROOKS, R. R., VIJAYKRISHNAN, N., KANDEMIR, M., AND IRWIN, M. 2004. Fault tolerant algorithms for network-on-chip interconnect. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI. (2004). IEEE, 46–51.

RADULESCU, A., DIELISSEN, J., GOOSSENS, K., RIJPKEMA, E., AND WIELAGE, P. 2004. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and fexible network configuration. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 878–883.

RIJPKEMA, E., GOOSSENS, K., AND WIELAGE, P. 2001. A router architecture for networks on silicon. In Proceeding of the 2nd Workshop on Embedded Systems (2001). 181–188.

RIJPKEMA, E., GOOSSENS, K. G. W., RADULESCU, A., DIELISSEN, J., MEERBERGEN, J. V., WIELAGE, P., AND WATERLANDER, E. 2003. Trade offs in the design of a router with both guaranteed and best-effort services for networks-on-chip. In Proceedings of the Design, Automation and Test in Europe Conference (DATE) (2003). IEEE, 350–355.

RIXNER, S., DALLY, W. J., KAPASI, U. J., KHAILANY, B., LÓPEZ-LAGUNAS, A., MATTSON, P. R., AND OWENS, J. D. 1998. A bandwidth-efficient architecture for media processing. In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (1998). 3–13.

ROSTISLAV, D., VISHNYAKOV, V., FRIEDMAN, E., AND GINOSAUR, R. 2005. An asynchronous router for multiple service levels networks on chip. In Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC). (2005). IEEE, 44–53.

SATHE, S., WIKLUND, D., AND LIU, D. 2003. Design of a switching node (router) for on-chip networks. In Proceedings of the Fifth International Conference on ASIC (2003). IEEE, 75–78.

SIA. 1997. National technology roadmap for semiconductors 1997. Technical report, Semiconductor Industry Association.

SIGUENZA-TORTOSA, D., AHONEN, T., AND NURMI, J. 2004. Issues in the development of a practical NoC: the Proteo concept. In Integration, the VLSI Journal (2004). Elsevier, 95–105.

SIMUNIC, T. AND BOYD, S. 2002. Managing power consumption in networks-on-chips. In Proceedings of the Design, Automation and Test in Europe Conference (DATE) (2002). IEEE Computer Society, 110–116.

SINGH, M. AND NOWICK, S. 2000. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC). (2000). IEEE Computer Society, 198–209.

SPARSØ, J. AND FURBER, S. 2001. Principles of Asynchronous Circuit Design. Kluwer Academic Publishers, Boston.

STERGIOU, S., ANGIOLINI, F., CARTA, S., RAFFO, L., BERTOZZI, D., AND MICHELI, G. D. 2005. ×pipes lite: A synthesis oriented design library for networks on chips. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (March 2005). IEEE.

SVENSSON, C. 2001. Optimum voltage swing on on-chip and off-chip interconect. Manuscript available on authors web page at http://www.ek.isy.liu.se/ christer/ManuscriptSwing.pdf.

SYLVESTER, D. AND KEUTZER, K. 2000. A global wiring paradigm for deep submicron design. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 19, 242–252.

SYSTEMC. 2002. The SystemC Version 2.0.1. Web Forum (www.systemc.org).

TAMIR, Y. AND FRAZIER, G. L. 1988. High-performance multiqueue buffers for VLSI communication switches. In Proceedings of the 15th Annual International Symposium on Computer Architecture (1988). IEEE Computer Society, 343–354.

TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMAN, H., JOHNSON, P., LEE, J.-W., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2002. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs (2002).

TORTOSA, D. A. AND NURMI, J. 2004. Packet scheduling in proteo network-on-chip. In Parallel and Distributed Computing and Networks (2004). IASTED/ACTA Press, 116–121.

VAIDYA, R. S., SIVASUBRAMANIAM, A., AND DAS, C. R. 2001. Impact of virtual channels and adaptive routing on application performance. IEEE Transactions on Parallel and Distributed Systems 12, 2 (February), 223 – 237.

VARATKAR, G. AND MARCULESCU, R. 2002. Traffic analysis for on-chip networks design of multimedia applications. In Proceedings of the 39th Design Automation Conference (DAC) (June 2002). ACM, 795–800.

VSI ALLIANCE. 2000. Virtual component interface standard Version 2. Available from VSI Alliance (www.vsi.org).

WANG, H.-S., ZHU, X., PEH, L.-S., AND MALIK, S. 2002. Orion: a power-performance simulator for interconnection networks. In Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (November 2002). IEEE Computer Society Press, 294–305.

WEBER, W.-D., CHOU, J., SWARBRICK, I., AND WINGARD, D. 2005. A quality-of-service mechanism for interconnection networks in system-on-chips. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (March 2005). IEEE, 1232–1237.

WIEFERINK, A., KOGEL, T., LEUPERS, R., ASCHEID, G., MEYR, H., BRAUN, G., AND NOHL, A. 2004. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE Computer Society, 1256–1261.

WIELAGE, P. AND GOOSSENS, K. 2002. Networks on silicon: Blessing or nightmare? In Proceedings of the Euromicro Symposium on Digital System Design (DSD) (September 2002). IEEE, 196–200.

WORM, F., THIRAN, P., MICHELI, G. D., AND IENNE, P. 2005. Self-calibrating networks-on-chip. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 2361–2364.

XANTHOPOULOS, T., BAILEY, D., GANGWAR, A., GOWAN, M., JAIN, A., AND PREWITT, B. 2001. The design and analysis of the clock distribution network for a 1.2 GHz alpha microprocessor. In Digest of Technical Papers, IEEE International Solid-State Circuits Conference, 2001 ISSCC. 2001. (2001). IEEE, 402 –403.

XU, J., WOLF, W., HENKEL, J., AND CHAKRADHAR, S. 2005. A methodology for design, modeling, and analysis of networks-on-chip. In International Symposium on Circuits and Systems (ISCAS) (May 2005). IEEE, 1778–1781.

XU, J., WOLF, W., HENKEL, J., CHAKRADHAR, S., AND LV, T. 2004. A case study in networks-on-chip design for embedded video. In Proceedings of Design, Automation and Testing in Europe Conference (DATE) (Febuary 2004). IEEE, 770–775.

ZHANG, H., GEORGE, V., AND RABAEY, J. M. 1999. Low-swing on-chip signaling techniques: Effectiveness and robustness. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 8, 3 (August), 264 – 272.

ZHANG, H., PRABHU, V., GEORGE, V., WAN, M., BENES, M., ABNOUS, A., AND RABAEY, J. M. 2000. A 1 V heterogeneous reconfigurable processor IC for baseband wireless applications. In International Solid-State Circuits Conference. Digest of Technical Papers (ISSCC) (2000). IEEE, 68–69.

ZIMMER, H. AND JANTSCH, A. 2003. A fault tolerant notation and error-control scheme for switch-to-switch busses in a network-on-chip. In Proceedings of Conference on Hardware/Software Codesign and System Synthesis Conference CODES ISSS (October 2003). ACM, 188–193.

Chapter 4

The ARTS Modeling Environment

This chapter consists of the following papers.

#2: Jan Madsen, Shankar Mahadevan, Kashif Virk and Mercury Gonzalez. "Network-on-Chip Modeling for System-Level Multiprocessor Simulation." *In Proceedings of the 24th Real-Time Systems Symposium (RTSS), Cancun Mexico.* IEEE, Dec. 2003: 265-274.

#3: Jan Madsen, Shankar Mahadevan, and Kashif Virk. "Network-Centric System-Level Model for Multiprocessor System-on-Chip Simulation." *Interconnect-Centric Design for Advanced SoC and NoC. Eds. Nurmi J., Tenhunen H., Isoaho J., and Jantsch A. Dordrecht, The Netherlands.* Kluwer Publications, 2004: 341-365.

#4: Shankar Mahadevan, Michael Storgaard, Jan Madsen, and Kashif Virk. "ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality" *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), Atlanta USA.* IEEE, Sept. 2005: 480-483.

From this group only Paper #3 and Paper #4 are presented in this chapter. Paper #3 covers the concepts and results presented in Paper #2 and therefore, Paper #2 is not presented here. We refer the interested readers to Appendix 6 for the full text of Paper #2. With regards to nomenclature, the ARTS framework in Paper #2 and #3 is referred to as 'abstract system-level model' or 'system-level RTOS modeling framework'.

4.1 Network-Centric System-Level Model for Multiprocessor System-on-Chip Simulation

Chapter 1

NETWORK-CENTRIC SYSTEM-LEVEL MODEL FOR MULTIPROCESSOR SOC SIMULATION

Jan Madsen
Shankar Mahadevan
Kashif Virk
Informatics and Mathematical Modeling, Technical University of Denmark
Richard Petersens Plads, building 322, DK2800 Kgs. Lyngby, Denmark
{jan,sm,virk}@imm.dtu.dk

1. Introduction

In this chapter, we present a modelling environment which supports design space exploration at the system-level for mapping an application onto the architecture platform while giving a central role to the effects of the network-on-chip (NoC). The primary goal of system-level modelling is to formulate a model within which a broad class of designs can be developed and explored. Ultimately, this allows designers to efficiently implement instances of systems within a single modelling style as supported by common design tools and methodologies. To support the designers of single-chip based embedded systems, which includes multiprocessor platforms running dedicated real-time operating systems (RTOS's) as well as the effects of on-chip interconnect network, a modeling/simulation environment is required to support the analysis of:

- Network performance under different traffic and load conditions.
- Consequences of different mappings of tasks to processors (software or hardware).
- Effects of RTOS selection, including scheduling, synchronization and resource allocation policies.

An on-chip network model can provide provisions for run-time inspection and observation of the communication. Using this approach,

2

implementations of the most promising network alternatives can be prototyped and characterized in terms of performance and overhead. Taking communication into account during hardware/software mapping is essential in order to obtain optimized solutions as emphasized in [13]. Also, [12] and [18] show the importance of evaluating the communication media and how the choice of communication clearly impacts the overall architecture of a SoC.

The rest of the chapter is organized as follows: In Section 2, we discuss various issues related to NoC modelling such as the requirements for modelling general network structures, the interface between the processing elements (PE's) and the network, and the possible usages of a NoC model. Section 3 gives an overview of our abstract SoC model with emphasis on modelling the NoC, while Section 4 gives a detailed description of the implementation of the NoC model. In Section 5, we present a simple example to illustrate the capabilities of our NoC model. Finally, Section 6 gives a summary and concluding remarks.

2. Issues in NoC Modelling

Architecturally, an on-chip network is defined by its topology and the protocol running on it. The topology concerns the geometry of the communication links while the protocol dictates how these links are utilized. Many combinations of topology and protocol exist for an efficient communication of one or more predominant traffic patterns. For example, in [14], packet-switched NoC concepts have been applied to a 2-D mesh network topology whereas in [10], such concepts have been applied to a butterfly fat tree topology. While there are several mature methodologies for modelling and evaluating the PE architectures, there is relatively little research done to port the on-chip communication to the system-level. In [20], attempts have been made to fill this gap by proposing a NoC modelling methodology based upon ideas borrowed from the object-oriented design domain and implementing those ideas in Ptolemy II.

The performance of a network is closely connected to its architecture. Network performance is measured in quantitative terms, such as latency, bandwidth, power consumption, and area usage, as well as, in qualitative terms, such as network configurability (static or dynamic), quality of service (QoS), etc. Predictability of performance is necessary for NoC designers to take early decisions based on the NoC performance before actual implementation. Numerous studies have been done for deadlock, livelock, congestion-avoidance, error-correction, connection setup/teardown, etc. to provide a certain predictable network behavior [6]. Even lower-level engineering techniques like low-swing drivers, signal encoding, etc., have been proposed to overcome network communication uncertainties [3, 4, 11]. Many of these network aspects are custom-tuned to fit the requirements of the application running on top of it.

Throughout this chapter, we use latency as the primary metric to ascertain the performance of a NoC. Network latency is defined as the time taken to move data from the source PE to the destination PE. It includes the message processing overhead, link delay, and the data processing delay at the intermediate nodes. Network latency is a function of the network topology (which determines the number of nodes and links comprising a network) and the communication protocol (which determines the processing requirements for routing and flow control). If two communicating tasks are allocated to different processors, data will have to be transferred over a communication medium and the message transfer time will depend on the message size and the state of the network.

The state of an on-chip network at any instant is given by the number of actively transmitting PE's and the messages within its nodes and links. The state of a network dictates which resources of the network are currently in use and which ones can be available for future use. This provides a measure of the network services available to the system, which would affect its performance. We define *network services* as the system-level characterization of the network resource allocation and scheduling activities. For a given topology-protocol combination, the effect of changes in network services changes the resources available for a given communication event, thus, affecting its latency.

2.1 Network Aspects

Since most of the future embedded applications are likely to be real-time applications running on multiprocessor SoC's, the fundamental properties required of future NoC's to provide these services are: multi-hop, concurrency, and sharing. Although different on-chip networks manifest different subsets of the above-mentioned properties, a network should contain all of them in order to be a successful NoC.

- Multi-hop implies segmented communication in which communication events (majority of messages) pass through intermediate nodes while traversing from the source to the destination.

- Concurrency implies multiple simultaneous communications. It represents the ability of the network to successfully carry out more than one communication at the same time.

- Sharing implies quasi-simultaneous resource usage. It, inherently, allows many communication events to occupy some or all of the resources in an interleaved fashion.

Though defined separately, these properties are closely related to each other by the underlying topology and protocol implementations. If no direct path exists between two communicating PE's, then multi-hop is required. In multi-hop networks, links are connected via nodes. This, inherently, introduces sub-divisioning (segmentation) of the network. Many communication events can, thus, occur in different segments of the network allowing concurrency. Concurrency, in essence, allows co-existence of different communication events. Sharing requires that links be connected to multiple source and destination pairs at the same time. Sharing, essentially, allows the creation of multiple communication events in the network. Sharing can be either spatial (resource sharing) or temporal (time sharing).

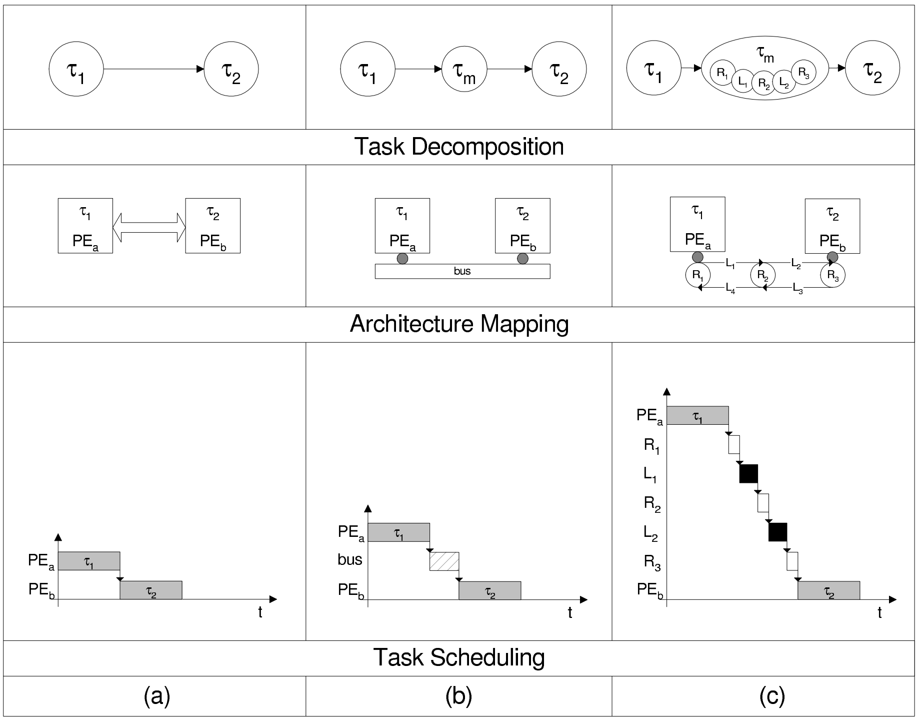


Figure 1.1. Communication modelling.

Example 1: Consider three sample communication network topologies (see Figure 1.1): (a) fully-interconnected or point-to-point, (b) bus, and (c) mesh. When modelling a fully-interconnected network, illustrated in Figure 1.1(a), at the system level, the inter-task communication can be assumed to be negligible. This type of network has no multi-hop or sharing capability but it does allow concurrency. The reuse and scalability potential of such a network is limited. In the case of a bus network, as illustrated in Figure 1.1(b), the inter-task communication cannot be neglected. Therefore, the task graph of an application can be extended by the insertion of message tasks (τ_m's) which represent the transfer of data between tasks. Such a network is shared but does not allow multi-hop or concurrency. Since only one message transfer can take place at a time, the bus quickly becomes a critical resource.

As the number of processors is increased, a careful selection of task and communication scheduling policies is required when mapping tasks onto processors and deciding upon the system implementation either in software or in hardware. The fully-interconnected and the bus-based network models can provide the best-case and the worst-case performance limits, respectively, while carrying out an initial estimate of the communication requirements of an application. However, a network-on-chip solution requires more sophisticated modelling and design techniques in order to handle muti-hop communication where the message transfer time may depend upon the traffic and the actual routing path through the network. In Figure 1.1(c), communication in a bi-directional mesh is shown. It has all the requisite network properties listed above and, therefore, it allows successful inter-task communication.

2.2 Network Boundary Issues

For modelling purposes, it is important to define a precise boundary between the functionality of the NoC and the PE's. For example, in Figure 1.1(b), if a PE is responsible for performing communication tasks as well, then there is an overlap between the tasks running on the PE's and the NoC. Similarly, in Figure 1.1(c), it is possible to design the nodes (R_1, R_2, and R_3) to buffer messages before transmission. On the other hand, referring back to Figure 1.1(b), if a provision exists for the PE's to "dump" their messages for communication to some temporary location without actually performing the communication tasks, it effectively decouples computation from communication. This temporary location is called a network interface (NI).

The OSI [1] layered communication model is a convenient way to explain the complexity inherent in the inter-task communication. These input and output buffers are jointly maintained by the transport layer and the network layer protocols. The transport layer interfaces with

6

the tasks and provides them with the message transport services. The network layer interfaces with the medium-access control layer and provides network access and message transmission services to the transport layer. When requested to send a message by a local task, the source transport layer places the message in the output buffer. From there, each outgoing message is delivered to the network under the control of the source network layer. After the message has traversed the network, the destination network layer places the message in the input buffer and notifies the destination transport layer. The destination transport layer then moves the message to the address space of the destination task and notifies the task of the arrival of the message. The activities of sending a message can be represented by a chain of tasks.

The source and the destination tasks are the predecessor and the successor of this chain of tasks, respectively. At the beginning and the end of the chain of tasks are the source and the destination transport protocol processing tasks. In between them, each task that accesses the network or transmits the message becomes ready for execution after its immediate predecessor completes, possibly with some delay introduced by the execution synchronization protocol used. In short, for modelling purposes, the various options for implementing these layers determine the NoC boundary [2]. In our model, these options boil down to two types of transitions in the task model:

- **Overlapped Model:** In this model, a PE not only initiates communication but also contributes to set it up. Thus, the computation capability of a PE is utilized to carry out the communication processing functions like message encapsulation, header creation, encoding, etc. In the OSI context, the PE implements all the layers upto the physical layer to handle communication. As a result, an inter-task overlap occurs between the task generating a communication request on a PE and the communication task complying to such a request.

- **Triggered Model:** In this model, a PE only triggers a communication event but communication is actually handled by the NI. This frees up the computation capability of the PE for other tasks scheduled on it. An NI takes data from a PE, encapsulates it, and ensures its successful transmission through the NoC. In the OSI context, the PE implements all the layers upto the transport layer, while the NI implements the rest of the layers below it, including the transport layer, to handle communication. As a result, there is no overlap of PE tasks demanding communication services with the communication task performing communication.

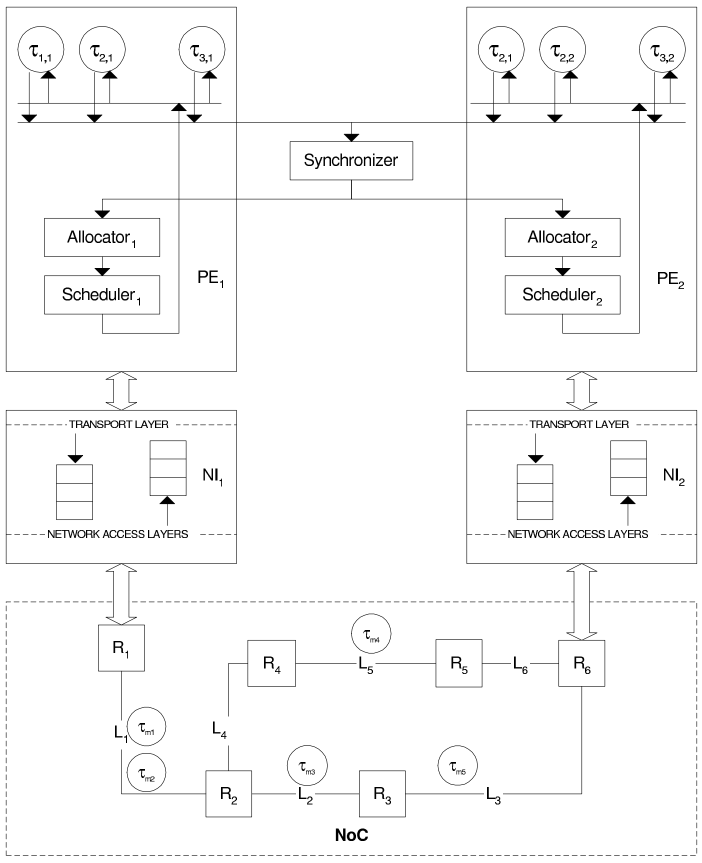


Figure 1.2. Network Interface.

From the above discussion, it is obvious that the choice of a NI can have global consequences on system-level scheduling (especially for large message sizes). When deciding the network boundary, it is important to break down the inter-task communication process into a sub-process that initiates communication and a sub-process that performs communication. Figure 1.2 illustrates this concept. For the overlapped model, the NI will be part of PE while for the triggered model, the NI is a separate entity as shown in the figure. The PE and the NoC models shown in this figure will be discussed later in the Section 4 of this chapter. The implementation of these sub-processes, whether it is within the perimeter of PE silicon or NoC silicon, only impacts power and area considerations but not task scheduling. A NI introduces additional complexity of resource management to the scheduling problem. Its implementation can

8

range from just a set of wires, in a simple case, to dedicated network processing capability with memory in a more complex case.

2.3 NoC Usage

Application design for embedded systems is a special challenge because embedded systems do not simply perform computations; they also interact with their environment. Thus the embedded systems must interface with and react to real processes. To achieve this goal, system designers must juggle real-time constraints, concurrency, and heterogeneity. The future SoC designers are faced with two major design challenges:

Platform Design: Finding good solution templates for the architecture platform under the constraints and characteristics set by the semiconductor technology on one side, and the application domain in question on the other side.

Platform-based Design: Given a platform architecture, how should it be configured (or instantiated) and how should the application, in terms of a description of multiple, concurrent processes, be mapped onto the platform while optimizing a number of design metrics, such as, performance, power consumption, memory utilization, and size, reusability, and flexibility.

Platform-based design is an efficient way to design complex system-on-chip products. It follows a meet-in-the-middle approach, starting with a functional system specification and a predesigned SoC Platform. Performance estimation models can help analyze different mappings between the functional modules of the application and the platform components. During these iterations, designers can try different platform customizations and functional optimizations.

Example 2: As the number of components in architecture platforms increases, the type of on-chip interconnection and communication schemes for processing elements, memories, and peripherals becomes important.

Figure 1.3 illustrates a simple example explaining how NoC modeling can facilitate design-space exploration. In this example, we consider a system that can use three sample network topologies (1-D torus, 1-D mesh, and bus), each interconnecting three processing elements $\{PE_a$, PE_b, $PE_c\}$ executing five tasks $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$. The initial mappings are: $\{\tau_1, \tau_2\} \mapsto PE_a$, $\{\tau_3\} \mapsto PE_b$, $\{\tau_4, \tau_5\} \mapsto PE_c$ (where, $\mapsto$ means 'maps to'), and having dependencies: $\{\tau_1, \tau_4\} \prec \tau_3$, $\tau_2 \prec \tau_5$ (where, $\prec$ means 'precedes'). Each task dependency is manifested by the insertion

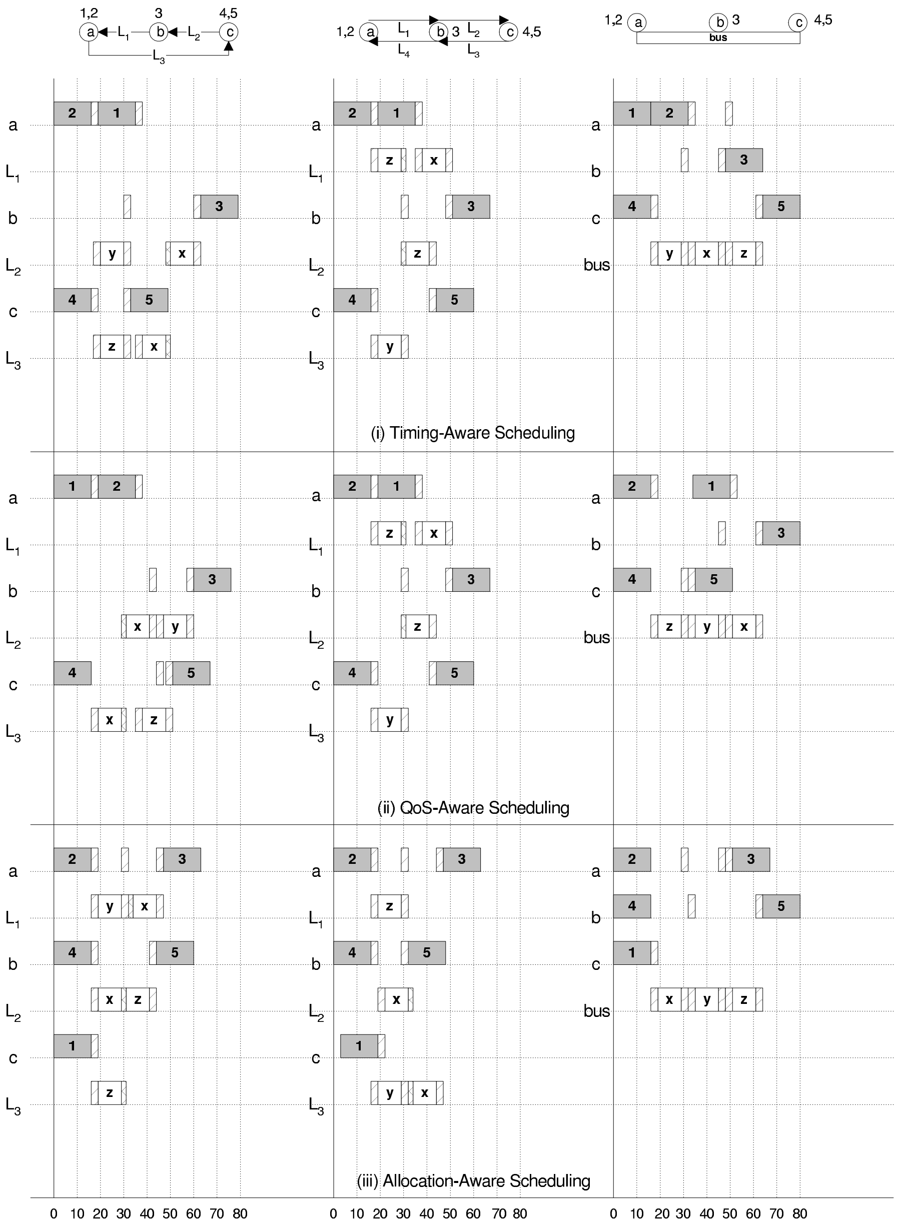


Figure 1.3. System-level modelling illustration

10

of a communication task between the inter-dependent task pairs. Thus $\tau_1 \prec \tau_{mx} \prec \tau_3$, $\tau_4 \prec \tau_{my} \prec \tau_3$, and $\tau_2 \prec \tau_{mz} \prec \tau_5$ (where τ_{mx}, τ_{my}, and τ_{mz} represent message tasks).

For simplicity, all the tasks have been assigned the same period (T = 100), execution time (BCET[1] = WCET[2] = 15), and deadline (d = 100). The time is measured in absolute time units. The tasks are mapped in such a way that none of them misses its deadline. For the purpose of this example, the overlapped NoC model is used and the system performance criterion is defined to be the time when all the tasks finish their execution, i.e., the earlier all the tasks finish their execution, the better is the system performance. The results of Example 2 are analyzed below.

2.3.1 Timing-Aware Scheduling. In the first row of Figure 1.3, Timing-Aware Scheduling scheme is depicted. In this scheme, the tasks executing on a system using the bus or the torus topologies finish in 80 time units whereas if the same system is using the mesh topology, the tasks take 65 time units to finish. The link contention is resolved randomly. As a bus comprises a single link, so the NoC resource allocator/arbiter does not have much freedom in its allocation. Therefore, the communication tasks are scheduled sequentially. On the other hand, the torus and the mesh networks have multiple ways to allocate their resources and schedule message tasks. The full potential of the torus network to handle concurrent communication is not exposed very well in this example but for the mesh network, it can be seen that its performance is better than the other two networks by over 10 time units.

From the point of view of link utilization[3], the torus and the mesh networks only use two out of three links and three out of four links respectively. So a possible network optimization can occur for these networks. On the other hand, if the tasks, τ_1 and τ_4, are scheduled concurrently on a torus network, there is a contention on link L_1. So the network has to be optimized accordingly to meet the timing requirements.

2.3.2 QoS-Aware Scheduling. In the second row of Figure 1.3, QoS-Aware Scheduling scheme is shown. In this scheme, the traffic originating from PE_a is assigned a higher priority and, therefore, it is allocated the contentious link whenever a link contention arises. In the case of a mesh network, there is no link contention, so it has no

[1]BCET is the Best-Case Execution Time
[2]WCET is the Worst-Case Execution Time
[3]Link Utilization is defined as the aggregation of the number of links occupied in the smallest time unit.

effect on system performance. However, in the case of a torus network, such a message scheduling scheme results in a performance gain of 5 time units. For a complex network with lots of nodes and links, such a performance gain can be significant (both for the torus and the mesh topologies). The bus architecture, on the other hand, would result in a communication bottleneck.

2.3.3 Task Allocation-Aware Scheduling. In the third row of Figure 1.3, we illustrate the effects of altering the allocation of tasks to the processing elements while selecting different networks. The new task allocation is: $\{\tau_2, \tau_3\} \mapsto PE_a$, $\{\tau_4, \tau_5\} \mapsto PE_b$, and $\{\tau_1\} \mapsto PE_c$ (the task dependencies are kept the same). Compared to the bus, the system performance advantage is significant with the segmented (the torus and the mesh) networks. The reasons for the poor system performance with the bus are the same as described above. From the point of view of link utilization, it is now higher with the torus and the mesh networks. Most of the links, though not all, are now used simultaneously without any contention.

2.4 Discussion

As mentioned earlier, the timing-, QoS-, and allocation-aware scheduling analysis of Figure 1.3 is based entirely on the finish-deadline of the task mapped to the allocated resource, where the resource is either a PE or a network resource, such as a node or a link. Additional quantifications such as memory, area and power are also possible to incorporate into the model. For example, if the power consumed per communicated-bit is assumed equal, then the comparison of the power profile of the allocation-aware 1-D mesh with the timing-aware bus will show a power spike within the time duration of 20 to 40 time units for the 1-D mesh network while exhibiting a stable power profile for the bus. Even within the 1-D torus network, the communication power profile down the column (in Figure 1.3) shows an aggregation which may be disadvantageous although link utilization has improved.

The main aim of this example exercise is to show how various options for performing design trade offs like resource types, resource allocation, task/message scheduling, etc. can be explored via the proposed NoC framework.

3. Framework for NoC Modelling

For the purpose of abstracting a system-level model, an embedded, real-time application can be represented as a collection of multiple, con-

12

current execution threads that are modelled as a set of dependent *tasks* under certain precedence and resource constraints which have to be executed on a number of programmable processors under the control of one or more RTOS(s).

A system-level model can, thus, comprise three types of basic components: tasks, RTOS services, and communication network, where the communication network is meant to provide communication services between the other system components.

The RTOS services can be further decomposed into independent modules that represent different basic RTOS services like task *scheduling*, resource *allocation*, and execution *synchronization*, where a scheduler models a real-time scheduling algorithm; a synchronizer models the dependencies among the tasks and, hence, both the intra- and inter-processor communications; and an allocator models the mechanism of resource sharing among the tasks. A modeling framework provides the mechanism by which the various components comprising a model interact [15]. It is a set of constraints on the components and their composition semantics and, therefore, it defines a model of computation which governs the interaction of components [7]. Using a modeling framework, a system-level model can be composed from the basic components in such a way that the nature of services provided by any of the components can be altered in a simple and straightforward manner independent of the other components. In our discussion of the system-level modelling framework so far, we have not incorporated the effects of a NoC but we are going to consider that aspect now.

In order to communicate, the tasks executing on different processing elements generate *messages* and submit them to the *communication network* for transmission. The real-time, inter-processing element *traffic* consists of messages that are continuously generated by their sources and delivered to their respective destinations. Such traffic includes *periodic* and *sporadic* messages that require some degree of guarantee for on-time delivery. In addition, there are also *aperiodic* messages. Aperiodic messages have soft timing constraints and expect the system to deliver them on a best-effort basis.

Periodic Messages are generated and consumed by periodic tasks, and their characteristics are similar to the characteristics of their respective source tasks. Therefore, the transmission of a periodic message can be represented by a periodic message task. By a similar argument, the transmission of an aperiodic message can be represented by an aperiodic task. Although an aperiodic message task, like an aperiodic task, does not have a relative deadline, it is still desirable to keep the average delay suffered by aperiodic message tasks to be as small as possible.

Sporadic message tasks have widely varying lengths and/or inter-arrival times. In general, sporadic messages represent burst communication and a sporadic message can be characterized in the same way as a sporadic task [16].

For the purpose of efficient transmission through the communication network, messages are fragmented into smaller-sized segments. The unit of data transmission at the network level is called a *packet*. Therefore, a message can be considered as a set of packets, where the packet size is bounded. Packet transmission is non-preemtive. Thus, a communication network can be modelled as a *communication processor* on which *message transmission tasks* are scheduled nonpreemptively on a fixed-priority basis. In this way, the effect of the inter-processing element communication is modelled automatically by the response times of the message transmission tasks on the network [16].

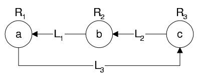
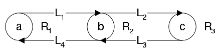
Modelling an on-chip communication network as a communication processor can reflect the demands on the network services. As a communication event within a network is modeled as a message task (τ_m) executing on the communication processor, therefore, when one PE intends to communicate with another PE, a τ_m is fired on the communication processor. Each τ_m represents communication between a set of two fixed, predetermined PE's only. Since a NoC supports concurrent communication, τ_m's need to be synchronized, allocated resources and scheduled accordingly. This reflects the property of the underlying NoC implementation, where the *NoC Allocator* reflects the topology and the *NoC Scheduler* reflects the protocol. Additional flow-control aspects, such as deadlock-avoidance, session-maintenance, acknowledge-based completion, etc. can also be implemented. Though the handling of those aspects either by the NoC Allocator or the NoC Scheduler depends upon the specific NoC architecture.

A resource database which is unique to each NoC implementation, contains information about all its resources. In a segmented network, these resources are laid-out as two-dimensional interconnects and comprise nodes (routers) and links. The algorithm for NoC allocation and scheduling map an τ_m onto the available network resources. The main focus of our discussion here is the networks which allow parallel communication, such as the segmented networks.

3.1 NoC Allocator

In a system-level NoC model, the role of the NoC Allocator is to translate the path requirements of the τ_m in terms of the resource requirements such as link bandwidth, storage buffers, etc. It has to minimize conflicts over the network resources. The links and the nodes in the communication path can be set aside dynamically (i.e., for the requested time-slot) in the resource database. If the resource reservation process is successful, the τ_m has to be queued for scheduling. When an τ_m releases a resource after usage, the resource is free to be assigned to another τ_m. However, if there is a contention over a resource, then resource arbitration has to occur. The NoC allocation patterns for two sample networks are shown in Table 1.1. The resource arbitration can be based on the underlying network implementation and will be discussed further shortly.

Table 1.1. A sample reservation for two sample networks.

Message Task	**Path**					
		1-D Torus			**1-D Mesh**	
		Resource Allocation	**Scheduling Needs**		**Resource Allocation**	**Scheduling Needs**
			Small Message Size	**Large Message Size**		**Small or Large Message Size**
τ_{mx}	a→b	L_1	Immediate	Preemptive	L_1	Immediate
τ_{my}	c→b	L_3, R_1, L_1	Immediate	Immediate	L_3	Immediate

3.2 NoC Scheduler

Message Scheduling constitutes a basic function of any distributed real-time system. The scheduling of real-time messages aims to allocate the medium shared between several nodes in such a way that the time constraints of the messages are respected. As outlined above, not all of the messages generated in a distributed real-time application are critical from the point of view of time. Thus, according to the time constraints associated with the messages, the following message scheduling strategies can be applied:

- *Guaranteed Strategy*: According to this strategy, a message is always guaranteed to be delivered within its deadline requirements. This strategy is generally reserved for messages with critical timing constraints.

- *Stochastic Strategy*: In stochastic message scheduling strategy, the time constraints of messages are met in a best-effort fashion at a pre-computed probability. This strategy is used for messages with soft timing constraints.

In a distributed real-time system, the above strategies can cohabit, to be able to meet various communication requirements, according to the constraints and the nature of the communicating tasks.

As messages have similar constraints as tasks (mainly deadlines), the scheduling of real-time messages uses techniques similar to those used in the scheduling of tasks but with a difference. Whereas, tasks, in general, can accept preemption without corrupting the consistency of the end result, the transmission of a message does not admit preemption. If the transmission of a message starts, all the bits of the message must be transmitted, otherwise, the transmission fails [5].

The NoC Scheduler has to execute the τ_m according to the particular network service requirements. It has to minimize resource occupancy while making sure that the τ_m's are delivered within the specified timing bounds. In a network, resource occupation is dictated by the message size. The concept is better illustrated using the example in Table 1.1, where scheduling needs for the same two sample networks are shown. For a mesh there is no conflict. The τ_m's get the required resources scheduled "immediately". But in the case of the torus, it may experience resource allocation conflict for link L_1. Here, in the event of a small message size, where τ_{mx} is finished before τ_{my} asks for L_1, there is no scheduling problem. The resources can be "immediately" assigned to the τ_m's. But in the case of a large message size were τ_{mx} is still running when τ_{my} asks for the link L_1, resource contention occurs. Thus the resource L_1 is required to be scheduled "preemptively". Preemptively, here, implies the degree of contention resolution.

For example, let us consider this from the network-designer's and the system-designer's view point. At the network-level, seeing the resource conflict as a network problem, the network designer may over-design link L_1 by providing excess bandwidth or introduce processing overhead, such as TDM-based message interleaving. These techniques would restore fair servicing for both τ_m's, reducing the degree of contention. However, at the system-level, it may be possible to reschedule the communication event between the PE's (either τ_{mx} or τ_{my}). This opens the possibility of alternate path allocations for the τ_m's or simply stall one of the traffics until the other has finished. System designers may even realize that "large messages" (to the extent where L_1 is contentious) never occur within the given system. This could save potential scheduling/computation overhead in terms of hardware real-estate, power, etc. at router, R_1, and on link, L_1, as was envisioned by the network designer. Thus, when seen from the system-level, a trade-off between the NoC resource allocation and the NoC scheduling would not only com-

16

plement better self utilization, but might give other useful insights for design improvements.

In the following section, we present an implementation of the NoC model in SystemC [19, 9]. This implementation can be viewed as an extension of the implementation an abstract RTOS model described above.

4. NoC Model Implementation

The above-mentioned ideas about forming an abstract system-level NoC model can be validated by implementation in a system-level modelling language such as SystemC. We will briefly describe here the SystemC implementation of the NoC model described above. Further details regarding the implementation can be found in [8] [17].

For the purpose of implementation, tasks are considered to be an abstract representation of the application and, therefore, have been characterized through a set of parameters, such as the worst- and the best-case execution times, context-switching overhead, deadline, period (if it is a periodic task), offset, resource requirements, and precedence relations. A task is modeled as a finite state machine (FSM) that sends the messages: `ready` and `finished`, to the scheduler which, in turn, sends one of the three commands to a task: `run`, `preempt`, and `resume`. In between the schedulers and the tasks, we have the synchronizer and the allocator acting as "logical command filters". By this scheme, each component can handle its relevant data independently of the other. For example, a task can determine when it is ready to run and when it has finished. The scheduler behaves in a reactive manner; scheduling tasks according to the indications received from them. Thus, as many tasks and schedulers can be added as desired. The same is the case with the synchronizer and the allocator models. They can hold the information regarding their services, i.e., which tasks depend on each other or, for the case of the allocator, what resources are needed by a given task.

The NoC model has exactly the same structure as the abstract RTOS model [8] but with some modifications. The message routing scheme implemented in the NoC model is that of fixed routing but the framework has provisions for implementing other routing schemes. The effects of the network interface or the task overlap due to message processing in the PE, as discussed in Section 2, have not been implemented for simplicity but the model can be extended to incorporate any of those effects.

4.1 Message Task

The message task (τ_m) has the same finite state machine (FSM) structure as the task model in the abstract RTOS model with some modifications to take out preemption and introduce resource requirements. The τ_m implementation accepts a number of arguments for its characterization.

- *Message Task ID*: enables the synchronizer and the NoC Scheduler to identify the τ_m sending the message.
- *NoC Scheduler ID*: is meant for the τ_m's to recognize their scheduler for exchanging various control messages.
- *Best Case Transmission Time (BCTT)*: is the lower bound on the transmission latency of a τ_m through the NoC.
- *Worst Case Transmission Time (WCTT)*: is the upper bound on the transmission latency of a τ_m through the NoC.
- *Offset*: is the setup time for a τ_m.
- *Resource ID*: is the ID tag for a resource (link, router, etc.) required by a τ_m.
- *Critical Section Length (CSL)*: the time duration for holding a resource.

The implementation of an τ_m can be viewed as an FSM that manages various counters after sending indications to the NoC Scheduler and the NoC Allocator and upon receiving commands from the NoC Scheduler.

4.2 NoC Allocator

The NoC Allocator manages its resource database upon receiving `request` and `release` indications from the τ_m's. The resources are allocated to the τ_m's dynamically and they are released by the τ_m's immediately after usage. This makes resource management very flexible allowing sharing and concurrency. In this implementation, the resources are served by the NoC Allocator on a 'first-come-first' basis but other allocation policies can be implemented as well. Whenever a requested resource is available, the NoC Allocator sends a `grant` indication to the NoC Scheduler and whenever a requested resource is occupied, there is a resource contention and the NoC Allocator sends a `refuse` indication to the NoC Scheduler for appropriate action.

18

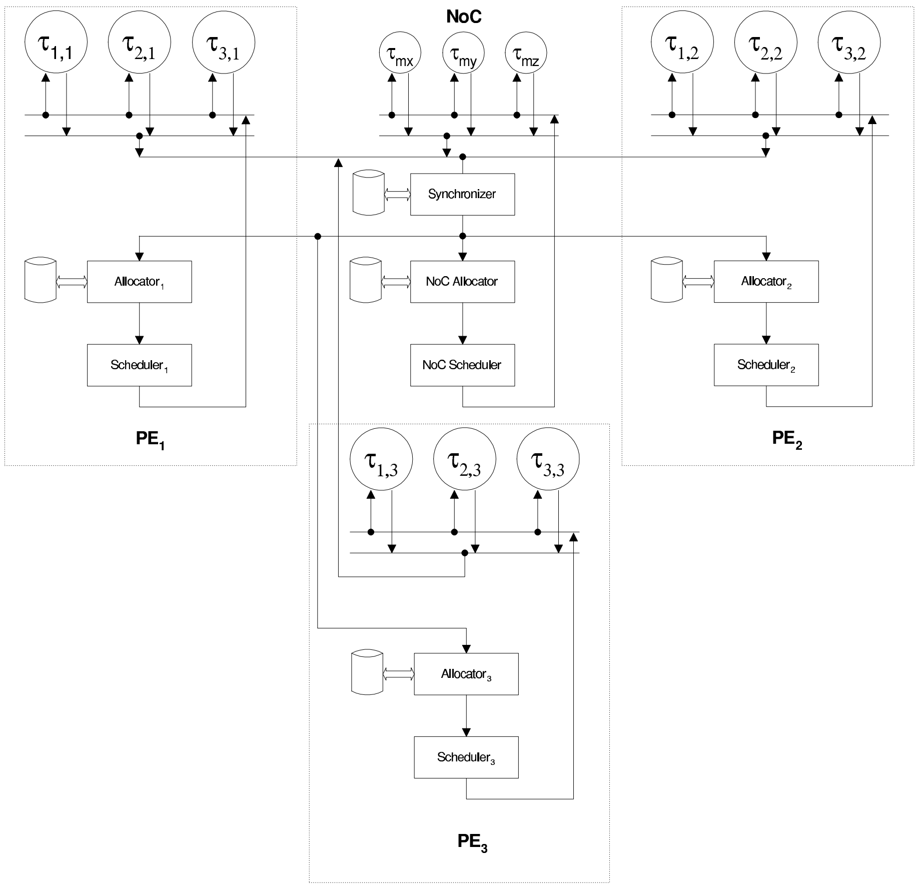


Figure 1.4. The system-level usage of the NoC model with the RTOS model.

4.3 NoC Scheduler

The NoC Scheduler receives the `ready` and `finished` indications from the τ_m's through the Synchronizer and the `grant` and `refuse` indications from the NoC Allocator. It then issues the `run` and `buffer` commands to the τ_m's. Whenever a task running on a PE, is finished and needs to communicate with a task running on another PE, it sends a `finished` indication to the synchronizer which maintains a task dependency database and passes the `ready` indication for the corresponding τ_m to the NoC Scheduler which issues the `run` command to that τ_m.

Whenever there is a resource contention, the NoC Allocator issues a `refuse` indication to the NoC Scheduler which then either terminates the execution of the requesting τ_m (equivalent to a message dropping) or blocks the τ_m from execution (equivalent to message buffering) till the requested resource becomes available again which is indicated by the `grant` indication sent by the NoC Allocator to the NoC Scheduler. The message dropping or buffering decision is taken by the NoC Scheduler according to its underlying network implementation.

5. Simulation Results

The results of our SystemC implementation of the NoC model from Figure 1.4 are presented in Figure 1.6 and Figure 1.7. The sample SoC-NoC setup is as shown in Figure 1.5. Here, the application is assumed to be decomposed into four tasks. Three PE's are selected to execute these tasks. The task mapping is: $\{\tau_1\} \mapsto PE_a$, $\{\tau_4\} \mapsto PE_b$, and $\{\tau_2, \tau_3\} \mapsto PE_c$. τ_2 has a higher priority than τ_3, so it can preempt τ_3 on PE_c. In this example, we look at a simple case where all the tasks are modeled identically with a period of 25 time units (except for the τ_2 with a period of 24 time units due to the priority-assignment scheme in the Rate-Monotonic Scheduling), execution time (both BCET and WCET) of 10 time units, and finish deadline of 22 time units.

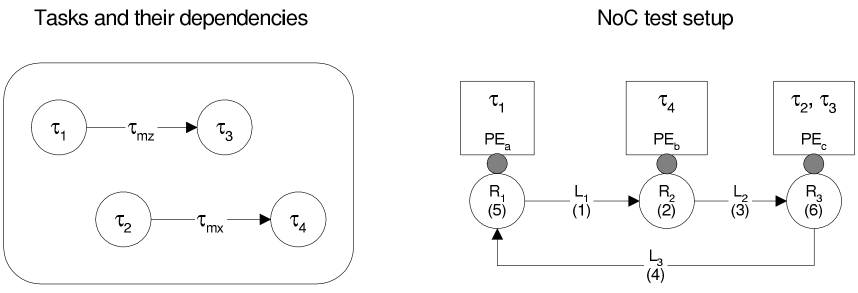
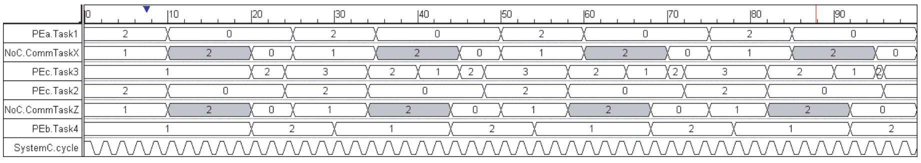


Figure 1.5. System simulation model

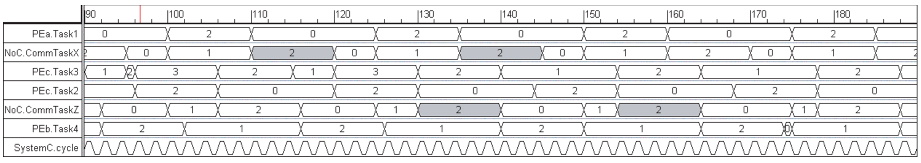
The communications between the tasks are modelled as τ_m's (as described in Section 4) which run on a torus network processor using store-and-forward routing protocol (with infinite buffer at the source and the destination nodes). The message task paths and dependencies are: τ_{mx}, from PE_a to PE_c using L_1, R_2 and L_2, and τ_{mz}, from PE_c to PE_b using L_3, R_1 and L_1. Thus, the link L_1 experiences possible contention. In our test SoC-NoC setup, these resources are tagged by an ID which is given in brackets (in Figure 1.5). We present two cases of interest.

In Figure 1.6(a) modelling of two concurrent communications is shown. As mentioned earlier, there is a link contention between τ_{mx} and τ_{mz} for

20



(a)

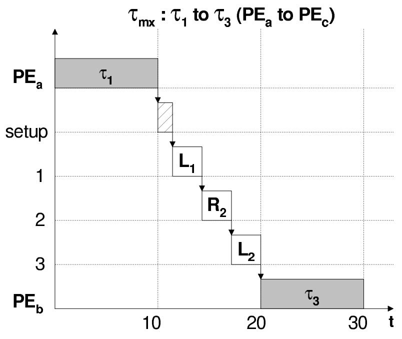


(b)

Figure 1.6. Simulation results for communication events (from 0 to 190 time units). State enumeration: 0=inactive, 1=ready, 2=running, 3=preempted.

```
0  Initializations
10 CommTask X released by the synchronizer
10 CommTask Z released by the synchronizer
11 task x (request resource# 1)-> allocator
11 NoC_allocator (granted)->NoC_scheduler
11 task z (request resource# 4)-> allocator
11 NoC_allocator (granted)-> NoC_scheduler
14 task x (release resource# 1)-> allocator
14 task x (request resource# 2)-> allocator
14 NoC_allocator (granted)-> NoC_scheduler
14 task z (release resource# 4)-> allocator
14 task z (request resource# 5)-> allocator
14 NoC_allocator (granted)-> NoC_scheduler
17 task x (release resource# 2)-> allocator
17 task x (request resource# 3)-> allocator
17 NoC_allocator (granted)-> NoC_scheduler
17 task z (release resource# 5)-> allocator
17 synchronizer (release)-> allocator
17 task z (request resource# 1)-> allocator
17 NoC_allocator (granted)-> NoC_scheduler
20 task x (release resource# 3)-> allocator
20 task x (finished)-> scheduler 2
20 synchronizer (finished)-> allocator
20 NoC_allocator (finished)-> NoC_scheduler
20 task z (release resource# 1)-> allocator
20 task z (finished)-> scheduler 2
20 synchronizer (finished)-> allocator
20 NoC_allocator (finished)-> NoC_scheduler

and so on...
```



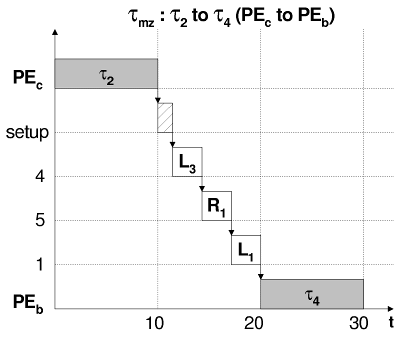


Figure 1.7. NoC allocation and scheduling for the first communication cycle along with the simulation log.

L_1. It is resolved by scheduling L_1 at different times among the τ_m's within the time-slot of 10 to 20 time units (and subsequent time slots). L_1 is used from 11 to 14 time units in τ_{mx} and from 17 to 20 time units in τ_{mz}. Figure 1.7 shows the log file of resource occupancy (Resource# 1, that is, the link L_1). The accompanying plots on the right provide a graphical representation (Note that 1 time unit is lost in network setup during simulation). Thus, our model clearly supports concurrent communication as observed in segmented networks.

Figure 1.6(b) shows the interplay of process modelling and interconnect activity. Consider the signal near the time period of 95 time units. Here, it is clear that τ_3 starts accepting the communication message and is then preempted by τ_2 on PE_c because of its higher priority. Once τ_2 is finished, τ_3 resumes and completes in time before deadline. Now consider the next execution of τ_3. Both τ_2 and τ_3 are in contention. The τ_3 does not even start instead, τ_2 starts on the PE_c. τ_3 here is not able to accept the message communicated to it by τ_1. This brings us to an interesting role of the NoC. In this simulation, we have enabled the routers to be able to buffer messages. Thus τ_{mx} finishes freeing up its resources although τ_2 has yet to begin. The τ_3, when finished, is, thus, able to initiate τ_{mz}, which is when τ_2 resumes.

Consider the case where the same torus network processor is running the wormhole routing (plots not provided). Then, in the preemption case, the τ_{mx} stalls, holding the link L_1. As τ_2 has already preempted τ_3 on PE_c, when it is complete, it would preempt τ_{mz}. But this would not be possible as the L_1 required here is busy in τ_{mx}, thus stalling τ_{mz}. This causes deadlock in the system. As seen earlier, we can resolve it either by introducing buffering in the routers or we have the freedom to choose an alternate network implementation or scheduling strategy. Thus, this example clearly demonstrates the global performance evaluation for co-design when both SoC and NoC are jointly modelled.

6. Summary

In this chapter, we have presented an abstract modeling framework based on SystemC which supports the modeling of multiprocessor-based RTOS's and their interconnection through a NoC. The aim is to provide the system designer of single-chip, real-time embedded systems with a simple modeling and simulation framework in which he/she can experiment with different task mappings, RTOS policies and NoC structures and protocols in order to study the consequences of local decisions on the global system behavior and performance. We have presented how our initial multiprocessor RTOS model has been extended to handle NoCs.

22

So far, our experimental work has been aimed at providing a proof-of-concept as demonstrated in Section 5. We are currently working on extending the NoC model to incorporate issues like, dynamic path routing, packet switching and power profiling. We are also working on a few large real-life examples as well as a schedule viewer based on the output from the monitors which will provide detailed and annotated views of the system behavior such as detailed network usage and power- and memory-profiles.

Acknowledgements

The SystemC-based modelling framework described in this chapter has been supported by SoC-MOBINET(IST-2000-30094), an EU-funded project.

References

[1] L. Benini and G. D. Micheli. Network on Chips: A New SoC Paradigm. *IEEE Computer*, 35(1):70–78, January 2002.

[2] P. Bhojwani and R. Mahapatra. Interfacing Cores with On-chip Packet-Switched Networks. In *IEEE Proceedings on VLSI Design*, pages 382–387, January 2003.

[3] W. Brainbridge and S. Furber. Delay Insensitive System-on-Chip Interconnect using 1-of-4 Data Encoding. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 118–126, 2001.

[4] J. Cong. An Interconnect-Centric Design Flow for Nanometer Technologies. In *International Symposium on VLSI Technology, Systems, and Applications*, pages 54–57, 1999.

[5] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems.* John-Wiley & Sons, 2002.

[6] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan-Kaufmann, 1998. 1st edition.

[7] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[8] M. J. Gonzalez and J. Madsen. Abstract RTOS Modeling in SystemC. In *Proceedings of the 20th IEEE NORCHIP Conference*, pages 43–49, November 2002.

[9] T. Grotker, G. M. S. Liao, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, New York, 2002.

[10] P. Guerrier and A. Greiner. A Generic Architecture for On-Chip Packet-Switched Interconnections. In *Design Automation and Test in Europe, DATE*, pages 250–256, March 2000.

[11] R. Ho and K. W. Mai. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[12] J-M. Daveau, T. B. Ismail, and A. A. Jerraya. Synthesis of System-Level Communication by an Allocation-Based Approach. In *Proceedings of the 8th International Symposium on System Synthesis (ISSS)*, pages 150–155, September 1995.

[13] P. V. Knudsen and J. Madsen. Integrating Communication Protocol Selection with Hardware/Software Codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1077–1095, 1999.

[14] S. Kumar, A. Jantsch, J-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, and A. Hemani. A Network-on-Chip Architecture and Design Methodology. In *IEEE Computer Society Annual Symposium on VLSI*, pages 117–124, April 2002.

[15] E. A. Lee. What's Ahead for Embedded Software? *IEEE Computer*, 33(9):18–26, September 2000.

[16] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[17] J. Madsen, K. Virk, and M. Gonzalez. Abstract RTOS Modelling for Multiprocessor System-on-Chip. In *International Symposium on System-on-Chip*, November 2003.

[18] V. J. Mooney and D. M. Blough. A Hardware-Software Real-Time Operating System Framework for SoC's. *IEEE Design & Test of Computers*, 19(6):44–51, Nov/Dec 2002.

[19] SystemC Workgroup. http://www.systemc.org.

[20] X. Zhu and S. Malik. A Hierarchical Modeling Framework for On-Chip Communication Architectures. In *International Conference on Computer-Aided Design (ICCAD)*, pages 663–670, 2002.

4.2 ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality

ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality

Shankar Mahadevan[†] Michael Storgaard[‡] Jan Madsen[†] Kashif Virk[†]

Informatics and Mathematical Modelling (IMM), Technical University of Denmark (DTU), Richard Petersens Plads 2800 Lyngby, Denmark

E-mail: [†]{sm, jan, virk}@imm.dtu.dk, [‡]{s011934}@student.dtu.dk

Abstract

Designing complex heterogeneous multiprocessor System-on-Chip (MPSoC) requires support for modeling and analysis of the different layers i.e. application, operating system (OS) and platform architecture. This paper presents an abstract system-level modeling framework, called ARTS, to support the MPSoC designers in modeling the different layers and understanding their causalities. While others have developed tools for static analysis and modeled limited correlations (processor-memory or processor-communication), our model captures the impact of dynamic and unpredictable OS behaviour on processor, memory and communication performance. In particular, we focus on analyzing the impact of application mapping on the processor and memory utilization taking the on-chip communication latency into account. A case-study of a real-time multimedia application consisting of 114 tasks on a 6-processor platform for a hand-held terminal shows our frameworks co-exploration capabilities.

1. Introduction

A key pre-requisite in the design of heterogeneous multiprocessor system-on-chip (MPSoC) is an abstract system-level model that enables evaluation options and make critical architectural decisions in advance of a detailed design. The scheduling problem, central to the analysis of the complexity of concurrent MPSoC programs, depends on the way in which the tasks are mapped on the processing elements (PE). This, in turn, is linked with the physical architecture of the computing platforms, i.e. with task execution latency of the PEs, memory constraints of the PEs which limits the number of tasks mapped to a given PE, and the amount of data to be transferred between tasks mapped to different PE, which will influence communication latency and dynamic memory allocation. When scheduling is handled by a real-time operating system (RTOS) and not statically during compile-time, the system analysis becomes particularly challenging.

We propose an abstract system-level modeling framework, called ARTS, which captures the cross-layer dependencies of application software, RTOS, and multiprocessor platform consisting of PEs connected through an on-chip network. In this paper we focus on two issues of importance for analyzing cross-layer dependencies; static and dynamic memory usage and communication latency due to network topology and protocol. We illustrate the capabilities of ARTS for modeling and analyzing heterogeneous MPSoC systems, by exploring a real-time multimedia application consisting of 114 tasks on a 6-processor MPSoC architecture for a hand-held terminal.

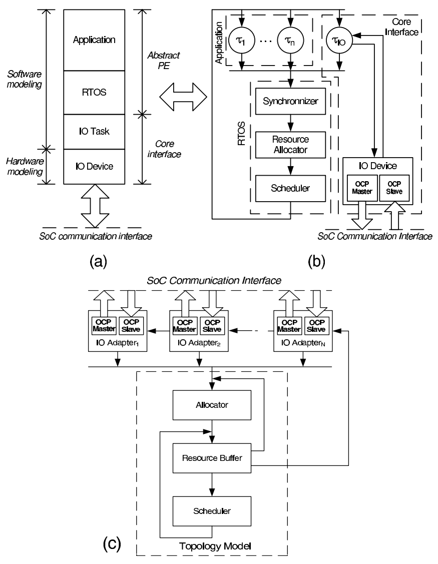


Figure 1. The PE Model: (a) Layer Structure (b) Block Diagram. (c) Network Model.

System-level models for design space exploration of embedded systems targeted for real-time applications has been proposed in [3, 8]. In [3], while supporting extensive RTOS capabilities to evaluate processor utilization, it does not address memory and communication concerns. Tools presented in [1, 4, 9] support processor-memory or processor-communication co-exploration and are important contribution in expanding the scope of design space exploration. In [4], the proposed framework is integrated with a two step co-exploration methodology of static-analysis followed by trace-driven simulation for design evaluation.

We do not propose any specific methodology for design exploration, but provide a flexible framework for designer driven exploration. Using our framework, algorithms such as energy-delay driven memory analysis [2] can be applied prior simulation to group and partition application tasks, or post-simulation traces collected such as those required to identify and tune platform tradeoff [7].

2. The ARTS Modeling Framework

This section presents the PE and application models, followed by details of the communication and memory extensions which are the focus of this paper. The models are implemented in SystemC.

2.1. PE and Application Model

In [5, 6], the PE and application model has shown to be sufficient to model the execution behaviour of a wide range of IP cores. In this paper, the model has been extended with a OCP (v2.0) based *core interface*, Figure 1(a) and (b), for inter-processor communication at RT and transaction levels.

The abstract Real-Time Operation System (RTOS) provides RTOS services, such as task *synchronization*, *allocation* of shared non-preemptive local resource between tasks and task *scheduling* for execution. Protocols supported are Direct Synchronization for the task synchronization, basic priority inheritance for the resource allocation and Rate Monotonic (RM) and Earliest Deadline First (EDF) for the task scheduling. The RTOS manages the tasks and its timing constrains, which is provided by the application model. The application model is based on static task graphs (or dataflow graph). For task modeling, a periodic and sporadic task model is available. Both models supports preemption.

The core interface consists of an IO task and IO device, modeling an IO device driver application and a hardware IO port respectively. The IO task manages the encoding/decoding of data to/from the SoC communication interface. The IO task is released for execution, whenever an inter-processor communication event starts (i.e. transmitting or receiving data). The IO device implements/manages the OCP SoC communication protocol.

2.2. Network model

The network model allows modeling of different communication topologies ranging from a single shared bus to a 1D/2D mesh NoC with minimal path routing. The model is characterized by having an abstract description of the topology but being able to support transmission of real data (e.g. at RTL). Further, it supports multithreaded out-of-order communication. Figure 1(c) shows a block diagram of the model.

The IO adapter model implements and manages the SoC communication protocol and the data conversion between the topology model and the SoC communication interface. When data is received from the SoC communication interface, it issues a process data package message to the topology model. Similar, when data package message is received from the topology model (indicating data has reached the destination node), it initiates a SoC transaction to the particular node.

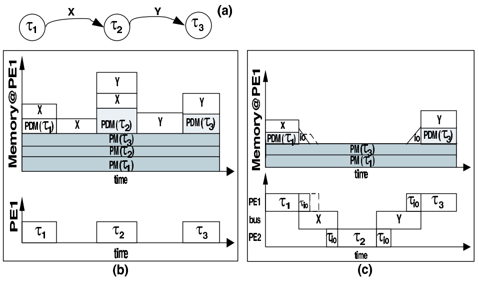


Figure 2. (a) Task graph. Memory profile of PE_1: (b) when all tasks run on PE_1, and (c) when τ_2 is mapped to different PE.

The topology model describes the communication topology. It ensures that a data package message is not released to the destination IO adapter, until a time interval, equal to the communication latency, has expired. The *allocator* models the actual topology and manages the usage of shared communication resource (e.g. links and routers, bus). It assigns resources to the messages as they are received. The *resource buffer* models the resource usage mechanism, by buffering a data package before releasing it back to the allocator again. The interaction between the allocator and the resource buffer models the chain of communication tasks in the communication layer (i.e. the usage of different links and routers for a particular transfer). The *scheduler* models the scheduling of data package messages in case of resource contention. Messages assigned to an occupied resource gets buffered in the scheduler, until the resource becomes available. The current protocol implemented is first-come-first-served.

2.3. Memory Model

The memory model, models both static memory allocation, due to program memory (PM) and dynamic allocation, due to total data memory of the task. The example in Figure 2 illustrates the memory model. It shows the scheduling and resulting memory profile (spilt into static and dynamic memory). The dynamic part is split into private data memory (PDM) needed while executing the task and data memory needed to store data for exchange among tasks.

The total data memory size of the tasks, which is allocated during runtime by the RTOS, is calculated based on precedence constraints. We take a conservative view, i.e. during execution, the task data memory profile is the sum of preceding and succeeding data edges. This is observed for data x of PE_1 (Figure 2(b)) which, is created and dynamically allo-

Applications	Tasks/ Edges	Deadline/ Period (ms)
GSM Encoder	53/80	20
GSM Encoder	34/55	20
MP3 Decoder	16/15	25
JPEG Encoder	6/5	500
JPEG Decoder	5/4	250

(a)

IP Cores	Frequency (MHz)
GPP0	25
GPP1	10
GPP2	6.6
FPGA	2.5
ASIC	2.5

(b)

Table 1. Application and IP Characteristics.

	PE#0	PE#1	PE#2	PE#3	PE#4	PE#5
Arch1: CT = 57006 μs, BC= 71						
IP	GPP0	GPP0	ASIC	GPP0	GPP0	ASIC
OS	RM	RM	-	RM	RM	-
Tasks	18	19	19	23	19	16
PEU (%)	100	59	47	23	61	24
Arch2: CT = 57568 μs, BC= 79						
IP	ASIC	FPGA	ASIC	FPGA	GPP0	ASIC
OS	-	RM	-	RM	RM	-
Tasks	18	19	19	23	19	16
PEU (%)	40	60	47	26	61	24
Arch3: CT = 58271 μs, BC= 70						
IP	GPP0	GPP0	ASIC	GPP0	GPP0	ASIC
OS	EDF	EDF	-	EDF	EDF	-
Tasks	18	19	19	23	19	16
PEU (%)	100	59	47	23	61	24
Arch4: CT = 58207 μs (Deadline Missed), BC= 97						
IP	GPP0	GPP0	ASIC	GPP0	GPP0	ASIC
OS	RM	RM	-	RM	RM	-
Tasks	19	9	24	24	15	23
PEU (%)	100	52	56	58	19	34

Table 2. Case Study Platform Architectures.

cated for the whole duration of task τ_1. As it has to be used by task τ_2, we have to keep the memory allocated until τ_2 has completed. After execution, only the succeeding data is saved, until the time where it is read by the succeeding task, or transferred to the NoC, after which it is deallocated.

Figure 2(c), shows a scenario with NoC transfer, requiring x and y to be transferred over the bus. τ_{io} emulates the device driver for the PEs and, dynamically allocates (y in PE_2) and deallocated (x in PE_1) the needed memory. If the IO task has to stall, i.e. due to bus congestions, the memory profile of the IO task will be a step function as illustrated with dashed lines in Figure 2(c). As the IO task is handled by the PE, any stall will result in an increased latency. Depending on OS scheduling, the time slot when data memory is initialized and deallocated has direct impact on network congestion.

3. Case Study

To illustrate the potential of our framework, we look at an embedded subsystem that executes GSM, JPEG and MP3 applications (Table 1(a)) - in all 114 tasks. Based on the PE choice of GPP, ASIC or FPGA (Table 1(b)); the tasks have different execution properties i.e. best- and worst-case task execution times. Further we can apply, RM or EDF scheduling to the RTOS. We experiment with different platforms, task partitioning and OS choice on a 6 PEs platform connected via a bus. Even for this simple platform, there are in all 15625 (5 IP cores characteristics applied to 6 PEs)

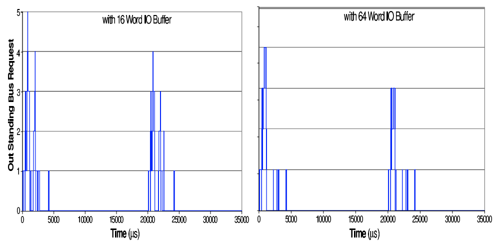


Figure 3. Bus Contention in Arch2.

possible architectural combinations, for a given partition and OS. Co-exploration in this multi-dimensional space, presents many suitable platform architectures and complex trade-off scenarios.

Table 2[1] shows four platform architecture instances, labeled from **Arch1** to **Arch4** from the co-exploration space. The changing characteristics within these platforms is either, IP cores, OS or task partitions. The Completion Time (CT) and the bus contention (BC) is also presented. Using the ARTS framework, we investigated these platform architectures in the context of modeling and understanding the causality between their system properties.

The correlation at $t = 0$ and $t = 20000\,\mu$s in BC and memory profile (Figure 3 and 4), is due to the GSM applications, which have a period of 20ms (Table 1(a)). However, the correlation is not identical and it depends on the OS applied to the platform architectures, and the task mapping. Following is the discussion of the system properties in additional detail.

PE Exploration: Arch1 and **Arch2**, which use identical interconnect and OS in their platforms, present interesting trade-off of performance *vs* flexibility. **Arch1**, with four GPP0s, provides greater programming flexibility (software), while **Arch2**, with two FPGAs, provides greater flexibility in configurable hardware. The difference in CT and BC, among the architectures, is small (Table 2), however the PE utilization (PEU) of **Arch2** is well balanced among the IPs, compare to **Arch1**. This is due to the presence of ASIC which brings added performance.

OS Exploration: For a given platform architecture, a change of OS on one PE may have non-local consequences on other system components. This is due to management and scheduling of task executions by the RTOS, which in turn influences the causality, for example with bus access and memory spread. In **Arch1** and **Arch3**, the switch from RM to EDF, albeit presents limited effect on PEU or CT or BC, it does impact the peak bus occupation (3 in **Arc3** oppose to 5 in **Arch3**). In this case it is favourable to use EDF, since majority of bus contention in earlier architectures where due to conflict between GSM and JPEG, where JPEG transfers large streams of data over the bus, blocking GSM. Due to

[1]The simulation time with complete result logging (PE Utilization, bus contention and memory profiling), for one platform architecture was 0.15sec on Intel Pentium IV® 1.99 GHz with 512 MB of RAM.

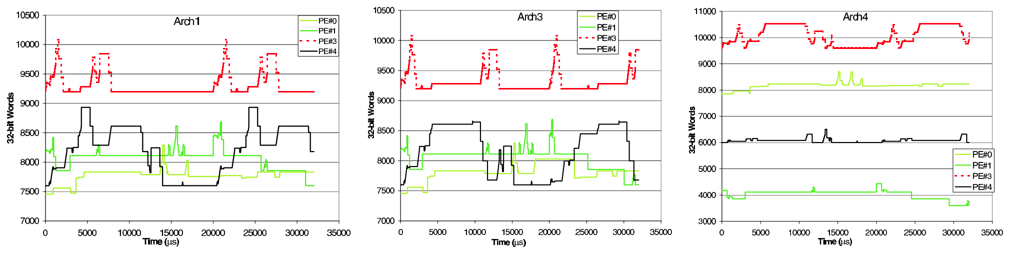


Figure 4. Memory Profile for GPP and FPGA PEs.

EDF, the conflicting JPEG task is swapped with local GSM task, which has higher priority due to its early deadline, there by reducing bus contention. The impact on memory spread can be seen in Figure 4.

Communication Exploration: Communication exploration can be undertaken at different granularity. For example at architectural level by using bus or multi-hop on-chip network topology model (Section 2.2), or at component level by changing the buffer size in the IO adapter. We modeled **Arch2** with IO buffer of size 16 words and then with 64 words, as seen in (Figure 3), which reduced the bus contention. The idea behind this experiment, is to show that even relatively minor tuning within one of the system components, could provide significant system-level gains, without needing deployment of "superior" alternatives, whose impact has not yet been ascertained.

Memory Exploration: The goal of updating the task partition, from that presented in **Arch1** to **Arch4** (Figure 4), was to reduce the peak memory usage of **Arch1**. However, the resulting architecture **Arch4** causes the MP3 to its deadline. It is because, the two task from concurrent branch of the MP3 Decoder are mapped on to the same PE. Interestingly, the partition shows a more balanced PEU, low bus contention and better peak-to-average memory usage, and thus may not be discarded lightly. Mapping the conflicting MP3 task to an alternate PE could potentially bring higher benefits compare to other architectures discussed so far.

The above explorations, shows that choosing an optimum platform architecture of a cross-layered complex design involves studying a large set of viable solution space.

4. Conclusions

The ARTS is a simulation-based framework for single-chip designers to model and explore complex MPSoC designs. In this paper, we have presented valuable extensions to this model by introduction of the communication model and a memory model. The versatility of quasi-static based application models, along with runtime independent execution model, combined with RTOS and communication platform, enables the ARTS framework to develop and explore a broad class of designs. This has been demonstrated in a co-exploration case study for multimedia applications typical in the hand-held device. We have shown various capability and features of our framework which allow selecting and tuning platform exploration under given system constraints. In future, we will extend the model to include dynamic power and area analysis, as additional parameters for trade-off analysis during MPSoC exploration.

References

[1] G. Braun, A. Wieferin, and A. Nohl. Processor/Memory Co-Exploration on multiple abstraction levels. In *Proceedings of Design, Automation and Testing in Europe Conference 2003 (DATE03)*. IEEE Computer Society, March 2003.

[2] W. Fornaciari, D. Sciuto, C. Silvano, and V. Zaccaria. Fast system-level exploration of memory architectures driven by energy-delay metrics. In *ISCAS*, 2001.

[3] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada. RTOS-Centric hardware/software cosimulator for embedded system design. In *Second IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES ISSS)*, September 2004.

[4] S. Kim, C. Im, and S. Ha. Efficient exploration of On-Chip bus architectures and memory allocation. In *Second IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES ISSS)*, September 2004.

[5] J. Madsen, S. Mahadevan, and K. Virk. Network-centric, system-level model for multiprocessor system-on-chip simulation. In J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, editors, *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 13, pages –. Kluwer, 2004.

[6] J. Madsen, S. Mahadevan, K. Virk, and M. Gonzalez. Network-on-chip modeling for system-level multiprocessor simulation. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS03)*, pages 82–92, 2003.

[7] A. Maxiaguine, Y. Zhu, S. Chakraborty, and W.-F. Wong. Tuning SoC platforms for multimedia processing: Identifying limits and tradeoffs. In *Second IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES ISSS)*, September 2004.

[8] R. L. Moigne, O. Pasquier, and J.-P. Calvez. A generic RTOS model for real-time systems simulation with systemc. In *Proceedings of the conference on Design, automation and test in Europe*, page 30082. IEEE Computer Society, 2004.

[9] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl. A system level processor/communication co-exploration methodology for multiprocessor system-on-chip platforms. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'04)*, page 21256. IEEE Computer Society, 2004.

Chapter 5

The RIPE Modeling Environment

This chapter consists of the following papers.

#5: Shankar Mahadevan, Federico Angiolini, Michael Storgaard, Rasmus G. Olsen, Jens Sparsø and Jan Madsen. "A Network Traffic Generator Model for Fast Network-on-Chip Simulation." *In Proceedings of Design, Automation and Testing in Europe Conference (DATE), Munich Germany.* IEEE, Mar. 2005: 780-785.

#6: Federico Angiolini, Shankar Mahadevan, Jan Madsen, Luca Benini and Jens Sparsø. "Realistically Rendering SoC Traffic Patterns with Interrupt Awareness." *IFIP Very Large Scale Integration Systems and their Designs Conference (VLSI-SoC), Perth Australia.* IEEE, Oct. 2005: 211-216.

#7: Federico Angiolini, Shankar Mahadevan, Jens Sparsø, Luca Benini and Jan Madsen. "A Reactive IP Emulator for Multi-Processor System-on-Chip Exploration." *Submitted for Journal Publication.*

From this group only Paper #7 has been presented as it is a comprehensive extension of concepts presented in Paper #5 and #6 with new implementation and case studies. We refer the interested readers to Appendix 7 and 8 for the full text of Paper #5 and #6. With regards to nomenclature, the RIPE framework in Paper #5 and #6 is referred to as simply ‘traffic generator’ or ‘reactive traffic generator’.

A Reactive IP Emulator for Multi-Processor System-on-Chip Exploration

Shankar Mahadevan, *Student Member, IEEE,*, Federico Angiolini,
Jens Sparsø, Luca Benini, *Senior Member, IEEE,* and Jan Madsen

Abstract

The design of Multi-Processor Systems-on-Chip (MPSoCs) emphasizes Intellectual Property (IP) based communication-centric approaches. Therefore, for the optimization of the MPSoC interconnect, the designer must develop traffic models that realistically capture the application behaviour as executing on the IP core. In this paper, we introduce a Reactive Intellectual Property Emulator (RIPE) that enables an effective emulation of the IP core behaviour in multiple (including bit- and cycle-true simulation) environments. The RIPE is built as a multi-threaded abstract instruction set processor and it can generate reactive *traffic modeling. We compare the RIPE models with cycle-true functional simulation of complex application behaviour (task synchronization, multitasking, input/output operations). Our results demonstrate high accuracy and significant speedups. Further, via a case study we show the potential use of the RIPE in a design space exploration context.*

I. Introduction

The primary design paradigm for Multi-Processor Systems-on-Chip (MPSoCs) is the separation of the communication and computation concerns, as this enables Intellectual Property (IP) reuse and shorter design time. However, to test and optimize the independently developed IP cores, and assess their collective performance in a MPSoC platform, one must understand the impact of the communication fabric on the application executing on the platform. Fabrics can span over a huge variety of architectures and topologies, ranging from traditional shared buses up to packet-switching Networks-on-Chip (NoC) [10], [14]. To properly assess functionality and performance, fabric designers build traffic models that test the interconnect under the most realistic application behaviour. To date, these traffic models can be grouped into two primary classes: stochastic models and IP-based models.

Shankar Mahadevan, Jens Sparsø and Jan Madsen are with the Technical University of Denmark; Federico Angiolini and Luca Benini are with the University of Bologna.

The stochastic models provide traffic similar to mathematical distributions such as uniform, Poisson, etc. As seen in [19] and [11], they have been used in the evaluation of different interconnect architectures and features. However, they do not capture the close correlations between different events as would be expected in a realistic MPSoC environment. To make an example, checks for a shared resource done by polling generate different amounts of traffic depending on the relative ordering of accesses to the resource. Thus, the usefulness of stochastic models is restricted to validating the correctness of the implementation of the interconnection backbone, and does not extend to application-specific optimization.

The IP-based models come in several flavours. Some are described at higher abstraction levels, such as Transaction-Level Models (TLM), and some at lower abstractions, such as Cycle-True Models (CTM). The IP-based TLMs used in [18] and [24] are very useful in fast exploration of the system fabric, however the loss of accuracy due to the highly abstracted description of IP models is an impediment to thorough fabric optimization. The detailed IP-based CTMs used in [20] and [16] provide an accurate picture for such an optimization, but are time-consuming to simulate, which disadvantages them for repeated use with alternate fabric architectures and/or feature implementations. The primary drawback, however, is that in both cases, the complete application, the operating system (OS) and the architecture have to be described within the model, in terms of an abstract system behaviour (TLMs) or detailed instruction-set behaviour (CTMs). Since the MPSoC specifications and designs are susceptible to repeated changes, this drawback is costly in terms of modeling and validation time, and may impact time-to-market - which is an ever shrinking constraint.

For the purposes of the interconnect designer, a valuable tool for exploration and optimization needs to meet important criteria, as addressed in [9]. These include

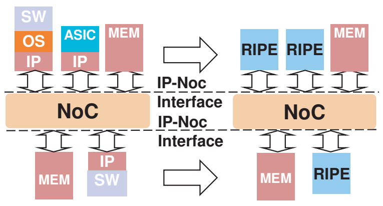


Fig. 1. RIPE as IP Replacement

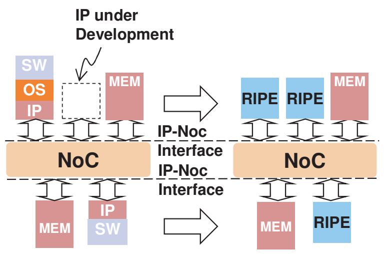


Fig. 2. RIPE as IP Mock-up

repeatability across different fabric architectures, flexibility for easy incorporation of changes in design specifications, and scalability and simulation speedups compared to other models. In this paper, we describe a Reactive IP Emulator (RIPE) which addresses all of the above criteria, and extends further by accurately capturing the communication behaviour that results from the multiple constraints imposed by

- the application,
- the OS,
- the architecture dynamics.

The RIPE enables a versatile and effective emulation of IP behaviour towards the MPSoC interconnect and other IPs in multiple test environments (including bit- and cycle-true). It is built as a multi-threaded instruction set architecture with OCP (Open Core Protocol) 2.0 [4] sockets at its ports. The RIPE allows for easy programming of sequences of communication transactions interleaved with idle waits, and is also capable of sensing feedback from the system. Thereby, it is able to capture the communication-sensitive portion of IP execution behaviour, such as in case of synchronization and interrupt events. The response of the RIPE to such events is governed by the state of the system resources (communication channels and shared memory areas), and mimics the behaviour observed with applications and OS executing on an IP core in a real MPSoC system. This is the essence of *reactiveness*. The main contribution of this work are (i) motivating its need, (ii) deriving its requirements, (iii) validating these requirements, and (iv) demonstrating the impact of RIPE in a co-exploration environment. Our RIPE approach has been proposed for complete realistic emulation of the hardware and the software layers which are stacked in an IP core, and which eventually determine its behaviour at the pinout boundary. This enables a complete decoupling of the simulation of the IP cores from the underlying interconnect fabric. The RIPE can be programmed to reproduce a range of behaviours from polling to interrupt-triggered context switches in presence of an OS. The requirements for such reactive behaviours are explored in detail in subsequent sections.

The RIPE device is designed for interconnect performance tuning and matches multi-threaded application requirements with a truly multi-threaded internal architecture, as will be extensively shown in this paper. Some of the RIPE concepts were originally introduced as a cycle-true OCP-based Reactive Traffic Generator (RTG) in [21] and [6]. The main objective there was to use a device to accurately play prerecorded system traces back. As illustrated in Figure 1, by swapping away IP cores for RIPE blocks in the reference cycle-true system, subsequent design space exploration of the interconnect is allowed to be performed at the same level of accuracy. We expand the scope of the RTG architecture in three ways:

- We support multi-threading in the architecture by maintaining multiple program counters and register files, in place of inflexible branching within a single thread.
- To validate this new architecture, the off-line toolchain used to convert the system traces into RIPE programs has been updated extensively.
- We demonstrate how a RIPE program manually written by the designer can provide insight on the relationship among the behaviour of the whole system and of its components. For example, variable densities of interrupt events can be investigated, or the impact of cache write-back *vs.* write-through policies. As illustrated in Figure 2, this expands the potential of the RIPE to the modeling of design features that are not yet fully implemented.

While still stating the suitability of RIPE for cycle-true environments, we now additionally prove its usefulness as a design space exploration tool when under less strict timing constraints. Additionally, we will show traffic profiling charts that will further motivate and validate the RIPE approach.

To validate our RIPE model and programming paradigm, we test the infrastructure against the bit- and cycle-true detailed MPARM model [20]. MPARM is a homogeneous multiprocessor simulation platform that supports many MPSoC platform configurations and application suites. As part of the validation, we use the MPARM software toolchain to partition and compile different benchmark applications onto the various IPs. These application partitions might conceptually be either routines executing on general purpose microprocessors, dedicated ASIC blocks, DMA engines, or any other device.

The rest of the paper is organized as follows. Section II introduces related work and motiviation, and is followed by a discussion of the requirements for IP emulation in Section III. Section IV details the RIPE model and presents a sample program for modeling application flow. In Section V we discuss the different potential ways to use the RIPE. Section VI describes the mechanism to validate the RIPE. Section VII presents results of validation and simulation for a range of complex benchmarks with and without OS, while Section VIII shows a case study where the RIPE is used as a useful tool for design space exploration. Finally, Section IX provides conclusions.

II. Related Work and Motivation

The use of IP emulation devices such as traffic generators (TGs) is not new, and several approaches and models have been proposed.

In [19], a stochastic TG model is used for fabric exploration, where the IP behavior is statistically represented by means of uniform, Gaussian, or Poisson distributions. A similar approach in [30] uses random and semi-deterministic distributions. The IP model used for NoC optimization in [11] takes into account the nature of MPSoC traffic such as real-time, short-data access, bursty, etc., however the injection rate is governed by statistical methods. In [29], an extra dimension of self-similarity is added to the stochastic model which is argued to assist in precise characterization of multimedia traffic by examining the "similarities" in traffic traces at the macroblock-level. Despite the refinements, the inherent probabilistic nature of the statistical approaches makes it less accurate, as each TG injects traffic in complete isolation from every other. As surveyed in [9], such stochastic models are therefore widely popular for analysis of macro-networks, *e.g.* Internet, that exhibit such behaviour, which is unlikely in MPSoC environment. The simplicity and simulation speed of stochastic models may make them valuable during preliminary stages of interconnect development, but, since the characteristics (functionality and timing) of the IP cores are not captured, such models are unreliable for optimizing communication fabric features.

A modeling technique which adds functional accuracy and causality is Transaction-Level Modeling (TLM), which has been widely used for SoC design [12], [15], [18], [22], [23], [25]. In TLM, Inter-Process Communication (IPC) is realized via channels that implement abstract blocking or non-blocking communication calls. Thus, it is argued that TLM enables higher simulation speed than pin-based interfaces via suppressing "uninteresting" details. In [22], [23], TLM has been used for bus architecture exploration. The communication is modeled as read and write transactions towards the bus. Depending on the required accuracy of the simulation results, timing information such as bus arbitration delay is annotated within the bus model. In [23] an additional layer called "Cycle Count Accurate at Transaction Boundary" (CCATB) is presented. Here, the transactions are issued at the same cycle as that observed in Bus-Cycle-Accurate (BCA) models. Intra-transaction visibility is traded off for a simulation speed gain. An average speedup of 1.55x is reported. While modeling the entire system at TLM, both [22] and [23] present a methodology for preserving accuracy with gain in simulation speed. Such models are efficient in capturing regular communication behaviour, but the fundamental problem of capturing system unpredictability in the presence of interrupts is not addressed.

In [24], a commercial TLM-based reactive workload generation framework is presented that is somewhat similar to our RIPE approach, wherein users can configure traffic patterns for handling synchronization and inter-IP events. Though limited to single-threaded architecture, it also claims to provide primitives for timing-dependent behaviour, wherein the user can tigger actions, which do not depend on application flows but on simulation time. Other commercial efforts also exists, including the OpenVERA [28] language and toolchain that, in addition to modeling concurrency and synchronization, also support verification from abstract level to RTL. Our RIPE, while not supporting some classes of timing-dependent behaviour, supports multi-threading (required for interrupt driven OS-supported context-switch) and traffic generation at multiple levels of abstraction, including cycle- and bit-true environment. More importantly, we have validated our approach with a cycle-true reference system (details provide in Section VI), with near 100% accuracy - a step which the commercial approachs have yet to demostrate, thereby, limiting the confidence in their usage.

In [20] (MPARM) and [16], complete cycle-true MPSoC systems including the full instruction set of the IP cores and the OS are described. This consequently impacts the simulation speed and the scalability of system. Further, the time required to investigate the performance impact of relatively minor changes in systems modeled in such a way is often inflated by the implementation time and then by a relevant simulation time. This hampers the use of such models for the iterative design space exploration process. To overcome the speedup limitation of such simulation-based approaches, an FPGA-based emulation

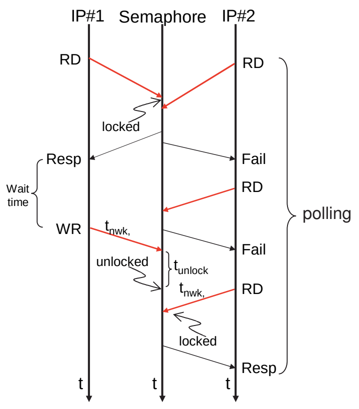


Fig. 3. Typical polling synchronization timeline.

platform has been proposed in [17]. Here, the registers in the traffic generator can be configured to generate different traffic patterns. However, the configurations use either the stochastic model or the trace-driven approach, and the reactiveness capability that is needed for accurate performance optimization is never mentioned. Further, the requirement of a state-of-the-art FPGA board, as used in the emulation, is not alway possible to meet.

Based on the above considerations, given the requirements of accuracy, repeatability, scalability, speed and flexibility set in Section I, no true IP emulator model that spans a range of abstraction levels and usage schemes seems to be available for the MPSoC designers. Our RIPE is meant to address this need and we will suggest a process for its usage at multiple abstractions.

The emulation of the IP core behaviour is not simply a matter of issuing communication transactions, *e.g.* by replaying traces collected from a reference system, an approach that we might call "cloning". Such an approach is clearly inadequate for co-exploration, for example when taking into account the behaviour of a NoC. Here, the variance of network latency results in unpredictable response delays. Such transaction time variability, either due to topological reasons or to congestion, should propagate to subsequent transactions, which would also be delayed in real systems. A simple example of such critical blocking is execution resumption after a cache refill request.

This observation leads to the concept of "time-shifting" behaviour: consecutive transactions are tied to each other, and are issued at times which are a function of the delay elapsed before receiving responses to previous transactions. However, even this model fails when multi-master systems come under scrutiny. The arbitration for resources

in such designs is timing-, and thus architecture-, dependent. Therefore, very different transaction patterns may be observed as a function of the chosen application model and interconnection design. For example, consider Figure 3, where two IP cores (IP#1 and IP#2) attempt to acquire the semaphore lock and, in case the ownership attempt fails, poll the location until success. It is clear that such polling checks for a shared resource will generate different amounts of traffic depending on the relative ordering of accesses to the resource. Time-shifting of traces is not going to be enough to reproduce such behaviour.

The picture is further complicated by the presence of interrupts. While interrupts in themselves do not typically imply an intensive load on the communication architecture, interrupt handling, possibly followed by OS-driven task rescheduling, can severely strain network resources with activity peaks, which in turn may indirectly affect other processors. This event-driven processor reaction must be realistically modeled in order to accordingly optimize the underlying interconnect fabric.

These observations motivate the need for an IP emulation device that is *reactive* to the changes in the system architecture and the application behaviour. It is only by taking both the hardware and software into account that a wide range of synchronization patterns, including OS-based interrupt handling, can be accurately translated into a realistic test load for an interconnect infrastructure under development.

Our RIPE is significantly different from either a purely behavioural encapsulation of application code into a simulation device, which would be in analogy with TLM modeling environment, or detailed instruction-set simulators, which would be closer to deployable hardware environment. However, it spans the behaviour of an IP over this spectrum of environments. The RIPE model we propose is aimed at faithfully creating traffic patterns as they would be generated by an *IP running an application*, not just by the application; this includes *e.g.* accurate modeling of *cache refills* and of latencies between accesses, allowing for cycle-true simulations. We now look at the requirements for modeling such reactiveness.

III. Reactive IP Emulator Requirements

Communication over a shared fabric can be categorized according to several different criteria, such as explicit (for example, data fetching) *vs.* implicit (for example, instruction cache refills), or computation-related (data processing) *vs.* synchronization-related (exchange of signals among processors to keep the status of the system consistent). Another possible criterium is:

I. Processor-initiated communication towards an exclusively owned slave peripheral (*e.g.* accesses to a

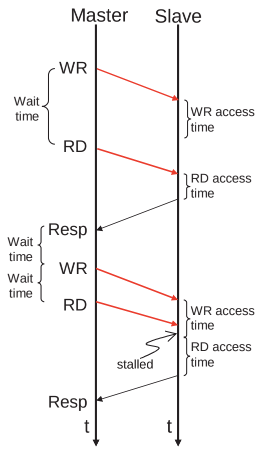


Fig. 4. Communication with Private Memory

private memory),

II. Processor-initiated communication towards an exclusively owned slave peripheral (*e.g.* accesses to a private memory),

III. System-initiated communication towards the processor (*e.g.* interrupts).

Figure 4 shows a simplistic model of Category I traffic, *i.e.* a master accessing a private slave. For such traffic, a trace containing the type and the timestamp of the communication events can be captured at the IP ports, and is subsequently sufficient to emulate the behaviour of the master via non-preemptive sequential communication transactions interleaved with an appropriate amount of idle wait cycles. To elaborate, consider the first two master transactions, a write (`WR`) and a read (`RD`). The time to service the `WR` transaction, which is a posted write, is just the network latency plus the slave `WR` access time. The `RD`, which in our case uses blocking semantics, pays an additional penalty because the response has to make its way back to the master. From the emulation point of view, this pattern is easily recordable: network latency and slave access time are unimportant factors, and the essential point to capture is just the delay between `WR` assertion and `RD` assertion (wait time), and between `RD` response and the following command. This is the essence of the "time-shifting" technique discussed in Section II. In a subsequent simulation with RIPEs replacing IP cores, these delays will be modeled by explicit idle waits in the RIPE, while the network latency will be dependent on the interconnect model under simulation. In the next set of transactions, where a `RD` closely follows a `WR`, the `RD` command reaches the slave before the latter has finished servicing the `WR`, and is thus stalled at the slave interface. This stalling behavior does not need to be explicitly captured in a RIPE model, since, from a processor perspective, it simply appears to be part of the slave response time.

Modeling requirements of Category I traffic can be predicted or inferred given an algorithmic specification. In [29] and [11] such inference is drawn to test the fabric architecture. However, for categories II and III listed above, it is almost impossible to predict traffic requirements without detailed models of the underlying hardware (such as cache replacement policies) and without simultaneously tracking the status of each processor and shared resource of the system. It is due to this requirement that most synthetic TG approaches find a roadblock limiting their applicability, but this is also the area RIPE is focusing onto. So, in describing the requirements for a reactive emulator, we will not consider dataflow issues, but instead the much more challenging synchronization traffic patterns. The capability of handling synchronization and system-initiated traffic is a first requirement for RIPE.

To understand the implications of these different MPSoC traffic categories, we looked at typical application behaviour in MPSoCs. Specifically, the following examples from real-life were considered [31]: multimedia data stream processing, time slicing mechanisms in OS schedulers, and I/O device handling. Depending on the underlying hardware architecture and on the application requirements, a range of synchronization schemes, each leading to different communication patterns, can be observed in these examples. To derive the requirements for our realistic IP emulation, we coded templates of these applications. We leverage the previously introduced (in Section I) MPARM simulation environment to execute these templates. By transforming the information collected during such execution into a RIPE program (a description of the process will be provided in Section VI), we can validate our approach, and compare the performance and accuracy achievable with the RIPE execution engine.

The details of these template programs are described next. The first example is about synchronization patterns typical of multimedia data stream processing, where multiple computational blocks are deployed in a pipelined fashion and communicate according to the producer/consumer paradigm. The latter examples are more strictly related to interrupt handling in presence of an underlying OS which performs context switching.

A. poll

In the simplest synchronization case (**"poll"**), one or more processors competing for a shared resource may poll a semaphore, performing an unpredictable number of accesses prior to lock acquisition and flow resumption.

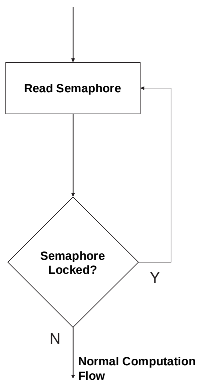


Fig. 5. poll application flow

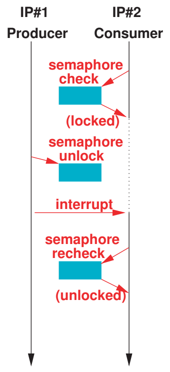


Fig. 6. Typical interrupt synchronization timeline

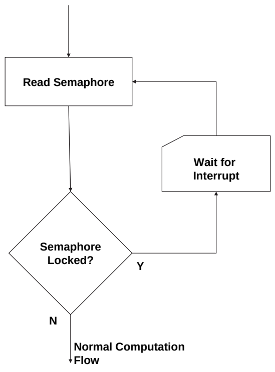


Fig. 7. pipe application flow

For this case, a single task is mapped onto every system core. Tasks are programmed to communicate with each other in a point-to-point producer-consumer fashion; every task acts both as a consumer (for an upstream task) and as a producer (for a downstream task), therefore logical pipelines can be achieved by instantiating multiple cores and tasks. Synchronization is needed in every task to check the availability of input data and of output space before attempting data transfers. To guarantee data integrity, semaphores are provided to assess such availability. For example, the consumer checks a semaphore before accessing producer output. Figure 3 presented earlier illustrates such a scenario. Here, two tasks *i.e.* IP#1 Producer and IP#2 Consumer, attempt to gain access to the same hardware semaphore, which controls an area of shared memory used for data exchange. IP#1 arrives first and locks the resource; the attempt by IP#2 thus fails. If the semaphore is found locked upon the first read, the IP *reacts* with a continuous polling strategy, whereby IP#2 regularly issues read events until eventually the semaphore is found unlocked. Figure 5 represents the application flow of the polling IPs. Since the transactions occur over a shared network fabric, the unlock event (WR) issued by IP#1 and the success of the next request (RD) event by IP#2 are interdependent. Only if the IP#2 RD event is issued at least $t_{\text{nwk,IP\#1}} + t_{\text{unlock,S}} - t_{\text{nwk,IP\#2}}$ after the unlocking by IP#1, then IP#2 will be granted the semaphore and additional polling events will not be required. Therefore, depending on network properties, a variable amount of transactions might be observed at the OCP interfaces of IP#1 and IP#2.

B. pipe

An interrupt-based task synchronization scenario (**“pipe”**) is illustrated in Figure 6. In terms of functionality, this case is similar to **poll**, except for semaphore release handling, which is now augmented by issuing interrupts. If the semaphore is found locked upon the first read, a polling could be performed, at a heavy price in terms of energy consumption, and possibly contributing to the saturation of the system interconnect. Instead, in this scenario, we implemented a mechanism which suspends the consumer task and resumes it only when the producer has data ready. The producer will notify this event by both unlocking the semaphore and sending an interrupt. Figure 7 shows the corresponding application flow within the IPs. Upon interrupt delivery, the consumer re-evaluates the semaphore value for fail-safe operation, and since this time it finds it free, it goes on to process the available input data. The producer follows a similar flow when attempting to push data to the output. In this example, the task is interacting with the OS of the IP cores to voluntarily suspend should certain conditions be true (*i.e.* finding a semaphore locked). Additionally, the task negotiates with the OS to be resumed upon interrupt receipt. The task may also want to ignore an interrupt in the following condition: it is possible that the upstream producer, or the downstream consumer, notifies availability of data or buffer space before the actual need for such resources, because the current task is still busy with previous internal processing.

C. multi

A task scheduling scenario (**“multi”**) is illustrated in Figure 8(a). In this case, two tasks (Task A and Task B) run on each IP; a variable amount of system processors may be present. No explicit communication is performed between tasks, neither intra- nor inter-core. The context switching between tasks is performed by the OS in response to an external interrupt, which may typically be sent by a timer device. The end of any task automatically triggers a context switch to the outstanding suspended task. Any nested interrupts arriving during the context switch are ignored, as would be expected in any well-behaving system. The tasks are not explicitly aware of any system synchronization going on, as they are not notified upon the receipt of an interrupt, and are just passively suspended and resumed by the OS. Since tasks can be asymmetric, a difference in OS scheduling might in turn translate into different traffic workloads.

D. IO

An I/O-aware application (**"IO"**) is illustrated in Figure 8(b). A single task is running on every system processor. These tasks do not communicate with each other, and perform independent computation. However, at random times, a system I/O device sends an interrupt to the IP cores to signal availability of data. In response to this signal, the IP executes an interrupt handler routine, which moves blocks of data across the system interconnect. When such handling is finished, normal operation is resumed. The interrupt handling is part of the functionality of an I/O device driver, and can be programmed as such.

Our RIPE models emulates a processor running an application. The application may or may not encompass OS behaviour, and may or may not be composed of multiple tasks per core. Apart from the **poll** scenario, in all the envisioned applications the OS plays an important role, with and both **multi** and **IO** involving some form of multiple tasks per core. The support for OS and multitasking modeling represents the second set of requirements for the RIPE.

The applications described above are timing-sensitive. However, within the single task, the overall performed computation does not change depending on the order of arrival of external events, and the data dependencies can be captured. Only the amount of computation between each pair of events can vary. Should an environment constraint not be satisfied, tasks always enter some form of suspension, albeit in very different manners in each of the examples. The different degree of awareness of OS functionality in each of these templates is important because it impacts the ability to annotate execution traces, as will be seen in Section VI. So, while an execution trace of these benchmarks shows varying traffic patterns depending on external timings, the major computation blocks are still

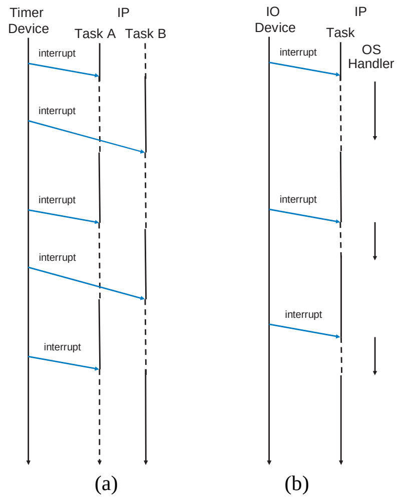


Fig. 8. Typical interrupt-triggered context switch timeline. (a) multi (b) IO

recognizable. Even though tasks with even more timing-dependent behaviour do exist, the effort required to model such tasks requires an intra-task notion of context switching. It is also worth stressing that, though not all interrupt-driven behaviours are represented, the applications we try to analyze here are definitely representative of a vast class of computation. The model we will propose can capture all such dynamics with proper insight on the mechanics of the applications and the OS.

The experimental results will prove that traces collected at the IP-fabric interface are sufficient to accurately reproduce the IP core communication, providing an important mechanism for RIPE validation. These traces should collect sequences of communication transactions, comprising of requests, responses and interrupt events, separated by time intervals with no communication, *i.e.* idle time. A reference simulation of the entire system should produce several traces, one per IP core interface.

IV. The RIPE Model

In this section, we motivate the choice of creating the RIPE as an instruction set processor, then describe its operation and implementation, which are capable of reproducing the required IP core reactiveness behaviour.

Instruction	Size (Words)	Description
Communication Instructions:		
Read(AddrReg)	1	Read from an address
Write(AddrReg, DataReg)	1	Write to an address
BurstRead(AddrReg, CountReg)	1	Burst read an address set
BurstWrite(AddrReg, DataReg, CountReg)	1	Burst write an address set
Flow Control Instructions:		
If(arg1, arg2, operand)	2	Branch on condition
Jump(label)	1	Branch direct
Idle(counter)	1	Wait for given no of cycles
SetRegister(reg, value)	2	Set register (load immediate)

TABLE I. RIPE instruction set.

A. Motivation for the Instruction Set Architecture

The RIPE must generate traffic patterns according to two different constraints. First, it must follow the directives set by the designer, who wants to inject a certain type of traffic in the system, typically shaped to emulate the traffic requirements of some application. Second, it has to respond dynamically to the external environment (congestion, synchronization events) in the same way the application that is being modeled would. To generate appropriate communication transactions "on-the-fly" respecting both requirements, an instruction set, supported by state registers and by a programming language, is a natural choice. By introducing a programmable paradigm, the RIPE can be used in association with manually written programs to generate traffic patterns typical of IPs still in the design phase, helping in the tuning of the communication performance or understanding the causality relationship with other IPs in the MPSoC. Hence, we choose an Instruction Set Architecture (ISA) for the RIPE implementation. This choice allows us to describe reactiveness characteristics of a wide range of IP cores at different levels of abstraction.

Additionally, this choice allows future deployment as a hardware device in test chips containing interconnect prototypes. In [17], the potential of this type of architecture has been shown within an FPGA-based emulation platform. The ISA approach, with a fixed device and user-written programs, avoids time consuming operations such as recompilation, in the case of behavioural models, or resynthesis, in the case of a hardware flow. Such steps would be required by a monolithic traffic generation device to emulate and study different applications on the same platform. In this paper, program execution will only be shown within a simulation model.

From the analysis of communication requirements in Section III, it can be postulated that three different RIPE entities might be needed:

- A RIPE emulating an IP master (a processor). This component must be able to issue conditional sequences of communication transactions separated by idle wait periods. Further, it must be sensitive to arrival of interrupt events and must support multiple threads.
- A RIPE emulating a private memory. This component must be able to respond to communication transactions issued by a master. The RIPE just has to model the access time but it does not have to provide a data structure for storage. It simply responds to a read transaction by providing a dummy value.
- A RIPE emulating a shared memory. This component must contain a data structure modeling an actual shared memory (since the values read by the masters may affect the application flow, *e.g.* current values of semaphores).

The second and third entities can be extremely simple in design, as their logic basically involves a small state machine to handle the communication protocol at the IP interface and possibly a storage element for corresponding memory accesses. In any case, for our tests within the MPARM framework we could use the equivalent MPARM blocks. Therefore, only the RIPE entity that emulates an IP master is described next, and is the main focus of this paper.

B. Instruction Set Architecture

The RIPE is implemented in SystemC [2] as a non-pipelined processor with a very simple instruction set, as listed in Table I. The RIPE program that controls the device behaviour contains code to model one or multiple tasks. These tasks might be actual tasks running on the IP core which is being modeled, or chunks of the OS layer, such as its native interrupt handlers and scheduler. The RIPE is capable of switching the execution flow among these tasks, as discussed later. Via the OCP 2.0 [4] master transaction interface, the RIPE is able to issue a sequence of communication transactions separated by idle wait periods, based on the programmed flow control conditions. The choice of the OCP protocol for the interface is motivated by the availability of this interface on the interconnect side within the MPARM reference system. Any other standard, such as AXI (Advanced eXtensible Interface) [7], could also

Special Registers	Name	Usage
Interrupt Registers:		
2	IntrpMaskReg	Masks or unmasks interrupts
3	TaskIDReg	Stores a task ID
5	SWIntrpReg	Sends a software interrupt from within the program
Other Registers:		
4	RDReg	Stores the data value returned by the Read(AddrReg) instruction

TABLE II. RIPE Special Registers.

be supported depending on the interface required by the interconnection under study.

The RIPE has a Program Counter (`PC`) register, an instruction memory and a register file for each task running on the core, but no data memory. Collectively, this state information drives the RIPE execution engine, whose state machine is described in the next section. The instruction set consists of a group of commands which issue OCP transactions (arguments are taken from the register file) and a group of flow instructions allowing the conditional programming of sequences of transactions and idle waits. Within the register file, most registers are general purpose, and their number can be configured.

Some registers are designated as special purpose; for example, since in specific flow control scenarios the data returned by a read command must be available for evaluation, RIPE provides in Register 4 the response to the preceding read. Table II shows all designated special purpose registers. Of the interrupt-related registers, Register 2 is used to mask critical sections of the RIPE program from interrupts. As seen in Section III, different applications require different responses to interrupt events. For example, in **IO** modeling, the main task is always interruptible, while once in the interrupt handling routine, additional (nested) interrupts should be temporarily skipped. In **pipe** modeling, the interrupt handling is more specialized; interrupts are only enabled after the task has suspended, while they are masked during normal operation. Register 5 allows the RIPE program to assert “software interrupts”, to which the RIPE model will react by loading the program and register set of the next thread. Register 3 can be programmed to hold the task ID of the next task to be loaded and run on the RIPE device out of the available task pool. The usage of the special registers will be shown in Section IV-D.

Software interrupts are managed internally by the RIPE model. In contrast, hardware interrupts are routed through external wires of the system fabric, and are available on the sideband portion (`SInterrupt`) of the OCP interface.

9

C. ISA Implementation

To execute the instructions discussed above, the RIPE model implements a simple non-pipelined engine where, within a single cycle, the instruction is fetched, decoded and executed. The RIPE can either initiate OCP transactions or perform flow control operations, including setting up register values.

The *Set Register* instruction executes the load of an immediate 32-bit value, which is written to the specified register (`SetRegister(reg, value)` in Table I). This opcode is two memory words long, as it has to accommodate the immediate data. The class of instructions relating to communication is designed to execute the OCP transactions. The OCP transactions are initiated with the address and data values that were set up in the register file in the preceding cycles. These instructions are blocking, *i.e.* the RIPE execution is suspended until completion of the OCP handshake, which for a read will include the latency of the response over the network. Currently, we support the basic signals and the burst extension of the OCP 2.0 specification. An extension to support out-of-order transactions could be achieved by the implementation of an outstanding instruction buffer. The class of instructions relating to flow control is used to realize the reactive behaviour. The `If` and `Jump` instructions are used to change the execution flow and the `Idle` instruction is used to fake the IP computation latency. The `If` opcode is two words long, to accomodate its operands and branch location.

A context switch among tasks in the task pool is realized simply by referring to the corresponding set of `PC` and register file. The explicit swapping operation, *i.e.* storing the state of current thread, loading the state of the next thread and eventually restoring the suspended thread, which is described in [6], is a byproduct of the presence of a single task memory in the device, and is no longer needed since each task now has its own independent program memory. The context switching is simultaneous with an incoming interrupt signal, thus avoiding inconsistencies. Upon interrupt notification, the `PC`, register file and program instruction memory are updated to the task ID read from the special-purpose Register 3.

The aforementioned instructions must be combined in a program and then transformed into a binary executable format for use within the RIPE ISA. The program syntax and the tool to generate RIPE executables are described next.

D. Programming Language and Assembler

The programming language to code traffic patterns of the RIPE is similar to an assembly language, though additional semantics are provided to make it user-friendly. It is best explained via the example shown in Figure 9, where a program to model the **IO** application is sketched. Statements starting with a semicolon (;) are inlined comments.

The RIPE program starts with a header describing the core and the task identifier: `MASTER[<coreID>, <taskID>]`. All of the tasks running on any given IP core are described within a single program, so that there is one program per RIPE device. Recall that **IO** models an application with a linear program flow, which can be suspended by the OS to process IO interrupts. Therefore, two tasks are described: task #0 (the main application) and task #1 (the interrupt handler).

The next few statements express initialization of the register file for this task. Unique labels should be used for register names/tags. This allows correct initialization and easy identification of the registers within the program. The `PC` is increasing by either one or two locations along the trace; this is because `SetRegister` and `If`, as seen in Table I, require longer operands and therefore fill two instruction slots. For task #0, the main body of the RIPE program, this is represented by a linear execution flow, composed of sequences of reads and writes, interleaved with register accesses (mostly, to set up transaction addresses and data). Flow control instructions might be inserted where appropriate, but are not needed in this model. Note the initialization of interrupt-related registers at the top of task #0; upon a hardware interrupt, the RIPE swaps the context to the task having the ID provided in `TaskIDReg`, *i.e.* to task #1 (the interrupt handler). Since task #0 can be suspended by OS to process I/O interrupts, `IntrpMaskReg` is set as unmasked, allowing for such suspension.

The OS-driven context switch traffic and the I/O handler routine are programmed in task #1. Within the interrupt routine (starting with label `IntrptHandler`), which is the critical section of the flow, interrupts are disabled. At the end of the flow, a software interrupt is triggered to restore the normal program flow to task #0. Upon another HW interrupt in the main task, the interrupt handler routine will be executed again from `PC 0`. The flow therefore mimics Figure 8(a).

An assembler was built to convert the human understandable RIPE program into a binary for execution on the RIPE device. There is a direct one-to-one correspondence between program instructions and the binary. Within the binary, the individual task sections are appended in order of their task ID. A header with a small task lookup table is prepended.

During setup, the RIPE device loads the binary, and based on the information encoded at the start of the binary file, it determines the number of tasks and the amount of program memory and the register file size to be allocated to each one.

```
MASTER[1, 0]                          ; Regular task              10
 ; Special Registers
   REGISTER IntrpMaskReg 0            ; Unmask Interrupts
   REGISTER TaskIDReg 1               ; Next Task ID
 ; General Purpose Registers (GPRs)
   REGISTER AddrReg 0xd0abcdef        ; Initialize address GPR
   REGISTER DataReg 0                 ; Initialize data GPR
   ...
BEGIN                                 ; Comments                  PC
   ; Normal application flow
   Idle(10)                           ; Idle for 10 cycles        0
   Read(AddrReg)                      ;                           1
   ...
   SetRegister(AddrReg, 0x10fedcab0)  ; Setup an address          121
   SetRegister(DataReg, 0x10abcdef0)  ; Setup a data value        123
   Write(AddrReg, DataReg)            ;                           125
   ...
END                                   ;                           1078
```

```
MASTER[1, 1]                          ; IO driver task
 ; Special Registers
   REGISTER IntrpMaskReg 0            ; Unmask Interrupts
   REGISTER SWIntrpReg 0              ; Disable SW Interrupts
   REGISTER TaskIDReg 0               ; Next Task ID
 ; General Purpose Registers (GPRs)
   REGISTER AddrReg 0                 ; Initialize address GPR
   REGISTER DataReg 0                 ; Initialize data GPR
   ...
BEGIN                                 ; Comments                  PC
 ; Interrupt Handling Routine
 IntrptHandler
  ; OS Suspension Routine
   SetRegister(IntrpMaskReg, 1)       ; Mask Interrupts           0
   SetRegister(AddrReg, 0x30bebeef)   ; Setup an address          2
   Read(AddrReg)                      ;                           4
   ...
  ; IO Routine
   SetRegister(AddrReg, 0x30beefcd)   ;                           39
   SetRegister(DataReg, 0x10101010)   ;                           41
   Write(AddrReg, DataReg)            ;                           43
   Idle(121)                          ;                           44
   ...
  ; OS Release Routine
   ...
   SetRegister(SWIntrpReg, 1)         ; Trigger SW Interrupt      106
   SetRegister(SWIntrpReg, 0)         ; Disable SW Interrupt      108
   Jump(IntrptHandler)                ;                           110
 ; End Interrupt Handling
END                                   ;
```

Fig. 9. RIPE Program for “IO" Example.

V. Using RIPE Programs

Depending on IP model availability to the designer, different ways exist to write RIPE programs which best represent the desired type of traffic.

A. Trace Parsing

In this scenario, as is seen in Figure 1, the availability of a pre-existing model for the IP under study is assumed.

```
MCmd WR MAddr 0x01bedfb0 MData 0x00015958 MBurstSingleReq 0 MBurstSeq INCR 0x4 MBurstLength 1 Time 6860265
SCmdAccept Time 6860295
SInterrupt SFlag 0x00000001 Time 6860310
MFlag Time 6860310
MCmd WR MAddr 0x010b48dc MData 0x00000008 MBurst SingleReq 0 MBurstSeq INCR 0x4 MBurstLength 1 Time 6860375
SCmdAccept Time 6860385
MCmd RD MAddr 0x0100acb0 MBurstSingleReq 1 MBurstSeq INCR 0x4 MBurstLength 4 Time 6860720
SCmdAccept Time 6860730
Resp Data 0xe5901000 Time 6860760
Resp Data 0xe2411001 Time 6860780
Resp Data 0xe5801000 Time 6860800
Resp Data 0xe14f0000 Time 6860820
MCmd WR MAddr 0x0102c040 MData 0x00000000 MBurstSingleReq 0 MBurstSeq INCR 0x4 MBurstLength 1 Time 6860830
SCmdAccept Time 6860840
```

Fig. 10. Trace syntax example.

In this case, the approach for RIPE program generation goes through two steps. First, a reference simulation is performed by using the available IP model, even if plugged into a different MPSoC platform from the final target one. In fact, since RIPE programs abstract from the transaction latency factor, a vary fast transaction-level model of the interconnect can be used in this stage to speed simulation up. An execution trace is collected. The trace is a very straightforward log of events on the OCP pinout; entries include requests, responses and interrupts, all of which annotated with timestamps. A sample trace snippet is sketched in Figure 10.

Second, the trace is parsed with an off-line tool. The output of the tool is the desired RIPE program. The resulting program is coded to behave exactly as the original IP model in the native system, and to behave as the core would do when plugged to a different interconnect. This program is now ready to be used for cycle-accurate interconnect design space exploration with extremely realistic test traffic.

This type of flow is useful whenever the pre-existing IP model is not available, due to licensing or technical issues, for the next co-exploration phase. In this case, the RIPE can provided a quick functional yet cycle-accurate port of the IP model to a MPSoC interconnect.

The off-line parsing tool must of course have some knowledge about the traced application in order to correctly analyze and rearrange execution traces into RTG programs. While this effort is not trivial, it is feasible and provides a path for validation of the presented RIPE device in a complete cycle-accurate flow, as described in Section VI.

B. Trace Parsing and Editing

In a related scenario, an IP model might be available, but it may differ under some respect from the IP that will eventually be deployed in the SoC device. In this scenario, the RIPE may be used to approximate the IP, as seen in Figure 2. The designer may then follow a route similar to the one outlined above, but with an additional step of editing the reference trace so that it more closely resembles that of the target IP. Some examples of the editing steps which are possible include:

- Removing or adding bus transactions to model a more or less efficient cache subsystem
- Removing or adding bus transactions to model a more or less comprehensive target Instruction Set Architecture (ISA)
- Altering the spacing among bus transactions to reflect different pipeline designs or timing properties
- Grouping or ungrouping bus accesses to reflect write-back *vs.* write-through cache policies

It is certainly reasonable to expect that the alteration time of the RIPE code will be substantially less than that required to develop or refine the target IP model, thus allowing for earlier exploration of the interconnect design space.

In this scenario, overall cycle accuracy with respect to the eventual system is of course not guaranteed. However, the RIPE will still be able to react with cycle accuracy to any optimization in the SoC interconnect. Provided that the transaction patterns are kept close to the ones of the target IP core, the approach will result in valuable guidelines.

C. Direct Development

Of course, RIPE programs can be written from scratch without reference IP traces. In this case, the flexible RIPE instruction set allows for a full-featured traffic generation system. The availability of built-in flow control management lets the designer implement the same synchronization patterns which are present in real world applications (see Section IV). Additionally, the application chunks enclosed within synchronization points can quickly be rendered by exploiting the flexible loop structures provided by the RIPE ISA, thus providing periodic traffic generation capabilities at least on par with those of traditional TG implementations as seen in [19] [11] and [17]. In the very first stages of development, the RIPE can also be deployed as a validation tool, to check the correct functionality of the interconenct under the load of the supported transaction types.

VI. Validation of RIPE

To test RIPE accuracy and viability, we set up a validation flow in a cycle-true environment, following the trace-based outline described Section V-A. As a first step, the user performs a reference simulation of the target applications where all IP cores are simulated using bit- and cycle-true models, to collect traces. Subsequently, the traces are processed into RIPE programs. The following sections describe these steps in detail.

A. Reference MPSoC System

To achieve validation, the RIPE model was integrated into the MPARM [20] reference system. MPARM is a homogeneous multiprocessor instruction-set simulation (ISS) platform with a configurable number of processors as IP masters with private and shared memories, and semaphore and interrupt devices. It also contains a port of RTEMS [3] - a real-time OS. The IP cores can be plugged onto one of several interconnect architectures, such as AMBA [8], STBus [27] and ×pipes [13]. The use of the OCP v2.0 protocol at the interfaces between the IP cores and the interconnect allows for easy exchange of native cores with RIPE blocks (Figure 1). To record execution traces, the OCP interface modules within the MPARM system (the AMBA AHB bus master) were adapted to collect traces of OCP requests, responses and interrupt events in a predefined file format (*.trc*).

It is worth stressing that the complexity of the applications described in Section III is not trivial from the modeling point of view. The amount of annotations that can be extracted from the application and its traces reflects the programmer's degree of knowledge and access to the application synchronization schemes, to the interrupt routines and to the OS internals.

B. Trace to RIPE Program

The RIPE validation flow is illustrated in Figure 11. During the reference simulation, traces are collected from all OCP interfaces in the system. The address and (if any) data fields of the transactions were also observed. Trace entries may contain one of many transaction types: single or burst read/write requests, assertion of hardware interrupt, arrival of response, etc. Figure 12(a) shows an example trace.

The next step is to convert the traces into corresponding RIPE programs (*.tgp*). The off-line translator tool outputs symbolic code; Figure 12(b) shows the RIPE program derived from traces in Figure 12(a). We will explain the translator operation in detail in SubsectionVI-C. Finally, an assembler is used to convert the symbolic RIPE program into a binary image (*.bin*) which can be loaded into the RIPE instruction memory and executed.

The off-line tool for trace to RIPE program conversion is written to exploit the sophisticated way application tasks can be described in RIPE programs and the multi-tasked architecture described in Section IV. The automated algorithm in the conversion flow is capable of detecting and capturing many synchronization behaviours, without the need for the designer to handle them manually, and is explained next. Validation of the trace collection and processing mechanisms can be achieved by collecting traces with IP cores running on different interconnects, and verifying the resulting *.tgp* and *.bin* programs to match. The conversion process is fully automated and the time taken for this process is discussed in Section VII.

C. Translator Operation

In this section we detail the working of the translator. We use the system traces given in Figure 12(a) as an example source for transformation into a RIPE program, and the result is in Figure 12(b).

As discussed in Section III, some prior knowledge about the IP core used in the reference simulation is required to accurately program the RIPE device. Apart from the sequence of transaction requests and responses, following is a list of information needed for correct operation of the translator:

- The global identifier of the IP core in the MPSoC system
- The clock period of the IP core
- The addressing ranges representing semaphore (pollable) resources
- The timestamp of interrupt events
- The timestamp of the return from an interrupt handling routine
- The timestamp of a spontaneous control yield

The first three pieces of information are encoded in the trace filename, the rest are explicitly or implicitly (provided some knowledge of the application functions) annotated within the trace file. For example, incoming interrupts are detected on the OCP pinout and explicitly recorded in the trace. On the other hand, returns from interrupt handling routines must be located implicitly by detecting known behaviour, such as a specific memory access at the end of the handler or at the return point in the main code. Based on the above information, we first

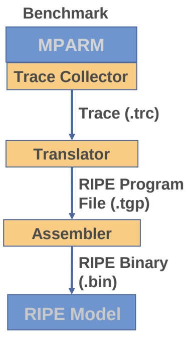
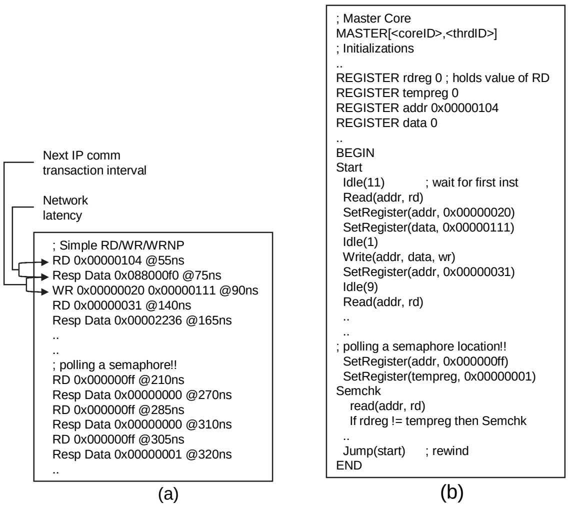


Fig. 11. Trace to RIPE Program Flow.


Next IP comm transaction interval
Network latency
(a)
(b)

```
; Simple RD/WR/WRNP
RD 0x00000104 @55ns
Resp Data 0x088000f0 @75ns
WR 0x00000020 0x00000111 @90ns
RD 0x00000031 @140ns
Resp Data 0x00002236 @165ns
..
..
; polling a semaphore!!
RD 0x000000ff @210ns
Resp Data 0x00000000 @270ns
RD 0x000000ff @285ns
Resp Data 0x00000000 @310ns
RD 0x000000ff @305ns
Resp Data 0x00000001 @320ns
..
```

```
; Master Core
MASTER[<coreID>,<thrdID>]
; Initializations
..
REGISTER rdreg 0 ; holds value of RD
REGISTER tempreg 0
REGISTER addr 0x00000104
REGISTER data 0
..
BEGIN
Start
  Idle(11)          ; wait for first inst
  Read(addr, rd)
  SetRegister(addr, 0x00000020)
  SetRegister(data, 0x00000111)
  Idle(1)
  Write(addr, data, wr)
  SetRegister(addr, 0x00000031)
  Idle(9)
  Read(addr, rd)
  ..
  ..
; polling a semaphore location!!
  SetRegister(addr, 0x000000ff)
  SetRegister(tempreg, 0x00000001)
Semchk
    read(addr, rd)
    If rdreg != tempreg then Semchk
  ..
  Jump(start)       ; rewind
END
```

Fig. 12. (a) MPARM Trace (b) RIPE Program.

describe the insertion of the `SetRegister` instruction within the RIPE program, which is critical to initiate the correct OCP transactions and flow control behaviour; and then we describe how the reactiveness is realized.

As seen in Figure 12(b), and described in Section IV-D, the RIPE program starts with the typical core identifiers. For the illustrative example in Figure 12(a), let the clock period be 5ns and the semaphore location be `0x000000ff`. Register `RDReg` is defined as the name of the special register where the value of read transactions is stored (Table II).

At the beginning of the trace file, the first communication request, a read (`RD`), occurs at 55ns, meaning the RIPE has to perform 11 (55/5) cycles of idle wait in the first place. Therefore, an `Idle` wait is observed in the RIPE program. When parsing this trace statement, the translator collects the `RD` address and initializes one of the registers marked as available in the register table (tagged as `addr` on top of the program). The response is received at 75ns; the translator simply skips to this timestamp, since response latencies are only dependent on the underlying network and the IP core (and so the RIPE) is simply blocked in the meanwhile. The next trace event of interest is the write `WR` request at 90ns. This means three ((90-75)/5) cycles have elapsed since the previous response is received. New values have to be set up in the address and data registers, which takes a cycle each (either for updating the already used `addr` and `data` or for setting up a new pair of registers). An ensuing `Idle` wait is added to fill the gap. This represents the “time-shifted’ behaviour discussed in Section II: if the RIPE program is run on a different interconnect where the read response latency is different, the write request will be accordingly shifted backward or forward from the 90ns timestamp.

Then, the following read request is translated into the corresponding `RD` program call, which is issued after ten cycles, one spent to set up the target address and nine in idle waiting. Please notice that write transactions in the OCP protocol can be posted, as we assume in this example; hence the time gap (equivalent to some processing time within the IP core that is being replaced) between the previous write command and the current read is noted by the translator in the RIPE program. The read is blocking until a response is received, five cycles later.

Now, consider the trace entries from time 210ns to 320ns. By identifying the address as belonging to a semaphore location and knowing the polling behaviour of the original IP core, the translator inserts the `Semchk` label and an `If` conditional statement. This statement checks whether the read value is equal to “1”, which reflects an unblocked semaphore. This loop effectively models the semaphore polling behavior. The semaphore address and expected unblock value are set up prior to the loop label to avoid repeated initialization, thus allowing for continuous polling at the maximum frequency rate for unlimited periods. Idle waits can obviously be added in the loop should the original IP core have a low-frequency polling behaviour. All master devices attempting to accessing this

```
; multi trace for Core ID #3
RD 0x00000104 @15ns
Resp Data 0x088000f0 @45ns
WR 0x00000020 0x00000111 @95ns
...
RD 0x00000031 @120ns
Resp Data 0x00002236 @225ns
SInterrupt @365ns
RD 0x00000031 @440ns
Resp Data 0x00002236 @465ns
RD 0x0000beef @540ns
Resp Data 0x00002236 @565ns
...
WR 0x00000020 0x00000111 @390ns
SInterrupt @595ns
Burst RD MAddr 0x01009340 Length 4 @620ns
Resp Data 0x00027864 @710ns
Resp Data 0x00029994 @730ns
Resp Data 0xe52de004 @750ns
Resp Data 0xe59f0004 @770ns
...
...
...
```

(a)

```
MASTER[3, 0]                            ; Initializations Task A
 ; Special Registers
   REGISTER IntrpMaskReg 0              ; Unmask Interrupts
   REGISTER TaskIDReg 1                 ; Next Task ID upon Interrupt
 ; General Purpose Registers (GPRs)
   REGISTER AddrReg 0x00000104          ; Initialize GPR labeled AddrReg
   REGISTER DataReg 0                   ; Initialize GPR labeled DataReg
   ...
BEGIN                                   ; Comments
   Idle(3)
   Read(AddrReg)                        ; RD @15ns
   Idle(8)
   SetRegister(AddrReg, 0x00000020)
   SetRegister(DataReg, 0x00000111)
   Write(AddrReg, DataReg)              ; WR @95ns
   ...
   SetRegister(AddrReg, 0x00000031)
   Read(AddrReg)                        ; RD @120ns
   Idle(15)
   SetRegister(AddrReg, 0x01009340)
   SetRegister(CountReg, 0x4)
   BurstRead(AddrReg, CountReg)         ; Burst RD @620ns
   ...
   SetRegister(SWIntrpReg, 0x00000001)  ; Trigger SW Interrupt
   SetRegister(SWIntrpReg, 0x00000000)  ; Disable SW Interrupt
END
```

(b)

```
MASTER[3, 1]                            ; Initializations Task B
 ; Special Registers
   REGISTER IntrpMaskReg 0              ; Unmask Interrupts
   REGISTER SWIntrpReg 0                ; Diable SW Interrupts
   REGISTER TaskIDReg 0                 ; Next Task ID upon Interrupt
 ; General Purpose Registers (GPRs)
   REGISTER AddrReg 0x00000031          ; Initialize GPR labeled AddrReg
   REGISTER DataReg 0                   ; Initialize GPR labeled DataReg
   ...
BEGIN                                   ; Comments
   Idle(26)
   Read(AddrReg)                        ; RD @440ns
   Idle(74)
   SetRegister(AddrReg, 0x0000beef)
   Read(AddrReg)                        ; RD @565ns
   ...
   SetRegister(AddrReg, 0x00000020)     ; Trigger SW Interrupt
   SetRegister(DataReg, 0x00000111)     ; Disable SW Interrupt
   Write(AddrReg, DataReg)              ; WR @390ns
   ...
   SetRegister(SWIntrpReg, 0x00000001)  ; Trigger SW Interrupt
   SetRegister(SWIntrpReg, 0x00000000)  ; Disable SW Interrupt
END                                     ;
```

(c)

Fig. 13. RIPE Program for “multi" Example. (a) MPARM trace, (b) Task A, and (b) Task B.

semaphore incorporate the same routine in their RIPE program, thus capturing the system dynamics.

Within the translator, a register allocation algorithm correctly sets up all the required data in registers before the OCP or the flow-control instructions that need them are scheduled for execution. It is possible that streams of closely packed communication requests may leave few or no interleaved idle cycles available for preparing their address (and data, if any). The solution is to exploit the slack (idle wait time) available further above in the transaction sequence for setting up register values for upcoming instructions. The translator algorithm attempts to use such slack as much as possible to prefetch register contents. However, if packed streams are very long, the problem may be further compounded by lack of free registers. In this case, the only solution is to increase the size of the register file. We expect the problem to occur with minimal frequency, as two idle cycles (for writes) or even just one (for reads) among transaction entries are enough to allow for streams of arbitrary length. Otherwise, the maximum length of streams will be directly limited by register file size. This is of no importance in the context of a simulation RIPE device (as in this paper), but would have an area penalty in a hardware implementation. In the event of lack of registers, the translator tool prompts the user to increase the size of the register file in the RIPE architecture and to attempt the translation again.

D. Handling Interrupt Reactiveness

As mentioned before in Section III, the amount of annotations that can be extracted from a trace reflects the degree of access the programmer has to the interrupt routine and to the OS internals. Specific locations within the trace file, such as interrupt handling routine entry and exit points, have to be recognized by the translator tool to optimally insert the corresponding code as a task into the RIPE task pool.

The trace files are always annotated with the time of occurrence of interrupt events. For the **IO** benchmark, the interrupt handling routine is supposed to be accessible by the programmer, as described in Section III; thus, a marker (a dummy transaction to a known address) can be added at the exit of the routine to tag it. The transactions within these bounds are detected as interrupt handling code and are encapsulated as such in the RIPE program. In Figure 9 we have seen the backbone of the **IO** RIPE program, where interrupt response blocks are handled so as to mimic Figure 8(b).

Using **multi** as an example, let us consider the interrupt-triggered reactiveness in more detail. Here, the trace files are annotated only with the time of occurrence of interrupt events. Indeed, recall that in the **multi** benchmark the interrupt handler is supposed to be completely out of the programmer's control, as it is tied to the OS scheduling code. The IP core toggles among the two tasks upon these interrupts. Additionally, control is never spontaneously released by means of SW interrupts: the previously active task is only resumed upon arrival of a HW interrupt. Thus, the translator's job is simply to capture the OCP transaction stream between two successive interrupts (identified by the `SInterrupt` tag in the trace) and append it to the corresponding task program, knowing that the scheduling pattern will be alternating. A minor inaccuracy in this approach is that the code of the OS which manages the rescheduling cannot be isolated by the translator, and is instead captured as a part of the instructions of the next task. Despite the above approximation, experimental results show a negligible accuracy skew.

Figure 13 shows the trace (a) and RIPE program (b) and (c) for a processor (in this case ID 3) performing two tasks in **multi** scenario. By default, in Figure 13(a), the set of instructions until the first HW interrupt (at 365ns) are identified with task A, which is then coded into corresponding

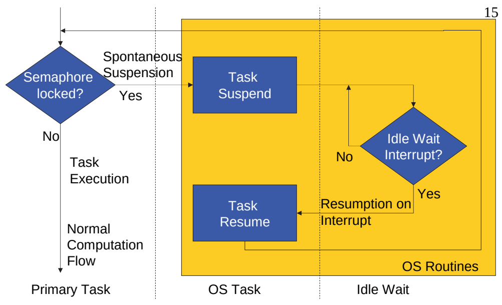


Fig. 14. Application flow of pipe.

program in Figure 13(b). Upon the HW interrupt, the next set of events are mapped to task B, which is then coded into corresponding program in Figure 13(c). Upon encountering the next interrupt (at 595ns), the translator toggles back to coding task A and this operation continues to the end of the trace. When appending subsequent execution blocks of the same task, the translator automatically adjusts the relative timing between transactions as if the task had executed without interruption. At the end of each task listing, a SW interrupt routine is inserted to yield control to any other task running on the processor whose execution is still incomplete. This matches what could be expected of well behaving OSes, where the end of one task prompts a non-timer-triggered rescheduling to switch to other pending tasks to finish the remaining portion of their instructions. Any further HW interrupts from the timer device are internally masked as meaningless during this final phase of execution, since there is only one schedulable task remaining. During execution, the RIPE ISS automatically supports context switching, as described in Section IV: upon an HW interrupt, the RIPE device simply loads the next instruction from the task whose ID is found in the `TaskIDReg` special register.

In the **pipe** scenario, the task is explicitly interacting with the OS internals, as described in Section III. Usually this interaction can be achieved by OS API calls, without direct access to the interrupt handler code, whose exit point is therefore assumed to be not accessible to the programmer. As a result, the only annotations of significance within the trace file are the synchronization points (semaphore checks) and the interrupt arrival time. The RIPE program thus mimics the flow shown in Figure 8(c), first by reading the semaphore location, then choosing to continue or suspend depending on the lock. Upon resumption by HW interrupt, a final (re-)check of the semaphore unlock is done to ensure safe task operation. Figure 14 shows the equivalent flow. In the RIPE program, this is realized via three tasks (dotted lines mark their boundaries). The primary task represents the main application flow. The interrupts are masked here, as the application is insensitive to HW interrupts unless in suspension state. If the semaphore is found locked, the flow is derouted to load the OS routine which leads the processor to an idle wait. The translator captures the chunk of trace after the semaphore check in an independent OS task, which always yields control to a third task consisting of an infinite loop of idle wait instructions. The easily identifiable sequence of transactions between the eventual arrival of the HW interrupt and the semaphore re-check is the OS wake-up routine to reschedule the suspended main program, and the translator appends it as a part of the OS task. In the RIPE program, HW interrupts are used to wake up from the suspension state within OS routines, while SW interrupts redirect the execution flow towards the main task. Note that `IntrpMaskReg` is set to "masked" for the regular program and OS execution, and is only unmasked within the suspension task.

After performing the translation described in this Section and after RIPE program assembling, a second set of simulations can be run on a platform with RIPE and a variety of interconnect fabrics, thereby evaluating performance of interconnect design alternatives.

VII. Validation Results

As discussed earlier, for validation we simulated the different benchmarks within the MPARM framework, first using the native ARM cores and then using the RIPE model, and compared the resulting benchmark statistics. We undertook this experiment for six benchmarks. Each was tested with one to twelve (1P-12P) system processors simultaneously plugged to the system interconnect, except where the application needed at least two or three cores for functional reasons. The aim was to ascertain the accuracy of the RIPE approach when stressed by complex transactions.

Four of the benchmarks are the applications described in Section III. Two more applications were added as a reference. **Cacheloop** is a dummy program, which continuously performs cache fetches. As such, it is generating no bus transactions, except for a few at boot and shutdown. It is intended as a metric of the maximum simulation time speedup achievable by replacement of IP cores with another simulation device. **Matrix** is a benchmark where the application involves one task per processor performing some private computation. Since no inter-core synchronization is used at all, modeling is very simple and could be achieved also by traditional TG approaches. The only source of uncertainty is due to the fact that all tasks compete for access to the same interconnection resource, which impacts transaction latency. This test is useful to

16

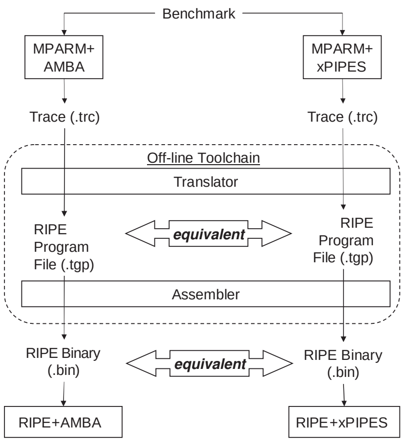


Fig. 15. RIPE and MPARM Accuracy Test.

see if RIPE is correctly responding in a "time-shifting" scenario, as discussed in Section II and III.

For **multi** and **IO**, we devoted one of the system cores to the generation of interrupts, emulating the role of a timer or an IO device; this processor is not generating any other traffic on the bus, and is just idling between interrupt generation events. The **pipe** benchmark does not need this, since interrupts are directly triggered by the same tasks which perform the computation.

In the first experiment, we only aimed at validating the trace collection and off-line processing environment. Figure 15 outlines the process. We ran the same benchmarks over two of the interconnects of MPARM, namely AMBA AHB [1] and the ×pipes [26] NoC, noticing very different execution times due to different latency and scalability features. Execution traces reflected these differences. However, after translation, a check across *.tgp* programs showed no difference at all, because the network latency factor is completely abstracted from in the RIPE programs. As a consequence, a trace collected on one interconnect could be used to generate a program to be run on another; the resulting execution would match that of the same benchmark natively run on the second interconnect. This result strengthens the postulate of the feasibility of an effective approach which decouples simulation of the IP cores and of the underlying interconnect fabric.

Table III summarizes the results of simulations done on the AMBA AHB interconnect with ARM processors from MPARM and then with RIPEs. The different columns relate to cumulative execution (Cmlt. Exec.) cycles of the benchmarks, the number of single read (SR), single writes (SW) and burst reads (BR) transactions observed on

Benchmarks	# IPs	RIPE					MPARM					Comparison				
		Cult. Exec. Cycles	SR	SW	BR	Sim Time (s)	Cmlt. Exec. Cycles	SR	SW	BR	Sim Time (s)	Accuracy				Speedup (x)
												Exec %	SR %	SW %	BR %	
SP Cacheloop	1	2500692	0	16	25	8	2500700	0	16	25	15	0.000%	0.000%	0.000%	0.000%	1.88
SP matrix	1	1324132	0	58751	92	5	1324138	0	58751	92	9	0.000%	0.000%	0.000%	0.000%	1.80
Cacheloop	2	2500916	0	32	51	10	2500908	0	32	51	26	0.000%	0.000%	0.000%	0.000%	2.60
	4	2501721	0	64	106	15	2501714	0	64	106	49	0.000%	0.000%	0.000%	0.000%	3.27
	6	2502565	0	96	156	22	2502558	0	96	156	67	0.000%	0.000%	0.000%	0.000%	3.05
	8	2503321	0	128	201	28	2503314	0	128	201	87	0.000%	0.000%	0.000%	0.000%	3.11
	10	2504137	0	160	251	35	2504130	0	160	251	117	0.000%	0.000%	0.000%	0.000%	3.34
	12	2504953	0	192	301	40	2504946	0	192	301	141	0.000%	0.000%	0.000%	0.000%	3.53
Matrix	2	1324711	0	117502	186	7	1324717	0	117502	186	16	0.000%	0.000%	0.000%	0.000%	2.29
	4	1326582	0	235004	374	12	1326588	0	235004	374	28	0.000%	0.000%	0.000%	0.000%	2.33
	6	1330971	0	352506	562	16	1330977	0	352506	562	39	0.000%	0.000%	0.000%	0.000%	2.44
	8	1421281	0	470008	750	22	1421272	0	470008	750	52	0.001%	0.000%	0.000%	0.000%	2.36
	10	1776352	0	587510	921	32	1776343	0	587510	921	77	0.001%	0.000%	0.000%	0.000%	2.41
	12	2131618	0	705012	1105	45	2131609	0	705012	1105	104	0.000%	0.000%	0.000%	0.000%	2.31
poll	2	881839	7176	71764	254	4	883977	7201	71764	254	10	0.242%	0.347%	0.000%	0.000%	2.50
	4	975267	18241	143596	508	8	976488	18183	143596	508	20	0.125%	0.319%	0.000%	0.000%	2.50
	6	1049145	31057	215460	762	12	1049965	31101	215460	762	30	0.078%	0.141%	0.000%	0.000%	2.50
	8	1139110	46044	287356	1016	17	1140199	46300	287356	1016	44	0.096%	0.553%	0.000%	0.000%	2.59
	10	1385053	71989	359284	1270	24	1385007	71966	359284	1270	62	0.003%	0.032%	0.000%	0.000%	2.58
	12	1678901	96756	431244	1524	36	1678804	96689	431244	1524	84	0.006%	0.069%	0.000%	0.000%	2.33
multi	2	1823882	14	85729	24764	9	1824135	14	85729	24764	19	0.014%	0.000%	0.000%	0.000%	2.11
	4	2224333	42	192745	52242	17	2225867	42	192745	52242	37	0.069%	0.000%	0.000%	0.000%	2.18
	6	2818936	70	299963	80158	30	2820912	70	299963	80158	60	0.070%	0.000%	0.000%	0.000%	2.00
	8	3482223	98	407707	109820	48	3482793	98	407707	109820	91	0.016%	0.000%	0.000%	0.000%	1.90
	10	4129205	126	515815	138427	64	4135736	126	515815	138427	136	0.158%	0.000%	0.000%	0.000%	2.13
	12	4800566	154	624107	167789	89	4801433	154	624107	167789	184	0.018%	0.000%	0.000%	0.000%	2.07
IO	2	1156047	2560	68494	18271	6	1158639	2560	68495	18271	12	0.224%	0.000%	0.001%	0.000%	2.00
	4	1446888	2560	145826	36966	11	1449109	2560	145827	36966	24	0.153%	0.000%	0.001%	0.000%	2.18
	6	1870491	2560	223166	55654	20	1872248	2560	223167	55654	39	0.094%	0.000%	0.000%	0.000%	1.95
	8	2325228	2560	300514	74435	31	2325625	2560	300515	74435	60	0.017%	0.000%	0.000%	0.000%	1.94
	10	2780595	2560	377947	93274	44	2781660	2560	377948	93274	95	0.038%	0.000%	0.000%	0.000%	2.16
	12	3241959	2560	455465	112037	62	3242080	2560	455466	112037	111	0.004%	0.000%	0.000%	0.000%	1.79
pipe	2	745386	2601	56004	16293	4	754998	2601	56004	16293	7	1.273%	0.000%	0.000%	0.000%	1.75
	4	1051512	5246	114118	33257	9	1055056	5247	114298	33313	16	0.336%	0.019%	0.157%	0.168%	1.78
	6	1430317	7888	171880	49895	16	1436149	7888	171880	49895	29	0.406%	0.000%	0.000%	0.000%	1.81
	8	1829005	10530	229675	66321	25	1833183	10530	229675	66321	44	0.228%	0.000%	0.000%	0.000%	1.76
	10	2240354	13172	287435	83114	37	2243537	13175	287975	83296	66	0.142%	0.023%	0.188%	0.218%	1.78

TABLE III. RIPE vs. ARM performance with AMBA.

18

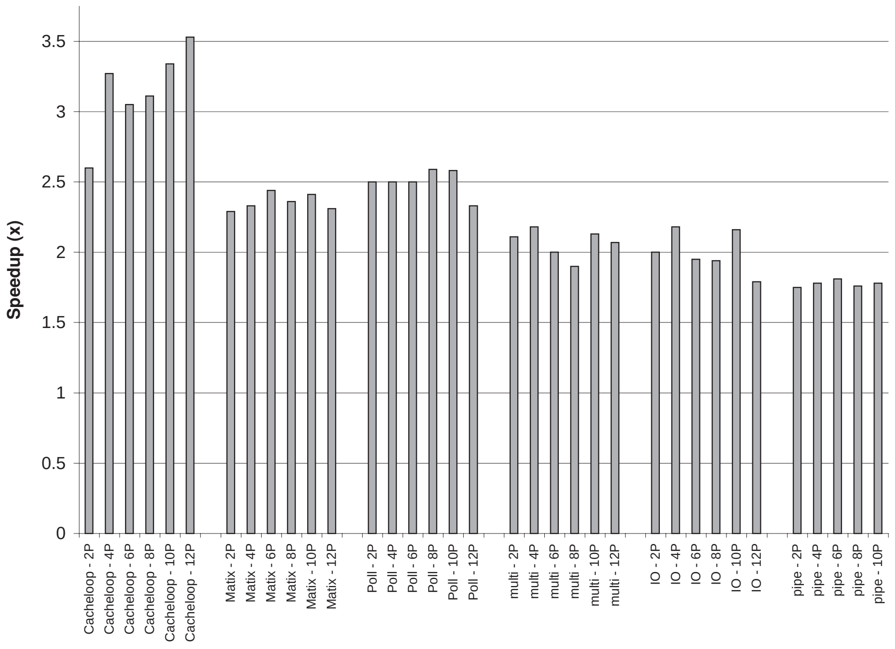


Fig. 16. RIPE vs MPARM Speedup.

the bus. The simulation time[1] The simulation time (Sim Time) is accounted in seconds. The column "Accuracy" is a measure of the accuracy of replacing IP cores with RIPEs, based upon the difference in simulated cycles and bus accesses, while the column "Speedup" describes the improvement in simulation time.

The table shows that replacing ARM processors with RIPEs yields excellent accuracy, over 99% in most cases, resulting in a faithful reproduction of the original execution flow and traffic pattern. The near-matching amount of read and write accesses validates the correctness of our RIPE program translation (see Section VI). Inaccuracies in execution time can be explained as follows. In **poll**, the amount of single reads is the primary source of inaccuracy. This is due to the compounding of minimal timing mismatches caused by the semaphore polling mechanism in RIPE programs. In the real system, the first few semaphore polls were found to occur at a slightly different rate than subsequent ones, due to assembler-level and caching effects. Eventually, polling occurs at periodic intervals. This initial timing mismatch is not captured in the RIPE, which performs all polling loops at the asymptotic rate. This causes RIPE to be affected by a small timing skew, which impacts subsequent simulation. As results prove, this has negligible consequences on the application flow, which is dominated by the interconnect delay.

The inaccuracies in OS- and interrupt-related benchmarks are due to minor issues in properly pinpointing different sections of OS code in the execution trace, as discussed before in Section VI. The near-matching statistics however fully prove the role of the RIPE as a powerful design tool to mimic complex application behaviour in replacement of a real IP core.

Scalability tests, performed by increasing the number of processors attached to the bus, exhibit two main different trends, as seen in Figure 16. **Cacheloop** exhibits a fundamentally monotonic trend, showing the advantage of replacing a progressively increasing amount of system cores with a faster device model. Other benchmarks show a fundamentally constant figure, or an increase with the number of processors which gets capped at some point (for example, **Matrix**). This seemingly strange behaviour can be explained by recalling that the system being sim-

[1]Benchmarks taken on a Pentium 4® 2.26GHz with 1 GB of RAM. The absence of disk swapping effects was checked during simulation. Especially for benchmarks with a short duration, time measurements were taken by averaging over multiple runs and care was put in minimizing disk loading effects.

	Interval among interrupts to same core (ms)	Notes 19
Reference	2	
Case I	1	
Case II	2	Processors receive interrupts staggered by a 0.5 ms offset
Case III	2	Two processors receive an extra interrupt just after the boot

TABLE IV. Interrupt issue frequency for four different multitasking patterns

ulated is also composed of the interconnect model and of some simulation support (simulation scheduler, statistics collection, etc.). Therefore, the simulation time cannot be decreased below a certain threshold. Further, an increase in the number of processors also implies more traffic on the interconnect, shifting the simulation load towards the latter and hindering any speedup. At a certain point, the fabric becomes completely saturated. In this condition, no further speedup is achievable at all because both ARM and RIPE execution time is dominated by idle waits for bus responses - a situation where the ARM simulation model can be as fast as whatever possible replacement. To support this analysis, we observe that the lowest speedup is achieved for **pipe**, which is also found to be the benchmark with the highest bandwidth requirements (and therefore the highest load on the interconnect model). We would like to stress that, as **Cacheloop** demonstrates, this decrease in simulation speedup is not a shortcoming of our RIPE approach, and is instead a direct consequence of benchmark and system behavior. **IO** and **multi** speedup figures are a bit higher than those of **pipe** also thanks to the presence of one basically idle processor devoted only to interrupt generation. In absolute terms, a gain of 1.75x to 3.53x was observed when running the benchmark code on RIPEs as opposed to ARM ISSs. This speedup is due to the removal of the computation logic within cores. It is noteworthy that even though speedup is not the primary objective of RIPE, it compares favorably to previous work in the area (a speedup of 1.55x is reported in [23]), especially given the fact that it is achieved at the cycle-true level of abstraction.

The time penalty for trace collection is small, and is incurred only once. For example, when running the relatively complex **pipe** benchmark on the AMBA interconnect with four ARM processors, a benchmark run augmented to collect reference traces takes 20 s, and subsequent translation and elaboration requires an additional 12 s for a 5.6 MB trace file. Only one such iteration is needed to validate the RIPE model and for subsequent design space exploration. Additionally, since processed RIPE programs are identical regardless of the reference interconnect in which raw traces were collected, such collection could be performed on top of a transactional fabric model, further reducing the impact of the reference simulation.

VIII. Case Study

To demonstrate the potential of the RIPE as a co-exploration tool, we look at a variant of the **multi** application, first discussed in Section III-C, in more detail. Specifically, we consider a five processor bus-based system with one RIPE configured to act like a timer device. This core triggers the delivery of interrupts at regular intervals to the other four RIPE devices, which as a result switch among two tasks. The two tasks are tuned to have very different bandwidth requirements; one task performs matrix manipulations (**MM**), and heavily relies on data caches to minimize memory transactions, while the second task performs streams of writes (**WS**) to a memory attached to the bus. The **WS** task is very demanding on the interconnect and can easily saturate it, therefore hurting overall system performance.

In this case study, using the RIPE, we test the behaviour of this system for different interrupt delivery policies and study the resulting traffic profiles (Fig. 17-20). This type of exploration may be useful to schedule bus accesses for real-time tasks in critical systems. The traffic plots show the profile of the bus traffic over time, expressed as transferred data words over a time window of 2 μs. This method of presentation is useful to note the load on the bus over the complete execution period, without the need for cumbersome investigation of correlation among different processors via individual bus activity plots.

For these experiments, to achieve maximum realism, the RIPE programs modeling the tasks on the four computation cores were created by translating MPARM execution traces. However, they could have easily been written by hand. In MPARM, interrupts are triggered by writing to a specific address of a memory-mapped device; therefore, to trigger the interrupts that should come from a timer device, we wrote a small RIPE program issuing OCP writes at the right times. In turn, this is achieved by parameterized idle waits. Such a program was written in a dozen of lines of RIPE code.

In all the plots, until about the 6000 μs mark, the bus activity during the OS boot is observed. The boot activity is irregular, but on average pretty intensive in terms of required bandwidth, since all the processors are loading the OS and application instructions from the memory across the interconnect. After this mark, application code begins to be executed. In Fig. 17, a straightforward scheduling policy is used: a timer interrupt is sent to each core simultaneously, therefore causing all of the cores to switch among **MM** and **WS** at the same time. Since interrupts arrive simultaneously to all processors, all of them are

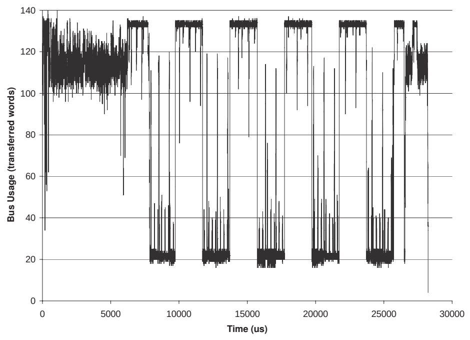
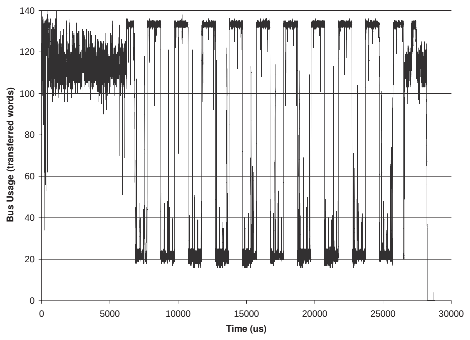


Fig. 17. Reference traffic pattern



20

Fig. 18. Case I

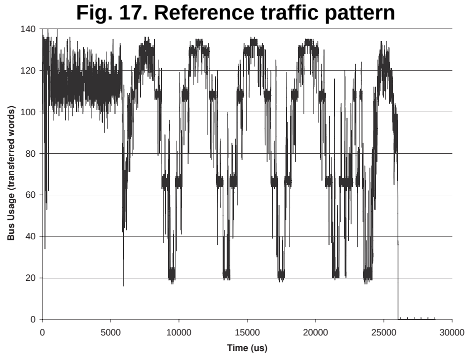


Fig. 19. Case II

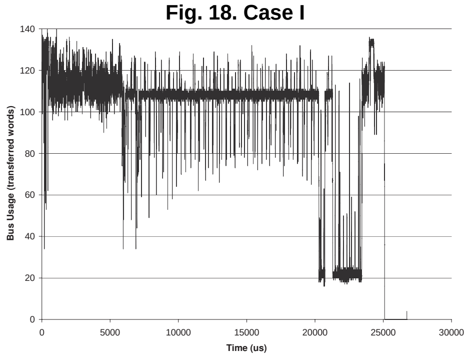


Fig. 20. Case III

in the same task group during any given time slice of execution. As expected, the bus load shifts depending on the task characteristics; the traffic profile exhibits a clear alternating pattern among two disproportionate usage values, with peaks above 130 and a floor of around 20 transactions per time window. The number of transitions between these two limits and the width of each peak correspond to the number of issued interrupt events and the interval between them (see Table IV). The tail of the plot is representing shutdown code, and is not relevant.

Since excessive contention inflates the response latency of the bus and therefore hurts performance, the traffic profile must be reshaped to decrease congestion. As is observed in Fig. 18, as compared to Fig. 17, doubling the interrupt issue frequency does little to mitigate the bus congestion issue; it only shifts the contention to a different time slot. Execution time remains constant at about 28200 μs.

Let us now consider the impact on the bus activity of staggering the interrupt events. In Fig. 19, we see the impact of issuing interrupts to each processor at the same frequency as in the reference case; however, the interrupts sent to each processor are staggered with respect to the interrupts sent to other cores by 25% of the original time window. As a result, an interrupt is sent every 500 μs, but two interrupts to the same processor are spaced 2000 μs apart. The traffic profile is smoother; thanks to staggering, **MM** tasks on some cores run in parallel to **WS** tasks on other cores. Over time, the system shifts from running four **MM** tasks to running four **WS** tasks and back, which results in a sinusoidal-like trend with visible steps. Peak congestion is only reached during a shorter fraction of the time, therefore reducing the execution time to about 26000 μs.

To balance the traffic even better, the clear choice is to always overlap two **MM** and two **WS** tasks. This is achieved in Fig. 20, where two processors are forced to perform a context switch just after the OS boot, and the subsequent interrupt pattern is the same as in Fig. 17. Thanks to much better traffic balancing, the bus never saturates, providing good performance and decreasing the execution time to 25200 μs.

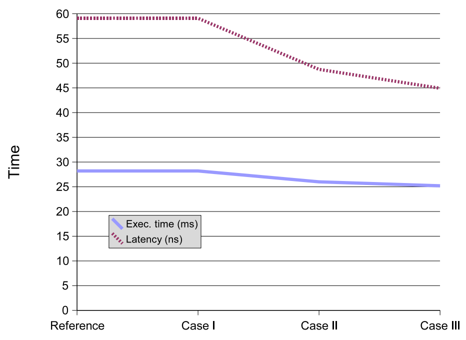


Fig. 21. Performance of the four synchronization patterns under test

In Fig. 21, the benchmark execution time and the average communication latency for a write transaction on the bus are plotted for the four configurations. As can be seen, Case I exhibits basically identical performance to the baseline, while Case II improves 18% on communication latency (and thus 8% on execution time) and Case III improves 24% on latency (and thus 11% on execution time). Therefore, Case III is the best among the alternatives under evaluation.

These experiments highlight that RIPE can be an extremely useful tool to explore communication bottlenecks even without having the real IP cores and benchmarks attached to the fabric. The flexibility guaranteed by the interrupt handling support provides the designer with additional degrees of freedom and accuracy, allowing a realistic system exploration even in presence of complex communication and synchronization patterns.

IX. Conclusions

In this paper, we identified the requirements to split the design of computation and communication entities in an MPSoC. Modeling requirements were derived from real-life applications, and they represent complex scenarios including an operating system layer and asynchronous interrupt-based synchronization. The key piece of the puzzle can be identified in reactiveness to external events and state. In this paper, we presented the RIPE device and its programming interface to provide support for the previously identified traffic generation functionality.

We have shown the usefulness of the RIPE device within different co-exploration domains, either to replace existing IP cores in new domains or to provide emulation of IP cores that are under development or even yet to be designed.

Experimental results show excellent accuracy figures when validating the RIPE against a reference system, and a respectable gain in simulation speed when taking into account previous literature and the cycle-accurate abstraction level. A case study is supplied to show the usefulness of RIPE in a design space exploration context.

Future work may carry the current RIPE design to silicon for on-chip traffic generation.

X. Acknowledgments

The authors from University of Bologna acknowledge financial support by Semiconductor Research Corporation (SRC) under contract 1188.

References

[1] The Advanced Microcontroller Bus Architecture (AMBA) homepage. www.arm.com/products/solutions/AMBAHomePage.html.

[2] The SystemC discussion forum. Web Forum (www.systemc.org).

[3] The Real-Time Operating System for Multiprocessor Systems. http://www.rtems.com.

[4] Open Core Protocol Specification, Release 2.0, 2003.

[5] IEEE, March 2005.

[6] F. Angiolini, S. Mahadevan, J. Madsen, L. Benini, and J. Sparsø. Realistically rendering SoC traffic patterns with interrupt awareness. In *IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*, September 2005.

[7] ARM. AMBA AXI Protocol Specification, version 1.0. www.arm.com, March 2004.

[8] ARM Holdings PLC. Advanced Microcontroller Bus Architecture (AMBA) specification rev 2.0, 2001.

[9] S. Avallone, A. Pescape, and G. Ventre. Analysis and experimentation of internet traffic generator. In *Proceedings of FTDCS*, 2004.

[10] L. Benini and G. D. Micheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70 – 78, January 2002.

[11] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. In *Journal of Systems Architecture*. Elsevier, 2004.

[12] L. Cai and D. Gajski. Transaction level modeling in system level design. CECS technical report 03-10, Center for Embedded Computer Systems, Information and Computer Science, University of California, Irvine, March 2003.

[13] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. xpipes: A latency insensitive parameterized Network-on-Chip architecture for multi-processor SoCs. In *Proceedings of the International Conference on Computer Design (ICCD)*. IEEE Computer Society, 2003.

[14] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference*, pages 684–689, June 2001.

[15] F. Fummi, P. Gallo, S. Martini, G. Perbellini, M. Poncino, and F. Ricciato. A timing-accurate modeling and simulation environment for networked embedded systems. In *Proceedings of the 42th Design Automation Conference (DAC)*, pages 42–47, June 2003.

[16] F. Fummi, S. Martini, G. Perbellini, M. Poncino, F. Ricciato, and M. Turolla. Heterogeneous co-simulation of networked embedded systems. In *Proceedings of Design, Automation and Testing in Europe Conference 2004 (DATE)*. IEEE, Febuary 2004.

[17] N. Genko, D. Atienza, G. D. Micheli, L. Benini, J. M. Mendias, R. Hermida, and F. Catthoor. A novel approach for network on chip emulation. In *International Symposium on Circuits and Systems*, pages 2365–2368. IEEE, 2005.

[18] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[19] K. Lahiri, A. Raghunathan, and S. Dey. Evaluation of the traffic-performance characteristics of System-on-Chip communication architectures. In *Proceedings of the 14th International Conference on VLSI Design*, pages 29–35, 2001.

[20] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. IEEE, 2004.

[21] S. Mahadevan, F. Angiolini, M. Storgaard, R. G. Olsen, J. Sparsø, and J. Madsen. A network traffic generator model for fast network-on-chip simulation. In *Proceedings of Design, Automation and Testing in Europe Conference 2005 (DATE)* [5].

[22] O. Ogawa, S. B. de Noyer, P. Chauvet, K. Shinohara, Y. Watanabe, H. Niizuma, T. Sasaki, and Y. Takai. A practical approach for bus architecture optimization at transaction level. In *Proceedings of Design, Automation and Testing in Europe Conference 2004 (DATE)*. IEEE, March 2003.

[23] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proceedings of 38th Design Automation Conference (DAC)*, pages 113–118. ACM, 2004.

[24] S. Schneider, U. Mueller, and D. Tiegelbekkers. A reactive workload generation framework for simulation-based performance engineering of system interconnects. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, September 2005.

[25] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the System-on-Chip interconnect woes through communication-based design. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 667 – 672, June 2001.

[26] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, and G. D. Micheli. ×pipes Lite: A synthesis oriented design library for networks on chips. In *Proceedings of Design, Automation and Testing in Europe Conference 2005 (DATE)* [5], pages 1188–1193.

[27] STMicroelectronics. The ST Bus. http://www.st.com/stonline/, 2004.

[28] Synopsys. OpenVERA Technology Backgrounder. White paper available from http://www.open-vera.com/, 2001.

[29] G. V. Varatkar and R. Marculescu. On-chip traffic modeling and synthesis for MPEG-2 video applications. In *Transcations on Very Larget Scale Integration (VLSI) Systems*, number 1, pages 108–119. IEEE, JANUARY 2004.

[30] D. Wiklund, S. Sathe, and D. Liu. Network on chip simulations for benchmarking. In *Proceedings of the 4th IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC)*. IEEE, 2004.

[31] W. Wolf. *Computers as Components:Principles of Embedded Computing System Design*, chapter 3. Morgan Kaufmann, 2001.

Part III

Appendix

Chapter 6

Network-on-Chip Modeling for System-Level Multiprocessor Simulation

Published in the Proceedings of the 24th Real-Time Systems Symposium 2003.

Complete citation:

Jan Madsen, Shankar Mahadevan, Kashif Virk and Mercury Gonzalez, "Network-on-Chip Modeling for System-Level Multiprocessor Simulation." *In Proceedings of the 24th Real-Time Systems Symposium (RTSS), Cancun Mexico.* IEEE, Dec. 2003: 265-274.

Network-on-Chip Modeling for System-Level Multiprocessor Simulation*

Jan Madsen Shankar Mahadevan Kashif Virk Mercury Gonzalez
Informatics and Mathematical Modeling
Technical University of Denmark
{jan, sm, virk}@imm.dtu.dk

Abstract

With the increasing number of transistors available on a single chip, the System-on-Chip (SoC) paradigm has evolved to exploit its full potential. As many processors can be accommodated on a single chip, this paradigm has forced a communication-centric, as opposed to a computation-centric, design view. Thus, the choice, management and modeling of the SoC interconnect is essential for an accurate evaluation and optimization of the global performance of a system. Recently, the notion of Network-on-Chip (NoC) has been introduced as a way to extend the classical bus-based interconnection, which is still the dominant interconnect structure for SoC's, into a dedicated, segmented and, possibly, packet-switched network fabric [2]. In this paper, we present a NoC model which, together with a multiprocessor real-time operating system (RTOS) model, allows us to model and analyze the behavior of a complex system that has a real-time application running on a multiprocessor platform. We demonstrate the potential of our model by simulating and analyzing a small multiprocessor system connected through different NoC topologies, and discus how the simulation model may be used during the design-space exploration phase.

1. Introduction

With the growing complexity of embedded systems and the capacity of modern silicon technology, there is a trend towards heterogeneous architectures consisting of several programmable and dedicated processors, implemented on a single chip, known as a System-on-Chip (SoC). As an increasing portion of applications is implemented in software which, in turn, is growing larger and more complex, dedicated operating systems will have to be introduced as an interface layer between the application software and the hardware platform [5]. On the other hand, the hardware platform will either be developed as a part of the design process or configured from an existing reconfigurable platform, which allows for the implementation of parts of an application as dedicated processors (ASIC's).

Modern silicon technologies, with minimum device geometries in the nanometer range (<100nm), have made it possible to integrate hundreds of processors on a single chip. In these deep submicron technologies, the on-chip interconnection fabric is a major source of delay and power consumption which is challenging the on-chip communication infrastructure and forcing a change from device-centric to interconnect-centric design methodologies. Traditionally, on-chip communication has either been conducted via dedicated point-to-point links or by shared media like a bus. Neither is very suitable for generalized communication handling in large systems [13]. A promising solution is to have a dedicated, segmented, and, possibly, packet-switched network fabric on the chip, a Network-on-Chip (NoC) [2].

Hence, when mapping an application onto its target platform, hardware/software codesign aspects [18] have to be taken into account. These include mapping of tasks onto software, hardware, or a combination of both, as well as task dependencies on the communication infrastructure. In order to do so, accurate modeling of the systems and all the interrelationships among the diverse processors, software processes and physical interfaces and interconnections, is needed. One of the the primary goals of system-level modeling is to formulate a model within which a broad class of designs can be developed and explored. To support the designers of single-chip based embedded systems, which includes multiprocessor platforms running dedicated real-time operating systems (RTOS's) as well as the effects of on-chip interconnect network, a system-level modeling/simulation environment is required to support an analysis of the:

- consequences of different mappings of tasks to processors (software or hardware),
- network performance under different traffic and load conditions,
- effects of different RTOS selections, including vari-

*Funded by SoC-Mobinet (IST-2000-30094).

ous scheduling, synchronization and resource allocation policies.

In this paper, we present a modeling environment based on SystemC [22] which can provide the SoC designers a software-like, system-level abstraction of the platform as well as supporting the three requirements mentioned above for system-level design-space exploration.

Most of the future embedded applications are likely to be real-time applications that will run on multiprocessor SoC's which are, essentially, distributed computing systems. In a multiprocessor or a distributed system, the processing elements can be connected through shared memory, dedicated communication links or a communication network. Instead of dealing with each specific application and system architecture, we deal with generalized abstract tasks, processing elements, and communication infrastructures. This not only broadens the applicability of our modeling framework, but also leads to a better understanding of the problem at hand.

We extend our previous work [9, 16] on the modeling of a multi-threaded application, running on a multiprocessor platform under the control of one or more abstract RTOS's, with a model of an on-chip network which can provide provisions for run-time inspection and observation of the on-chip communication. Using this system-level design approach, implementations of the most promising network alternatives can be prototyped and characterized in terms of performance and overhead. Taking communication into account during hardware/software mapping is essential in order to obtain optimized solutions as emphasized in [14].

The paper is organized as follows: Section 2 describes current trends and related work in the field of communication network modeling for multiprocessor environments. In Section 3, we provide a brief overview of our previously proposed RTOS model and discuss its extension to include the NoC model. Section 4 presents our main ideas on NoC modeling. It provides the methodology for developing a network model for usage at the system-level. This model seamlessly handles the allocation and scheduling of communication events within the NoC as driven by the requirements from the tasks running on the PE's in a SoC. A SystemC implementation of a torus network is also discussed. The results of our implementation and simulation of the model are given in Section 5. Further, in Section 6, we extend this discussion to the effects of select design-space exploration choices on global system performance. Section 7, finally, provides conclusions and the future direction of our work.

2. Related Work

One of the essential elements of making a transition from ad-hoc system-on-chip (SoC) designs to a disciplined SoC design approach is taking a rigorous, though flexible, approach towards the design of on-chip communication networks that interconnect IP blocks of all variety, including the processing elements (PE's). A network-on-chip (NoC) approach, driven by a consistent design methodology, is bound to lead to dramatic changes in how SoC's will be designed in the future. The partitioning and mapping of tasks onto complex architectures (homogeneous or heterogeneous) is a well-known hardware/software codesign problem [18]. [8, 12, 16, 17, 18, 19] further explain allocation, scheduling and synchronization in RTOS's. But the notion of the on-chip communication medium has been quite primitive. It has, generally, been viewed as an overhead where no other useful work can be accomplished. Thus, it is assumed to occur instantaneously or it is given a token fixed overhead time. This approach is suboptimal and error-prone requiring further iteration before design closure. [12] and [14] clearly show the importance of evaluating the communication media and how the choice of a communication architecture clearly impacts the overall architecture of a SoC. In [1], a communication model for codesign has been described, but it is limited and cannot account for specific NoC features for design-space exploration at the system level.

There, already, exists plenty of research literature on the communication modeling for multiprocessors with different interconnection topologies to characterize their communication performance, for example [3]. Moreover, in [2] and [23], the concept of on-chip, packet-switched micro-networks has been introduced that borrows ideas from the layered design methodology for data networks. In [15] the layered, packet-switched NoC design concepts have been applied to a 2-D Mesh Network Topology whereas in [10], similar concepts have been applied to a Butterfly Fat Tree Topology. While there are several mature methodologies for modeling and evaluating the processing element architectures, there is relatively little research done to port the on-chip communication to system-level. In [24], attempts have been made to fill this gap by proposing a NoC modeling methodology based upon the ideas borrowed from the object-oriented design domain and implementing those ideas using an existing CAD framework – Ptolemy II. However, the authors have conjectured about the performance gains achievable by the porting of their proposed modeling framework to SystemC. In [21], a theoretical framework for modeling real-time applications running on multiprocessor systems has been developed that models the inter-processing element communication with a link processor. But such attempts are quite ad-hoc and no generalized approach has, so far, been reported to our knowledge.

In our proposed abstract system modeling framework, an embedded, real-time application is represented as a collection of multiple, concurrent execution threads that are modeled as a set of dependent tasks under certain precedence and resource constraints. Such tasks, in turn, are modeled as a chain of sub-tasks executing on, possibly, different processing elements. Based on the abstract system model, three distinct, but closely-related problems are identified, namely, execution synchronization, resource allocation and priority assignment/scheduling. The inter-processing element communication is modeled by modeling a communication network as a communication processor and the message transmission through the network as a communication task running (concurrently) on the communication processor. Using this approach, we have demonstrated that our, previously proposed [9, 16], abstract RTOS model can be extended to include an abstract NoC processor that can effectively model the system-level effects of any NoC architecture.

3. Abstract RTOS Modeling

As discussed earlier, at the system level, the application software may be modeled as a set of tasks which have to be executed on a number of processing elements (PE's) under the control of one or more RTOS(s). For details on the model and how it is implemented in SystemC (including the use of the Master-Slave library), we refer to [9] and [16].

Briefly, our system model is designed following the principle of composition, as described in [20], and consists of three types of basic components: tasks, RTOS services, and links, where the links provide communication between other system components. We have used SystemC 2.0 as the implementation language of our model. Although, any language could have been used, the choice of SystemC is mainly due to the fact that it is an extension of the C++ programming language and has a built-in simulation kernel that supports concurrency. In addition, it supports the design process from system-level down to both hardware and software implementations. The SystemC Master-Slave library provides a very elegant way of handling concurrent messages sent by the tasks to the RTOS services. This allows each RTOS service to deal with a single message at a time independently of the other. Figure 1 shows the Abstract RTOS Model and Figure 2 presents the overall system model, including the NoC model which will be described in the next section. In this section, we focus on the RTOS modeling which corresponds to the PE's. The RTOS services are composed from independent modules that model different basic RTOS services. A scheduler models a real-time scheduling algorithm. A synchronizer models the dependencies among tasks and, hence, both intra- and inter-processing element communications. An allocator models the mechanism of resource sharing among tasks.

The model is designed such that any of the RTOS services can be changed in a simple and straight forward manner. Tasks are considered to be abstract representations of

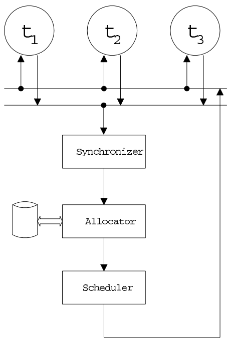


Figure 1. *Abstract RTOS model.*

the application and are characterized by a set of parameters, such as the worst- and the best-case execution time, context switching overhead, deadline, period (if it is a periodic task), offset, resource requirements, and precedence relations. A task is modeled as a finite state machine (FSM) which can send the messages: `ready` and `finished`, to the scheduler which, in turn, can send one of the three commands to the tasks: `run`, `preempt`, and `resume`. In between the schedulers and the tasks, we have the synchronizer and the allocator acting as "logical command filters". As a way to maintain composition, each module handles its relevant data independently of the other. For example, a task determines when it is ready to run and when it has finished. In this way, the scheduler behaves in a reactive manner; scheduling tasks according to the data received from them. Thus, we can add as many tasks and schedulers as we desire. The same is the case with the synchronizer and the allocator models. They hold the information regarding their services, i.e., which tasks depend on each other or, for the case of the allocator, what resources are needed by a given task.

4. NoC Modeling

Architecturally, a network is characterized by its *topology* and the *protocol* running on it. The topology concerns the geometry of the communication links on the chip while the protocol governs the usage of these links. Many combinations of topology and protocol exist for the efficient communication of one or more predominant traffic patterns. The *performance* of a network is measured in quantitative terms such as latency, bandwidth, power consumption and area usage, and in qualitative terms such as network reconfigurability (dynamic or static), quality of service (QoS), etc.

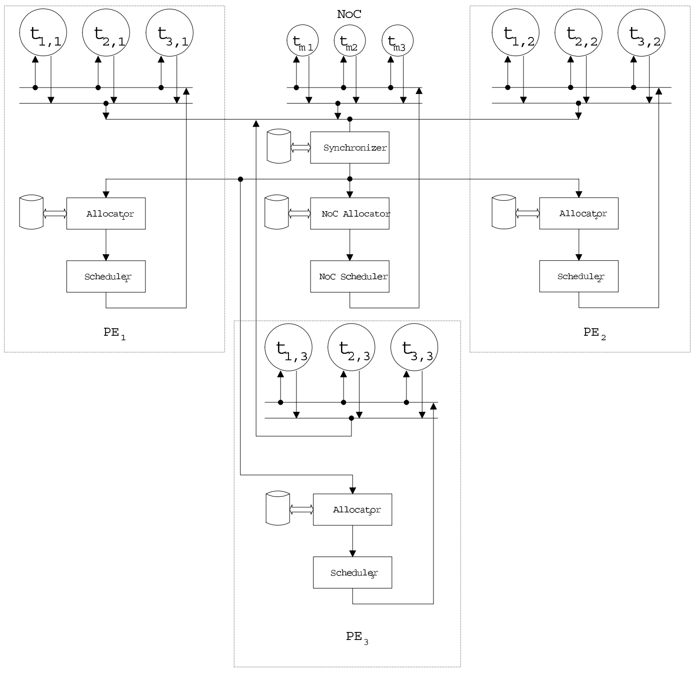


Figure 2. *The Network-on-Chip model.*

Predictability of performance is necessary for NoC designers to take early decisions based on the NoC performance before actual implementation. Numerous studies have been done for deadlock, livelock, congestion-avoidance, error-correction, network setup/tear-down, etc. to provide a certain predictable network behavior [7]. Even lower-level engineering techniques like low-swing drivers, signal encoding etc., have been proposed to overcome network communication uncertainties [4, 6, 11]. Many of these aspects are custom-tuned to fit the requirements of the underlying application.

Throughout this paper, we use *network latency* as a primary factor for grading the performance of a network. The network latency is defined as the time taken to move data from a source PE to a destination PE. It includes the message processing overhead at the PE's, link delays and the data processing delays at the intermediate nodes [14]. It is a function of the topology (which determines the number of nodes and links) and the protocol (which defines the processing requirements for routing and flow-control).

The *state* of a network at any instant is given by the number of actively transmitting PE's and the messages within its nodes and links. The state of a network dictates which resources of the network are currently in use and which ones can be available for future use. This provides a measure of the *network services* available to the system, which affect its performance. We define network services as the system-level characterization of network resource allocation and scheduling. For a given topology-protocol combination, changes in network services, change the resources available for a given communication event, thus, affecting its latency.

For the purpose of forming a system-level NoC simulation model, unlike a network simulator, we have abstracted away all the above-mentioned low-level network details except the most essential ones (e.g., topology, latency, etc.). We treat the on-chip communication network as a *communication processor* to reflect the servicing demands. A communication event within this network is modeled as a *message task*, τ_m, executing on the communication processor.

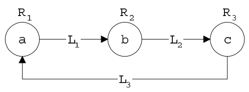
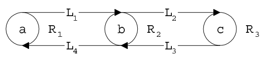
Message Task	Path					
		Torus			Mesh	
		Resource Allocation	Scheduling Needs		Resource Allocation	Scheduling Needs
			Small Message Size	Large Message Size		Small or Large Message Size
τ_{mx}	a→b	L_1	Immediate	Preemptive	L_1	Immediate
τ_{my}	c→b	L_3, R_1, L_1	Immediate	Immediate	L_3	Immediate

Table 1. *A sample reservation for two sample networks.*

When one PE wants to communicate with another PE, a τ_m is fired on the communication processor. Each τ_m represents communication only between two fixed set of predetermined PE's. Since a NoC supports concurrent communication, τ_m's need to be synchronized, allocated resources and scheduled accordingly. This is a property of the underlying NoC implementation, where the NoC allocator reflects the topology and the NoC scheduler reflects the protocol. A resource database, which is unique to each NoC implementation, contains information on all its resources. In a segmented network, these resources are laid-out as two-dimensional interconnects and are a collection of nodes (routers) and links. The NoC allocation and scheduling algorithms map a τ_m onto the available network resources. Here, we mainly illustrate this for the networks which allow parallel communication to occur, such as the segmented networks.

4.1 NoC Allocator

The allocator translates the path requirements of a τ_m in terms of its resource requirements such as bandwidth, buffers, etc. It attempts to minimize resource conflicts. The links and nodes in a communication path are set aside dynamically (i.e., only for the requested time slot) in the resource database. If the resource reservation process is successful, the message task is queued for scheduling. The resource allocation for two sample networks is shown in Table 1. If there is a contention over a resource, then resource arbitration occurs. The arbitration mechanism is based on the underlying network implementation and is discussed shortly. In this discussion, the resources are regarded as non-preemptable. Therefore, a resource is free to be assigned to another τ_m only after the τ_m, which is already occupying that resource, has released it.

4.2 NoC Scheduler

The NoC scheduler executes the τ_m's according to the particular network service requirements. It attempts to minimize resource occupancy. In a network, resource occupation is dictated by the size of the message. This concept is better illustrated using the example in Table 1, where the scheduling needs for two sample networks are shown. For a mesh there is no resource conflict. The τ_m's get the required resources allocated 'immediately'. But in the case of a torus, it might experience a resource conflict for the link L_1. Here, in the event of a small message size, where τ_{mx} is finished before τ_{my} asks for L_1, there is no scheduling problem. The resources can be 'immediately' assigned to the τ_m's. But in the case of a large message size, where τ_{mx} is still running when τ_{my} asks for the link L_1, resource contention occurs. Thus, the scheduling of the messages has to be performed preemptively.

Let us consider the above example from the points of view of the network-designer and the system-designer. At the network-level, seeing the resource conflict as a network problem, the network designer may over-design link L_1 by providing excess bandwidth or introduce processing overhead, such as TDM-based message interleaving. These techniques would restore fair servicing for both the τ_m's, reducing the degree of contention. However, at the system-level, it may be possible to reschedule the communication event between the PE's (either τ_{mx} or τ_{my}). This opens up the possibility of an alternate path assignment for the τ_m's or simply stalling one of the traffics until the other has passed. System designers may even realize that large message sizes (to the extent where L_1 is contentious) never occur within the system. This could save potential scheduling/computation overhead in terms of hardware real-estate, power, etc. at router R_1 and on link L_1 as was envisioned by the network designer. Thus, when seen from the system-level, a trade-off between the NoC resource allocation and scheduling would not only complement better self-utilization, but might provide other useful insights for design improvements. Towards this, we implement a NoC model for system-level evaluation.

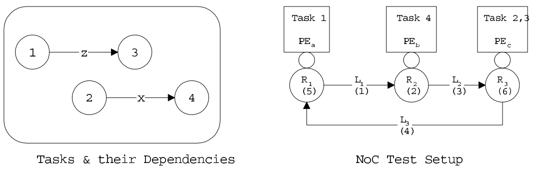


Figure 3. *System simulation model.*

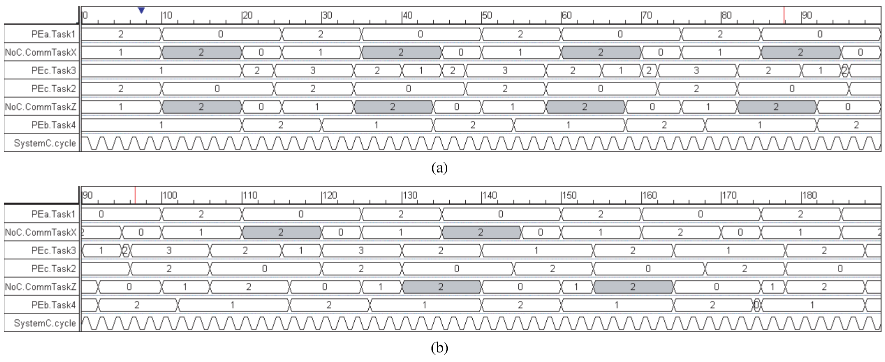


Figure 4. *Simulation results for communication events. State enumeration: 0=inactive, 1=ready, 2=running, 3=preempted.*

4.3 Implementation

The NoC model has exactly the same structure as the abstract RTOS model but with some modifications to its constituent module blocks. The main idea while implementing the NoC model was to preserve the existing structure of the abstract RTOS framework and to reuse the existing code fragments as much as possible so that no extra complexity is added and the code size does not grow too much so as to compromise the simulation speed. The message routing scheme currently implemented in our NoC model is that of fixed routing but the framework does have provisions for implementing other routing schemes.

4.3.1 Message Task

The message task has the same FSM structure as the Task model in the abstract RTOS model with some modifications to take out preemption and introduce resource requirements. The τ_m implementation accepts a number of arguments for its characterization. The *Message Task ID* enables the Synchronizer and the NoC Scheduler to identify the τ_m sending the message. Similarly, the *NoC Scheduler ID* is meant for the τ_m's to recognize their scheduler for exchanging various control messages. The lower- and the upper-bounds on the transmission latency of an τ_m through the NoC are defined by the *BCET (Best-Case Execution Time)* and the *WCET (Worst-Case Execution Time)*. If a message task has a certain setup time before it is released, then its *offset* is nonzero. A list of resources (links, routers, etc.) required by a τ_m during its execution is furnished in the form of *Resource ID's* and the time durations for holding those resources are specified as *CSL's (Critical Section Lengths)*. The implementation of a τ_m can be viewed as a FSM that manages various counters after sending messages to the NoC Scheduler and the NoC Allocator and upon receiving commands from the NoC Scheduler.

4.3.2 NoC Allocator

The NoC Allocator manages its resource database upon receiving `request` and `release` messages from the τ_m's. The resources are allocated to the τ_m's dynamically and they are released by the τ_m's immediately after usage. This makes resource management very flexible. In this implementation, the resources are served by the NoC Allocator on a first-come-first basis but other allocation policies can be implemented as well. Whenever a requested resource is available, the NoC Allocator sends a `grant` message to the NoC Scheduler and whenever a requested resource is occupied, there is a resource contention and the NoC Allocator sends a `refuse` message to the NoC Scheduler for an appropriate action.

4.3.3 NoC Scheduler

The NoC Scheduler receives the `ready` and `finished` messages from the τ_m's through the Synchronizer and the `grant` and `refuse` messages from the NoC Allocator. It then issues the `run` and `buffer` commands to the τ_m's. Whenever a Task running on a PE, is finished and needs to communicate with a Task running on another PE, it sends a `finished` message to the Synchronizer which maintains a task dependency database and passes the `ready` message for the corresponding τ_m to the NoC Scheduler which issues the `run` command to that τ_m.

Whenever there is a resource contention, the NoC allocator issues a `refuse` message to the NoC Scheduler which then either terminates the execution of the requesting τ_m (equivalent to message dropping) or blocks the τ_m from execution (equivalent to message buffering) till the requested resource becomes available again which is indicated by the `grant` message sent by the NoC Allocator to the NoC Scheduler. The message dropping or buffering decision is taken by the NoC Scheduler according to its underlying network implementation.

5. Results

The results of our SystemC implementation of the NoC model from Figure 2 are presented in Figure 4 and Figure 5 and illustrated in Figure 6. The sample SoC-NoC setup is shown in Figure 3. The application is assumed to have been decomposed into four tasks (τ_1, τ_2, τ_3, and τ_4). Three PE's (PE_a, PE_a, and PE_a) are selected to execute these tasks. The task mappings are: $\{\tau_1\} \mapsto PE_a$, $\{\tau_4\} \mapsto PE_b$, and $\{\tau_2, \tau_3\} \mapsto PE_c$. τ_2 has a higher priority than τ_3, so it can preempt τ_3 on PE_c. In this example, we look at a simple case where all the tasks are modeled identically with a period of 25 time units (except for τ_2, which has a period of 24 time units due to the priority-assignment scheme in the Rate Monotonic scheduling), an execution time (both BCET and WCET) of 10 time units and a deadline of 22 time units.

The communications between the tasks are modeled as τ_m's (as described in Section 4) which execute on a communication processor simulating a torus network using the store-and-forward routing protocol [7] (with infinite buffer at the source and the destination nodes). The message task paths and dependencies are: τ_{mx}, from PE_a to PE_c using L_1, R_2, and L_2, and τ_{mz}, from PE_c to PE_b using L_3, R_1 and L_1. Thus, the link L_1 experiences a possible contention. In

```
 0 Initializations
10 CommTask X Released by the Synchronizer
10 CommTask Z Released by the Synchronizer
11 task x (request resource# 1)-> allocator
11 NoC_allocator (granted)->NoC_scheduler
11 task z (request resource# 4)-> allocator
11 NoC_allocator (granted)-> NoC_scheduler
14 task x (release resource# 1)-> allocator
14 task x (request resource# 2)-> allocator
14 NoC_allocator (granted)-> NoC_scheduler
14 task z (release resource# 4)-> allocator
14 task z (request resource# 5)-> allocator
14 NoC_allocator (granted)-> NoC_scheduler
17 task x (release resource# 2)-> allocator
17 task x (request resource# 3)-> allocator
17 NoC_allocator (granted)-> NoC_scheduler
17 task z (release resource# 5)-> allocator
17 synchronizer (release)-> allocator
17 task z (request resource# 1)-> allocator
17 NoC_allocator (granted)-> NoC_scheduler
20 task x (release resource# 3)-> allocator
20 task x (finished)-> scheduler 2
20 synchronizer (finished)-> allocator
20 NoC_allocator (finished)-> NoC_scheduler
20 task z (release resource# 1)-> allocator
20 task z (finished)-> scheduler 2
20 synchronizer (finished)-> allocator
20 NoC_allocator (finished)-> NoC_scheduler
   and so on...
```

Figure 5. *Simulation log.*

our SoC-NoC test setup, the resource ID is given in brackets (next to the resource label in Figure 3). We present two cases of interest:

In Figure 4(a), modeling of two concurrent communications is shown. As mentioned earlier, there is a link contention between τ_{mx} and τ_{mz} for L_1. It is resolved by scheduling L_1 at different times among the τ_m's within the time-slot of 10 to 20 time units (and subsequent time slots). L_1 is used from 11 to 14 time units in τ_{mx} and from 17 to 20 time units in τ_{mz}. Figure 5 shows the log file of resource occupancy (Resource# 1 is link L_1). Figure 6 provides a graphical representation (Note that 1 time unit is consumed in the network setup during simulation). Thus, our model clearly supports concurrent communication as observed in segmented networks.

Figure 4(b) shows the interplay of process modeling and interconnect activity. Consider the signal titled *PE_c Task 3* (τ_3) in Figure 4(b) at a point close to the time period of 95 time units. Here, it is clear that τ_3 starts accepting the communication message and is then preempted by τ_2 on PE_c because of its higher priority. Once τ_2 is finished, τ_3 resumes and completes in time (at time 120) before its deadline. Now consider the next execution of τ_3. Both τ_2 and τ_3 are in contention. τ_3 does not even start; instead, τ_2 starts on the PE_c. τ_3, here, is not able to accept the message com-

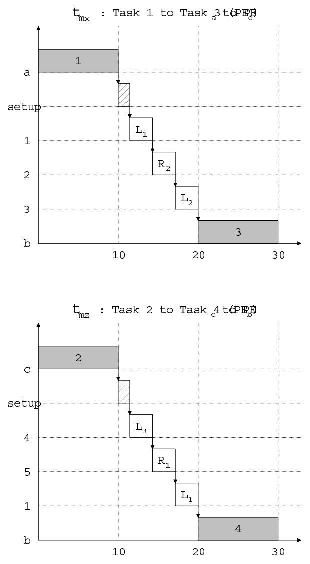

t_{mx} : Task 1 to Task 3 (PE_a to PE_c)
a
setup
1
2
3
b
1
L_1
R_2
L_2
3
10
20
30
t_{mz} : Task 2 to Task 4 (PE_a to PE_b)
c
setup
4
5
1
b
2
L_3
R_1
L_1
4
10
20
30

Figure 6. *NoC allocation and scheduling for the first communication cycle.*

municated to it by τ_1. This brings us to an interesting role of the NoC. In this simulation, we have enabled the routers to be able to buffer messages. Thus the τ_{mx} finishes freeing up its resources although τ_2 has yet to begin. τ_3, when finished, is thus able to initiate τ_{mz}, which is when τ_2 resumes.

Consider the case where the same torus network processor is running wormhole routing (plots not provided). Then, in the preemption case, the τ_{mx} stalls, holding the link L_1. As τ_2 has already preempted τ_3 on PE_c, when it is complete, it would attempt τ_{mz}. But this would not be possible as the link L_1 required here is busy in τ_{mx}, thus stalling τ_{mz}. This causes deadlock in the system. As seen earlier, we can resolve it either by introducing buffering in the routers or we have the freedom to choose an alternate network implementation or scheduling strategy. Thus, even this simple example clearly demonstrates the global performance evaluation for codesign when both SoC and NoC are jointly modeled.

6. Design-Space Exploration

Figure 7 illustrates how our proposed NoC model can be used for design-space exploration at the system level. We have used three sample network topologies: torus, mesh, and bus. The assignment of tasks to the PE's are: $\{\tau_1, \tau_2\} \mapsto PE_a$, $\{\tau_3\} \mapsto PE_b$, and $\{\tau_4, \tau_5\} \mapsto PE_c$. All the tasks have the same period, execution time (BCET=WCET) and a deadline of 100, 15 and 100 time units, respectively. It is assumed that the tasks are mapped on the PE's in such a way that none of them misses its deadline. The task dependencies are: $\tau_3 \prec \{\tau_1, \tau_4\}$ and $\tau_5 \prec \tau_2$. The dependencies for the tasks mapped onto different PE's translate into τ_m's as described in Section 4. In this illustration, we have labeled them as x, y, and z. The link and the node utilization for each corresponding topology-protocol combination alters for these τ_m's. For simplicity, we model all link occupancies to be 10 time units and node processing times to be 2.5 time units. Besides, the task and the communication model, in this analysis we have also included the time spent at the network interface for message transfer from the PE's to the NoC. This is assumed to be about 3 time units. It is incurred twice, once at the source and then at the destination, for each communication event.

The three rows in Figure 7 show the network performance for three different scheduling-architecture combinations. The performance of the system is judged by its scheduling. In the first row, basic timing-aware scheduling is illustrated. Here, the networks are quite primitive, i.e., the link contention is resolved randomly. The best-effort scheduling for the torus network and the bus consumes about 80 time units. The mesh network utilizes 65 time units. The bus is a singular entity and, hence, the NoC allocator does not have much freedom in its allocation. The scheduling of the communication is, therefore, sequential. On the other hand, the torus and the mesh networks have multiple ways to allocate and schedule their resources. As the example is relatively small, therefore, the full potential of concurrent communication is not obvious for the torus network. But it is obvious for the mesh network. It is about 10 time units less than the other networks. Regarding the link utilization[1], both for the torus and the mesh networks, one link each is not used in this architectural setup (L_1 for the torus and L_4 for the mesh network). Thus, if the system is not under the constraints of meeting the timing-bounds, a possible network optimization exists. On the other hand, in the torus network, if τ_1 and τ_4 are scheduled together, there is a contention on link L_1, so a network optimization to meet the timing-bounds is required.

In the second row of Figure 7, we illustrate one possible network optimization, namely, the effects of source-based QoS routing. Any traffic from PE_a is considered to have a higher priority and, hence, is assigned the contentious resource (when necessary). For a mesh network, there is no effect as the link occupancy is not in conflict. But consider its effect on the torus network. It gives about 5 time units

[1]Link Utilization is defined as the aggregation of the number of links occupied in the smallest time unit

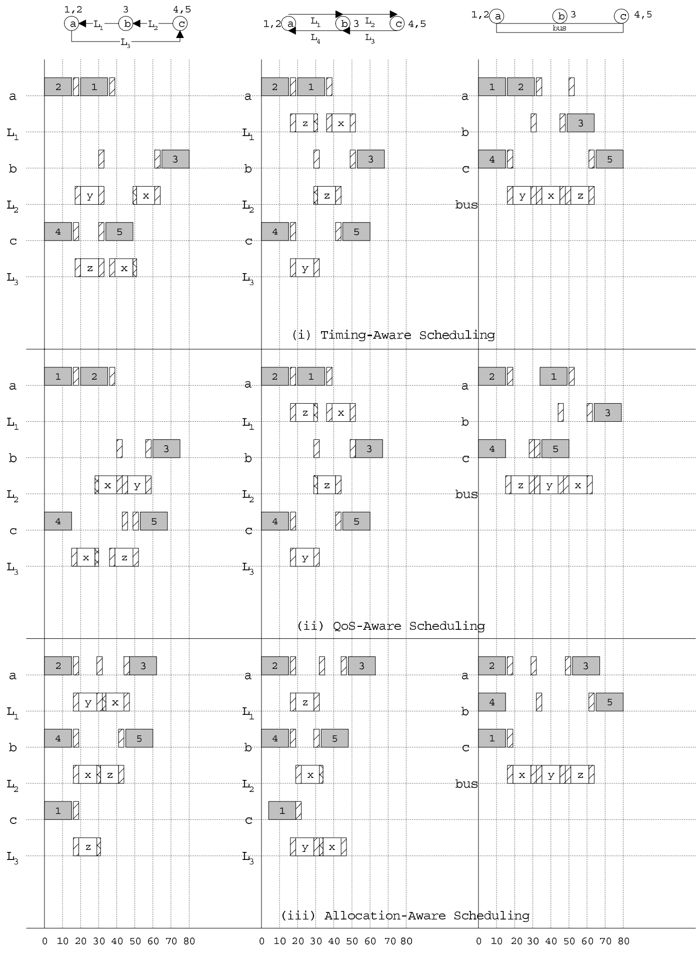


Figure 7. *Illustration of the system-level design-space exploration.*

better performance than the regular torus network. For a complex system with multiple links and nodes and handling numerous messages, these advantages are expected to be significant (both for torus and mesh). The bus architecture, on the other hand, would become a bottleneck in communication.

Having looked at how a manipulation of the network affects the overall performance, at the system-level, one can even expect to change the allocation of tasks based on the network choice. This is illustrated in the last row. The new allocation under consideration is: $\{\tau_2, \tau_3\} \mapsto PE_a$, $\{\tau_4, \tau_5\} \mapsto PE_b$, and $\{\tau_1\} \mapsto PE_c$. The advantage in terms of overall system execution time is considerable for the segmented network compared to the bus. The reasons for the poor performance of the bus are the same as the ones stated earlier. In the case of the torus and the mesh networks, the link utilization is high now. Many links, though not all, are used simultaneously without any contention. We have not considered QoS assignment in this case, but its effect on performance, especially, in a large system might be considerable.

Using these illustrations, similar analysis for memory and power utilization can be easily performed as well. There are many possibilities of trade-offs during each iteration; namely to change the resource requirements, resource allocation, or scheduling. The overall idea is to assist the codesign process to converge while satisfying the desired performance criteria.

7. Conclusions

We have presented an abstract modeling framework based on SystemC which supports the modeling of multiprocessor-based RTOS's and their interconnection through a NoC. The aim is to provide the system designer of single-chip, real-time embedded systems with a simple modeling and simulation framework in which one can experiment with different task mappings, RTOS policies and NoC structures and protocols in order to study the consequences of local decisions on the global system behavior and performance. We have presented how our initial multiprocessor RTOS model has been extended to handle NoCs. So far, our experimental work has been aimed at providing a proof-of-concept as demonstrated in Section 5. We are currently working on extending the NoC model to incorporate issues like, dynamic path routing, packet switching and power profiling. We are also working on a few large real-life examples as well as a schedule viewer based on the output from the monitors which will provide detailed and annotated views of the system behavior such as detailed network usage and power- and memory-profiles.

References

[1] A. Baghdadi and N-E. Zergainoh. Design Space Exploration for Hardware/Software Codesign of Multiprocessor Systems. In *Proceedings of the 11th International Workshop on Rapid System Prototyping (RSP)*, pages 8 – 13, 2000.

[2] L. Benini and G. D. Micheli. Network on Chips: A New SoC Paradigm. *IEEE Computer*, 35(1):70 – 78, January 2002.

[3] S. H. Bokhari. Communication Overhead on the Intel Paragon. NASA Contractor Report 1982(11), NASA Langley Research Center, September 1995.

[4] W. Brainbridge and S. Furber. Delay Insensitive System-on-Chip Interconnect using 1-of-4 Data Encoding. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 118 – 126, 2001.

[5] A. S. Cassidy, J. M. Paul, and D. E. Thomas. Layered, Multi-Threaded, High-Level Performance Design. In *Design Automation and Test in Europe, DATE*, pages 954–959, March 2003.

[6] J. Cong. An Interconnect-Centric Design Flow for Nanometer Technologies. In *International Symposium on VLSI Technology, Systems, and Applications*, pages 54 – 57, 1999.

[7] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan-Kaufmann, 1998. 1st edition.

[8] A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modelling for System-Level Design. In *Design Automation and Test in Europe, DATE*, pages 132–137, March 2003.

[9] M. J. Gonzalez and J. Madsen. Abstract RTOS Modeling in SystemC. In *Proceedings of the 20th IEEE NORCHIP Conference*, pages 43 – 49, November 2002.

[10] P. Guerrier and A. Greiner. A Generic Architecture for On-Chip Packet-Switched Interconnections. In *Design Automation and Test in Europe, DATE*, pages 250 – 256, March 2000.

[11] R. Ho and K. W. Mai. The Future of Wires. *Proceedings of the IEEE*, 89(4):490 – 504, April 2001.

[12] J-M. Daveau, T. B. Ismail, and A. A. Jerraya. Synthesis of System-Level Communication by an Allocation-Based Approach. In *Proceedings of the 8th International Symposium on System Synthesis (ISSS)*, pages 150 – 155, September 1995.

[13] A. Jantsch and H. Tenhunen. *Networks on Chip*. Kluwer Academic Publishers, 2003.

[14] P. V. Knudsen and J. Madsen. Integrating Communication Protocol Selection with Hardware/Software Codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1077 – 1095, 1999.

[15] S. Kumar, A. Jantsch, J-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, and A. Hemani. A Network-on-Chip Architecture and Design Methodology. In *IEEE Computer Society Annual Symposium on VLSI*, pages 117 – 124, April 2002.

[16] J. Madsen, K. Virk, and M. Gonzalez. Abstract RTOS Modelling for Multiprocessor System-on-Chip. In *International Symposium on System-on-Chip*, November 2003.

[17] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens. Communication Scheduling. In *International Conference on Architectural for Programming Languages and Operating Systems*, 2000.

[18] G. D. Micheli, R. Ernst, and W. Wolf. *Readings in Hardware/Software Co-Design*. Morgan-Kaufmann, 2001. 1st edition.

[19] V. J. Mooney and D. M. Blough. A Hardware-Software Real-Time Operating System Framework for SoC's. *IEEE Design & Test of Computers*, 19(6):44 – 51, Nov/Dec 2002.

[20] J. Sifakis. Modelling Real-Time Systems - Challenges and Work Directions. In *EMSOFT, Lecture Notes in Computer Science 2211*, October 2001.

[21] J. Sun and J. Liu. Synchronization Protocols in Distributed Real-Time Systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 38 – 45, May 1996.

[22] SystemC Workgroup. http://www.systemc.org.

[23] T. T. Ye, L. Benini, and G. D. Micheli. Packetized On-Chip Interconnect Communication Analysis for MPSoC. In *Design Automation and Test in Europe, DATE*, pages 344 – 349, March 2003.

[24] X. Zhu and S. Malik. A Hierarchical Modeling Framework for On-Chip Communication Architectures. In *International Conference on Computer-Aided Design (ICCAD)*, pages 663 – 670, 2002.

Chapter 7

A Network Traffic Generator Model for Fast Network-on-Chip Simulation

Published in the Proceedings of Design, Automation and Testing in Europe Conference 2005.

Complete citation:

Shankar Mahadevan, Federico Angiolini, Michael Storgaard, Rasmus G. Olsen, Jens Sparsø and Jan Madsen. "A Network Traffic Generator Model for Fast Network-on-Chip Simulation." *In Proceedings of Design, Automation and Testing in Europe Conference (DATE), Munich Germany.* IEEE, Mar. 2005: 780-785.

A Network Traffic Generator Model for Fast Network-on-Chip Simulation

Shankar Mahadevan[†] Federico Angiolini[‡] Michael Storgaard[†] Rasmus Grøndahl Olsen[†]
Jens Sparsø[†] Jan Madsen[†]

[†] Informatics and Mathematical Modelling (IMM)
Technical University of Denmark (DTU)
Richard Petersens Plads, 2800 Lyngby, Denmark
e-mail: {sm, -,-, jsp, jan}@imm.dtu.dk

[‡] Dipartimento di Elettronica, Informatica e Sistemistica (DEIS)
University of Bologna
Viale Risorgimento, 2 40136 Bologna, Italy
e-mail: {fangiolini}@deis.unibo.it

Abstract

For Systems-on-Chip (SoCs) development, a predominant part of the design time is the simulation time. Performance evaluation and design space exploration of such systems in bit- and cycle-true fashion is becoming prohibitive. We propose a traffic generation (TG) model that provides a fast and effective Network-on-Chip (NoC) development and debugging environment. By capturing the type and the timestamp of communication events at the boundary of an IP core in a reference environment, the TG can subsequently emulate the core's communication behavior in different environments. Access patterns and resource contention in a system are dependent on the interconnect architecture, and our TG is designed to capture the resulting reactiveness. The regenerated traffic, which represents a realistic workload, can thus be used to undertake faster architectural exploration of interconnection alternatives, effectively decoupling simulation of IP cores and of interconnect fabrics. The results with the TG on an AMBA interconnect show a simulation time speedup above a factor of 2 over a complete system simulation, with close to 100% accuracy.

1 Introduction

An important step in the design of a complex System-on-Chip is to select the optimal architecture for the on-chip network (NoC). In order to do so, it is imperative to analyze and understand network traffic patterns through simulation. This can be accomplished at various stages in the design flow, from abstract transaction level models (TLM) to bit- and cycle-true models. In many cases, only the most detailed models prove capable of capturing important aspects of communication performance, e.g. the latency associated with resource contention. The obvious drawback of these approaches is slower simulation speed.

In this paper, we focus on enabling the exploration of different NoC architectures at the bit- and cycle-true level by increasing the speed of the complete SoC simulation. This is a key advantage, since architectural exploration typically involves carrying out the same set of simulations for each design alternative, and simulations may consist of millions of clock cycles each.

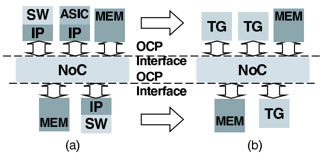


Figure 1. Simulation Environment with bit- and cycle-true: (a) IP-cores, (b) TG model.

We assume as a requirement the availability of a reference SoC design, consisting of IP cores and of a NoC, and of an application partitioned and compiled onto the various IP cores. This application might either be software executing on programmable IP cores, or code synthesized into dedicated hardware. This reference system will be used to collect an initial trace of the IP cores' behavior. In order to increase simulation speed for subsequent design space exploration, we propose to replace the IP cores with Traffic Generators (TG) which emulate their communication at the interface with the network, as illustrated in Figure 1.

The goal is to perform only one reference simulation using bit- and cycle-true simulation models of the IP cores running the target application, and to speed up subsequent variants of that simulation using traffic generators coupled with accurate models of the alternative interconnects only. While the internal processing of IP cores does not need thorough replication by the generators, and can often be modeled by waiting for an amount of cycles between network transactions, the unpredictability of network latency of different NoC architectures may lead to changes in the number and relative ordering of transactions. Thus, traffic generators should have at least some reactive capabilities, as will be explained in Section 3.

In order to capture reactive behavior, we propose a TG implementation as a very simple instruction set processor. Our approach is significantly different from a purely behavioral encapsulation of application code into a simulation device, in analogy with TLM modeling. The TG model we propose is aimed at faithfully replicating traffic patterns generated by a *processor running an application*, not just by the application; this includes e.g. accurate modeling of cache refills and of latencies between accesses, allowing for cycle-true simulations. At the same time, this approach allows a straightforward path towards deployment of the TG device on a silicon NoC test chip.

To evaluate the TG concept, we have integrated the proposed TG model into MPARM [8], a homogeneous multiprocessor SoC simulation platform, which provides a bit- and cycle-true SoC simulation environment. The current version of MPARM supports several NoC architectures, e.g. AMBA [8], STBus and the ×pipes [3], and leverages ARMv7 processors as IP cores. The use of the OCP [1] protocol at the interfaces between the cores and the interconnect allows for easy exchange of IP cores for TGs, as indicated in Figure 1.

The rest of the paper is organized as follows. Section 2 introduces related work, and is followed by a discussion of the requirements for modelling traffic patterns in Section 3. Section 4 details the TG implementation, and Section 5 describes how communication traces are extracted and turned into programs executing on the TG. Section 6 presents initial simulation results which show the potential of our TG approach. Finally, Section 7 provides conclusions.

2 Related Work

The use of traffic generators to speed up simulation is not new, and several traffic generator approaches and models have been proposed.

In [6], a stochastic model is used for NoC exploration. Traffic behavior is statistically represented by means of uniform, Gaussian, or Poisson distributions. Such distributions assume a degree of correlation within the communication transactions which is unlikely in a SoC environment. Traffic patterns in SoC systems have shown to be reactive and bursty [2, 7]. The simplicity and simulation speed of stochastic models may make them valuable during preliminary stages of NoC development, but, since the characteristics (functionality and timing) of the IP core are not captured, such models are unreliable for optimizing NoC features.

A modeling technique which adds functional accuracy and causality is transaction-level modeling (TLM), which has been widely used for NoC and SoC design [4, 5, 9, 10, 11]. In [9, 10], TLM has been used for bus architecture exploration. The communication is modeled as read and write transactions which are implemented within the bus model. Depending on the required accuracy of the simulation results, timing information such as bus arbitration delay is annotated within the bus model. In [10] an additional layer called "cycle count accurate at transaction boundary" is presented. Here, the transactions are issued at the same cycle as that observed in bus-cycle-accurate models, thus intra-transaction visibility is traded-off for simulation speedup. While modeling the entire system at higher abstraction i.e. TLM, both [9] and [10] present a methodology for preserving accuracy with gain in simulation speed.

We would like to underline that our approach is dual with respect to TLM. While transaction-level models usually represent interconnects as a collection of available services and emphasize local processing on IP cores, the platform we describe is composed of accurate models for the interconnect, while processing resources are abstracted away. Simulation speed is gained like in TLM models, but the purpose of this gain is enabling accurate assessment of interconnect performance, not of core or application performance. The above methods are suitable for feature exploration once the NoC architecture has been chosen, but are not thought for NoC exploration itself.

3 Traffic Modeling Requirements

The generation of a traffic pattern emulating that of a real IP core can be faced at varying degrees of accuracy.

At the most basic level, a trace with timestamps can be collected in the reference system and then be independently replayed, an approach that we might call "cloning". This approach is clearly inadequate when the variance of network latency is taken into account; whenever a transaction is delayed, either due to hardware design or congestion, the effect should propagate to subsequent transactions, which would also be delayed in real systems. A simple example of such critical blocking is a cache refill request.

This observation leads to the deployment of "timeshifting" traffic generators: adjacent transactions are tied to each other, and are issued at times which are a function of the delay elapsed before receiving responses to previous transactions. This implicitly means that the trace collection mechanism must include not only timestamps for processor-generated commands, but also for network responses. However, even this model fails when multi-core systems come under scrutiny: the arbitration for resources in such designs is timing-, and thus architecture-, dependent. Therefore, very different transaction patterns may be observed as a function of the chosen interconnection design. To make an example, checks for a shared resource done by polling generate different amounts of traffic depending on the relative ordering of accesses to the resource.

As a consequence, the need for "reactive" TG models is justified. Such models must have some knowledge about the system architecture and about the application behavior to correctly generate (and not just duplicate) traffic patterns

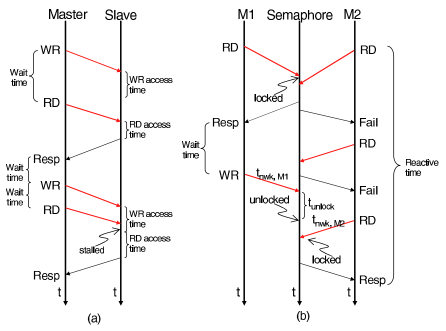


Figure 2. Two typical MPARM transactions.

across different underlying networks. A TG should be able to mimic the behavior of an IP core even when facing unpredictable network performance, e.g. due to resource contention, packet collisions, arbitration and routing policies.

To illustrate the requirements driving the development of our TG model, we will now describe how two typical transactions occurring in the MPARM modelling and simulation environment can be reproduced. MPARM features in-order, single-pipeline ARM cores as system masters and two types of memory as system slaves: private (only accessible by one master) or shared (visible by all masters in the system). Figure 2 shows examples of the two types of communication: (a) processor-initiated communication towards an exclusively owned slave peripheral, and (b) processor-initiated communication towards a system-shared slave peripheral. We call network latency (t_{nwk}) the time taken for the communication to traverse from the master OCP interface to the slave OCP interface and vice versa; this latency depends both on the chosen architecture and on the network congestion at the time of communication.

In Figure 2(a), the first two master transactions are a write (WR) and a read (RD). The time to service the WR transaction, which is a posted write, is just the network latency plus the slave access time. The RD, which in MPARM uses blocking semantics, pays an additional penalty because the response has to make its way back to the master. From the TG point of view, this pattern is easily recordable: network latency and slave access time are unimportant factors, and the essential point to capture is just the delay between WR assertion and RD assertion, and between RD response and the following command. In a subsequent simulation with traffic generators replacing cores, these delays will be modeled by explicit idle waits in the TG, while the network latency will be dependent on the NoC model to simulate. In the next set of transactions, where a RD closely follows a WR, the RD command reaches the slave before the latter has finished servicing the WR, and is thus stalled at the slave interface. This stalling behavior does not need to be explicitly captured in a TG model, since, from a processor perspective, it simply appears to be part of the slave response time. This simplistic example of a master accessing a private slave proves that if the type and the timestamp of the communication events are captured, the behaviour of the master can be emulated via non-preemptive sequential communication transactions interleaved with an appropriate amount of idle wait cycles.

In Figure 2(b), two master devices (M1 and M2) attempt to gain access to a single hardware semaphore. M1 arrives first and locks the resource; the attempt by M2 thus fails. In MPARM, semaphore checking is performed by polling, i.e. M2 regularly issues read events until eventually the semaphore is granted to it. Since the transactions occur over a shared network fabric, the unlock event (WR) issued by M1 and the success of the next request (RD) event by M2 are dependent. Only if the M2 RD event is issued at least $t_{\mathrm{nwk,M1}} + t_{\mathrm{unlock,S}} - t_{\mathrm{nwk,M2}}$ after the unlocking by M1, then M2 will be granted the semaphore and additional polling events will not be required. Therefore, depending on network properties, a variable amount of transactions might be observed at the OCP interfaces of M1 and M2. This is the *reactive behavior* that needs to be captured by the TG model: both M1 and M2 need to react to accommodate the network latency. Thus, the simplistic model which could be applied to transactions towards a privately owned slave now needs to be extended with additional information about the master process execution, about system properties, and about input/output data. In detail, the TG must be able to recognize polling accesses (i.e. a knowledge of what addressing ranges represent pollable resources) and must add support for recording of actual data transfers (e.g., writing a “1” or a “0” to a shared memory location might be the difference between locking or unlocking a resource).

We take the above discussion as a requirement to implement accurate TG models. The examples in Figure 2 demonstrate that traces collected at the IP-NoC interface are sufficient to accurately reproduce the IP’s communication, provided that the reactive behavior of the master IP cores is taken into account. These traces should collect sequences of communication transactions, comprising of requests and responses, separated by time intervals with no communication, i.e. idle time. A simulation of the entire system should produce several traces, one per IP core interface.

4 Implementation of the traffic generators

In this section we describe in some detail the implementation of our traffic generators. As mentioned before, our proposed TG model is designed as a simulation tool, but allows future deployment as a hardware device. Within the simulation environment for NoC exploration, the emphasis is on simulation speedup, while within a hardware instance the emphasis is on ease of (re)programmability and a small silicon footprint in order to support implementation of test chips containing NOC prototypes.

Conceptually, three different TG entities might be needed: (1) A TG emulating a processor (an OCP master). This TG must be able to issue conditional sequences of traces composed of communication transactions separated by idle/wait-periods; (2) A TG emulating a shared memory (an OCP slave). This TG must contain a data structure modeling an actual shared memory (since the values read by the masters may affect the sequence of transactions seen at the master IP cores); and (3) A TG emulating a slave memory (an OCP slave). This TG must be able to respond, possibly with dummy values, to communication transactions issued by a master. Only the first is actually required for deployment in a simulation environment, which already provides its own system slaves, thus only this entity will be described in the present paper. On the other hand, it is important to notice that both slave TG modules are much simpler in design with respect to the master TG, as their logic basically just involves a small state machine to handle OCP transactions.

For the processor TG, we have implemented and modeled a multi-cycle processor with a very simple instruction set as listed in Table 1. The processor has an instruction memory and a register file, but no data memory. The instruction set consists of a group of instructions which issue OCP transactions (whose arguments are set up in registers) and a group of instructions allowing the programming of conditional sequencing and parameterized waits such that the required traces can be implemented/programmed. The process for deriving TG programs from traces obtained in the reference simulation is explained in Section 5.

5 The TG simulation flow

In order to use the traffic generators, a user must first perform a reference simulation using bit-true and cycle-true IP models. It is interesting to note that, at this stage, the interconnect does not yet need to be accurately modeled, allowing for time savings. During this simulation, traces are collected from all OCP interfaces in the system. For this purpose, the OCP interface modules within the MPARM platform (the network interfaces in the case of the ×pipes interconnect, the bus master in the case of AMBA AHB) were adapted to collect traces of OCP request and response communication events into a predefined file format (*.trc*). The address and (if any) data fields of the transactions are also observed. Trace entries are single or burst read/write transactions. Figure 3(a) shows an example trace.

The next step is to convert the traces into corresponding TG programs (*.tgp*). A translator outputs symbolic code; Figure 3(b) shows the TG program derived for traces in Figure 3(a). Finally, an assembler is used to convert the symbolic TG program into a binary image (*.bin*) which can be

Instructions	Description
OCP Instructions:	
Read(addr)	Read from an address
Write(addr, data)	Write to an address
BurstRead(addr, count)	Burst read a range of addr.
BurstWrite(addr, data, count)	Burst write an address set
Other Instructions:	
If(arg1, arg2, operand)	Branch on condition
Jump(location)	Branch direct
SetRegister(reg, value)	Set register (load immediate)
Idle(counter)	Wait for given no of cycles

Table 1. OCP-master TG instruction set.

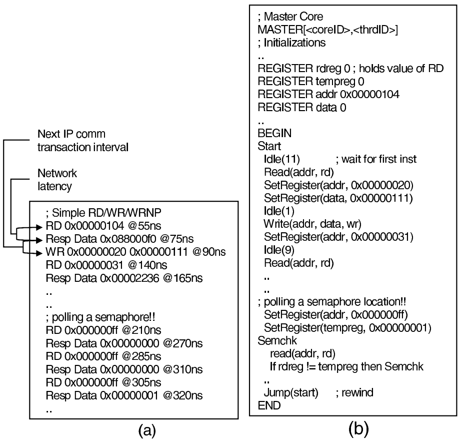
loaded into the TG instruction memory and executed. Execution might be within a simulation model (which is the approach presented in this paper) or in hardware on a NoC test-chip. Validation of the trace collection and processing mechanism can be achieved by collecting traces with IP cores running on different interconnects, and verifying the resulting *.tgp* and *.bin* programs to match. The conversion process is fully automated and the time taken for this process is discussed in Section 6.

As seen in Figure 3(b), the TG program starts with a header describing the type of core and its identifier. The next few statements express initialization of the register file. Register `rdreg` is defined as special register where the value of RD transactions is stored.

By looking at the code in Figure 3(a), it is possible to notice that the first communication events in the trace occur at time 55ns, 75ns, and 90ns. We assume each TG cycle to take 5ns, the same as the IP core for which the trace is collected. At the beginning of the simulation, the TG has no instruction to perform until the 11th (55/5) cycle, so an `Idle` wait is observed. The trace of the RD event is followed by a response, at a time which is dependent on the network latency. The IP core is blocked until this response arrives. A WR event occurs three ((90-75)/5) cycles after the response is received; these cycles are partially spent for TG internal operations (data and address register setting), and an ensuing `Idle` wait is added to fill the gap. Then the following RD instruction is translated into the corresponding `Read` program call after 10 cycles, one of which is taken to set up the RD address. This is blocking until a response is received, 5 cycles later.

Now, consider the trace entries from time 210ns to 320ns. By identifying the address as belonging to a semaphore location and knowing the polling behaviour of the MPARM IP core, the translator inserts the `Semchk` label and an `If` conditional statement. This statement checks whether the read value is equal to "1", which reflects an unblocked semaphore. This loop effectively represents the semaphore polling behavior. All master devices attempting to access this address incorporate the same routine in their TG program, thus capturing the system dynamics.

At this stage, additional simulations can be run on a plat-



```
; Simple RD/WR/WRNP
RD 0x00000104 @55ns
Resp Data 0x088000f0 @75ns
WR 0x00000020 0x00000111 @90ns
RD 0x00000031 @140ns
Resp Data 0x00002236 @165ns
..
..
; polling a semaphore!!
RD 0x000000ff @210ns
Resp Data 0x00000000 @270ns
RD 0x000000ff @285ns
Resp Data 0x00000000 @310ns
RD 0x000000ff @305ns
Resp Data 0x00000001 @320ns
..
```

```
; Master Core
MASTER[<coreID>,<thrdID>]
; Initializations
..
REGISTER rdreg 0 ; holds value of RD
REGISTER tempreg 0
REGISTER addr 0x00000104
REGISTER data 0
..
BEGIN
Start
  Idle(11)        ; wait for first inst
  Read(addr, rd)
  SetRegister(addr, 0x00000020)
  SetRegister(data, 0x00000111)
  Idle(1)
  Write(addr, data, wr)
  SetRegister(addr, 0x00000031)
  Idle(9)
  Read(addr, rd)
  ..
  ..
; polling a semaphore location!!
  SetRegister(addr, 0x000000ff)
  SetRegister(tempreg, 0x00000001)
Semchk
   read(addr, rd)
   If rdreg != tempreg then Semchk
  ..
  Jump(start)     ; rewind
END
```

Figure 3. (a) MPARM Trace to (b) TG Program.

form with traffic generators and a variety of interconnect fabrics, thereby evaluating performance of NoC design alternatives. Compared with the reference setup, where the interconnect fabric could be modeled at a high level, the target NoC should now be simulated at the cycle- and bit-true level to carefully assess its performance. Validation of the TG model can be achieved by coupling the TG with the same interconnect used for tracing with IP cores, and checking the accuracy of the IP core emulation. Results for this validation, and for tests on different interconnects than the reference one, will be presented in the next Section.

6 Results

We simulated within the MPARM framework, using the AMBA NoC, and four benchmarks. The first benchmark was a single-processor application (SP matrix manipulation), with the purpose of assessing accuracy and speedup in the simplest environment. The second benchmark (Cacheloop) was a test performing idle loops within the processors' cache, and only minimal bus interaction; this allowed an assessment of the speedup provided by the TG model when scaling the number of processors in the system up to twelve. Finally, the remaining two benchmarks (MP matrix manipulation and DES encryption/decryption) were multiprocessor tests stressing synchronization and resource contention with traffic patterns as discussed in Section 3, and were used mainly to ascertain the accuracy of the whole design flow when stressed by complex transactions.

In the first experiment we aimed at validating the trace collection/processing environment. We ran the same benchmarks over AMBA and ×pipes, noticing very different execution times due to different latency and scalability features. However, after translation, a check across *.tgp* programs showed no difference at all. This result demonstrates the feasibility of an approach which decouples simulation of the IP cores and of the underlying interconnect fabric.

#IPs	Cumulative Execution Time			Simulation Time		
	ARM	TG	Error	ARM	TG	Gain
SP matrix:						
1P	6610680	6610659	0.00%	73 s	34 s	2.15x
Cacheloop:						
2P	2500903	2500913	0.00%	47 s	14 s	3.36x
4P	2501760	2501701	0.00%	87 s	22 s	3.95x
6P	2502558	2502640	0.00%	127 s	29 s	4.38x
8P	2503404	2503522	0.00%	163 s	37 s	4.41x
10P	2504250	2504404	0.01%	197 s	42 s	4.69x
12P	2505096	2505286	0.01%	239 s	51 s	4.69x
MP matrix:						
2P	3276505	3276030	0.01%	66 s	25 s	2.64x
4P	3528038	3530759	0.08%	128 s	42 s	3.05x
6P	3691454	3697854	0.17%	195 s	61 s	3.20x
8P	3997878	4058812	1.52%	260 s	82 s	3.17x
10P	4881007	4902806	0.45%	334 s	106 s	3.15x
12P	5901290	5901131	0.00%	432 s	143 s	3.02x
DES:						
3P	978080	980098	0.21%	26 s	10 s	2.60x
4P	1054839	1057944	0.29%	34 s	11 s	3.09x
6P	1491570	1492274	0.05%	53 s	20 s	2.65x
8P	1959755	1960575	0.04%	73 s	30 s	2.43x
10P	2441026	2441743	0.03%	95 s	42 s	2.26x
12P	2927359	2927218	0.00%	125 s	62 s	2.02x

Table 2. TG vs. ARM performance with AMBA.

Table 2 summarizes the results of simulations done on the AMBA AHB interconnect with ARM processors and then with TGs. The left columns report the number of simulated cycles, while the right ones illustrate simulation time[1]. The column "Error" is a measure of the accuracy of replacing IP cores with TGs, based upon the difference in simulated cycles, while the column "Gain" describes the improvement in simulation time.

The table shows that replacing ARM processors with TGs yields excellent accuracy, close to 100% for small numbers of processors, while guaranteeing a speedup factor of 2 to 4. This speedup is mostly due to the drastic simplification in the amount of logic needed to generate communication transactions, compounded in small part with the elimination of any adaptation layer in the system since the TG is natively implemented with an OCP interface. This speedup compares favorably to previous work in the area (a speedup of 1.55x is reported in [10]), and must be evaluated by taking into account the fact that it involves no shift in the level of abstraction of the simulations.

[1]Benchmarks taken on a multiprocessor Xeon® 1.5 GHz with 12 GB of RAM, eliminating any disk swapping effect. Especially for benchmarks with a short duration, time measurements were taken by averaging over multiple runs and care was put in minimizing disk loading effects.

Inaccuracies in execution time can be explained as follows. In Cacheloop, and until about 6-8 processors in MP matrix and four in DES, the TG platform shows an acceptable accuracy degradation. This is due to the compounding of minimal timing mismatches caused by the conversion from traces to TG programs. When adding even more processors, however, accuracy improves again, because the AMBA bus starts to saturate, causing the processors to idle wait for bus arbitration for long amounts of time. The congestion is serious enough to dominate the effect of the timing mismatches. Since TGs cannot save simulation complexity if the replaced processors are in idle state, this is also the reason causing the speedup to get smaller with large numbers of processors in MP matrix and DES. For Cacheloop, which always executes from the local caches without any bus traffic, this phenomenon does not appear. Thus, the reduced speedup is not a property of the TG.

The impact of trace collection is small, and is incurred only once. For example, when running the MP matrix benchmark on the AMBA interconnect with four ARM processors, a plain benchmark run takes 128 s; the benchmark run with TG tracing enabled takes 147 s, and subsequent parsing and elaboration requires an additional 145 s for a 20 MB trace file[1]. Only one such iteration is needed to be able to take advantage of 2x to 4x speedups in subsequent design space exploration. Additionally, since processed TG programs are identical regardless of the reference interconnect in which raw traces were collected, such collection could be performed on top of a transactional fabric model, further reducing the impact of the reference simulation.

7 Conclusions

Experimental results prove the viability of a TG-based approach which decouples simulation of IP cores and of interconnect fabrics. Even in presence of unpredictable contention for shared resources in a multiprocessor environment, our TG model proved capable of delivering speedups in the order of 2x to 4x when run on AMBA while keeping a remarkable accuracy.

The TG model we propose provides a wide range of features, with a simple but powerful instruction set allowing for sophisticated flow control and therefore a variety of communication patterns. It is very useful for fast and accurate verification and exploration of different NoC architectures, which is the motivation of this work. While this paper was focused on simulation speedup, the TG may also be used as a flexible tool in a variety of platforms. The TG might be used in association with manually written programs to generate traffic patterns typical of IP cores still in the design phase, helping in the tuning of the communication performance between the underlying NoC and that IP core.

Our future work includes synthesis of the TG device, and support for processors allowing out-of-order transactions. Research will also include analysis of the behavior of a system in which multiple tasks run on a single processor and are dynamically scheduled by an OS, either based upon timeslices (preemptive multitasking) or upon transition to a sleep state followed by awakening on interrupt receipt. Context switching-related issues will need to be modeled or predicted.

8 Acknowledgments

The work of Shankar Mahadevan is partially funded by SoC-Mobinet, Nokia, and the Thomas B.Thrige Foundation. The work of Federico Angiolini is partially funded by ARTIST.

References

[1] Open Core Protocol Specification, Release 2.0 http://www.ocpip.org, 2003.

[2] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. In *Journal of Systems Architecture*. Elsevier, 2004.

[3] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. xpipes: A latency insensitive parameterized Network-on-Chip architecture for multi-processor SoCs. In *Proceedings of 21st International Conference on Computer Design*, pages 536–539. IEEE Computer Society, 2003.

[4] F. Fummi, P. Gallo, S. Martini, G. Perbellini, M. Poncino, and F. Ricciato. A timing-accurate modeling and simulation environment for networked embedded systems. In *Proceedings of the 42th Design Automation Conference*, pages 42–47, 2003.

[5] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[6] K. Lahiri, A. Raghunathan, and S. Dey. Evaluation of the traffic-performance characteristics of System-on-Chip communication architectures. In *Proceedings of the 14th International Conference on VLSI Design*, pages 29–35, 2001.

[7] K. Lahiri, A. Raghunathan, G. Lakshminarayana, and S. Dey. Communication architecture tuners: A methodology for the design of high-performance communication architectures for System-on-Chips. In *Proceedings of the 2000 Design Automation Conference, DAC'00*, pages 513–518, 2000.

[8] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*. IEEE, 2004.

[9] O. Ogawa, S. B. de Noyer, P. Chauvet, K. Shinohara, Y. Watanabe, H. Niizuma, T. Sasaki, and Y. Takai. A practical approach for bus architecture optimization at transaction level. In *Proceedings of Design, Automation and Testing in Europe Conference 2004 (DATE03)*. IEEE, March 2003.

[10] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proceedings of 38th Design Automation Conference (DAC'04)*, pages 113–118. ACM, 2004.

[11] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the System-on-Chip interconnect woes through communication-based design. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 667 – 672, June 2001.

Chapter 8

Realistically Rendering SoC Traffic Patterns with Interrupt Awareness

Published in the IFIP Very Large Scale Integration Systems and their Designs Conference 2005.

Complete citation:

Federico Angiolini, Shankar Mahadevan, Jan Madsen, Luca Benini and Jens Sparsø. “Realistically Rendering SoC Traffic Patterns with Interrupt Awareness.” *IFIP Very Large Scale Integration Systems and their Designs Conference (VLSI-SoC), Perth Australia.* IEEE, Oct. 2005: 211-216.

Realistically Rendering SoC Traffic Patterns with Interrupt Awareness

Federico Angiolini[‡] Shankar Mahadevan[†] Jan Madsen[†] Luca Benini[‡] Jens Sparsø[†]

[†] Informatics and Mathematical Modelling (IMM)
Technical University of Denmark (DTU)
Richard Petersens Plads, 2800 Lyngby, Denmark
e-mail: {sm, jan, jsp}@imm.dtu.dk

[‡] Dipartimento di Elettronica, Informatica e Sistemistica (DEIS)
University of Bologna
Viale Risorgimento, 2 40136 Bologna, Italy
e-mail: {fangiolini, lbenini}@deis.unibo.it

Abstract

In Multi-Processor System-on-Chip (MPSoC) design stages, accurate modeling of IP behaviour is crucial to analyze interconnect effectiveness. However, parallel development of components may cause IP core models to be still unavailable when tuning communication performance. Traditionally, synthetic traffic generators have been used to overcome such an issue. However, target applications increasingly present non-trivial execution flows and synchronization patterns, especially in presence of underlying operating systems and when exploiting interrupt facilities. This property makes it very difficult to generate realistic test traffic. This paper presents a selection of applications using interrupt-based synchronization; a reference methodology to split such applications in execution subflows and to adjust the overall execution stream based upon hardware events; a reactive simulation device capable of correctly replicating such software behaviours in the MPSoC design phase. Additionally, we validate the proposed concept by showing cycle-accurate reproduction of a previously traced application flow.

1 Introduction

Undertaking realistic exploration and optimization of the SoC interconnect is an important but time-consuming step in designing a multiprocessor SoC. It requires cycle-true simulation models of both the IP cores and the interconnect to be simultaneously available and ready to interoperate. Shrinking product lifetimes force the designers to consider the deployment of Traffic Generators (TGs), *i.e.* of devices capable of generating inter-core transactions. However, it is possible that neither stochastic traffic patterns [3], nor playback of prerecorded transaction traces collected on a reference system, are sufficient for interconnect optimization. The former approach fails to correctly capture the time distribution of traffic spikes, while in the latter approach, the deployment of different cores and interconnect architectures with respect to the reference platform leads to unpredictable system behaviour. This includes variance in the number and relative ordering of transactions, thus rendering the use of prerecorded traces inaccurate. An example is synchronization by semaphore polling, which can require an unknown number of bus accesses before getting lock ownership.

An even greater challenge is posed by inherently asynchronous communication events such as interrupts. While interrupts themselves typically have a low impact on communication resources, interrupt handling can severely impact network traffic with activity peaks. Interrupt notification is generally supposed to alter the application flow, *e.g.* by notifying the need for a change of operating mode, such as rescheduling and context switching in the Operating System (OS), which may result in completely different communication requirements. Thus, a model describing IP core traffic should feature extensive reactive capabilities to mimic the behaviour of the core when facing unpredictable environmental events and network performance. Additionally, awareness of the multiprocessor nature of the target platform, which implies synchronization requirements, should be provided.

In this paper, we present a *reactive traffic generator model*, encompassing an instruction set and a programmable simulation device. The novel feature of our approach is that any knowledge about the behaviour of the final system can be thoroughly taken into account and rendered by means of *TG programs*. Based on the flow control written in the TG program, described later in Section 3, the TG model realistically adjusts its output depending on complex external synchronization events, like semaphore interaction and interrupt notification. The result are traffic patterns that closely resemble those of the real application with OS, running on top of the real IP core, while not giving up any accuracy on the handling of multiprocessor synchronization and intercommunication issues (*i.e.* idle waits, task interruption, and so on). The proposed cycle-true TG approach allows for the separation of computation and communication concerns, so that designers can focus on accurate exploration of the SoC interconnect.

Previously, in [5], we had proposed a traffic generator model capable of capturing the dynamics of core-initiated communication (reads, writes). However, it could only handle simple test cases, *i.e.* single task per processor, with the OS transparent to the application flow. The contribution of the current paper is an extension of the model aimed at accurately capturing system-initiated communication, such as interrupts, and the related response, *i.e.* the OS-driven interrupt handling mechanism. As a demonstration of the flexibility and accuracy of the proposed model, this paper will show how it can be applied to more complex and true-

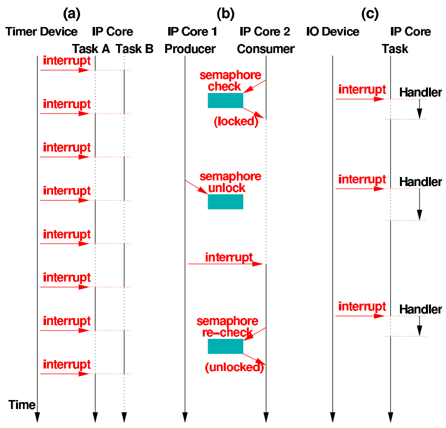


Figure 1. Interrupt-aware Applications. Dotted lines represent suspended tasks.

to-life applications, with general-purpose ARM processors running an OS in a multicore environment.

The rest of the paper is organized as follows. Relevant interrupt-aware applications to be modeled are discussed in Section 2. Section 3 presents details of the proposed implementation of the TG, specifically stressing flow control handling in presence of interrupts. Section 4 describes possible ways to write programs for execution on top of TGs, and Section 5 highlights an example TG deployment. Section 6 presents simulation results which document the potential of our TG approach. Section 7 contrasts our results to previous work. Finally, Section 8 provides conclusions.

2 Interrupt-based Synchronization Scenarios

Using interrupts, many communication and synchronization schemes are possible among tasks in a multiprocessor environment. To analyze such a wide variety of patterns [8], we identified three applications, interacting both among themselves and with the underlying OS, which highlight interrupt handling scenarios typical of real systems. These applications perform relatively light computation but exhibit non-trivial flow patterns, which makes them much more difficult to model than computation-intensive tasks. As such, these test cases are used to derive requirements of the most typical interrupt-based flow controls. The application templates we identified are:

- A multi-tasking application (**“task”**), as in Figure 1(a). In this case, two tasks run on each processor; a variable amount of system processors may be present. No explicit communication is performed between tasks, neither intra- nor inter-core. The context switching between tasks is performed by the OS in response to an external interrupt, which may typically be sent by a timer device. It is important to notice that, if tasks are asymmetric, any rescheduling translates into different traffic workloads for the communication fabric. This effect must be captured.

- A task synchronization application (**“pipe”**), as in Figure 1(b). For this case, a single task is mapped onto every system core. Tasks are programmed to communicate with each other in a point-to-point producer-consumer fashion; every task acts both as a consumer (for an upstream task) and as a producer (for a downstream task), therefore logical pipelines can be achieved by instantiating multiple cores. Synchronization is needed in every task to check the availability of input data and of output space before attempting data transfers. To guarantee data integrity, semaphores are provided to assess such availability. For example, the consumer checks a semaphore before accessing producer output. If this semaphore is found initially locked, a continuous polling might be attempted, but at the expense of wasted energy and saturation of the system interconnect. Instead, we implemented an interrupt mechanism which, in such a scenario, suspends the consumer task and resumes it only when data is ready.

- An IO-aware application (**“IO”**), as in Figure 1(c). A single task is running on every system processor. These tasks do not communicate with each other, and perform independent computation. However, at random times, a system I/O device sends an interrupt to all of the cores to signal data availability. In response to this signal, the processors execute an interrupt handler routine, which moves data blocks across the system interconnect. When such handling is completed, tasks resume their normal operation.

In real life, a **task**-style application flow can be observed in time-slicing mechanisms in OS schedulers, while **pipe** models the processing of multimedia datastreams. The **IO** flow is commonly found in applications interacting with input/output devices. It is clear from the presented application behaviours that accurately capturing the interrupt propagation (and therefore the synchronization schemes) requires thorough analysis.

The applications described above are timing-sensitive. However, within the single task, the overall performed computation does not change depending on the order of arrival of external events, and data dependencies can be captured. Only the amount of computation between each pair of events can vary. Should an environment constraint not be satisfied, tasks always enter some form of suspension, albeit in very different manners in each of the three examples. So, while an execution trace of these benchmarks shows varying traffic patterns depending on external timings, the major computation blocks are still recognizable.

Even though tasks with even more timing-dependent behaviour do exist, modeling such tasks requires an intra-task notion of context switching, which we omit here. It is worth

Instruction	Description
OCP Instructions:	
Read(AddrReg)	Read from an address
Write(AddrReg, DataReg)	Write to an address
BurstRead(AddrReg, CountReg)	Burst read an address set
BurstWrite(AddrReg, DataReg, CountReg)	Burst write an address set
Other Instructions:	
If(arg1, arg2, operand)	Branch on condition
Jump(label)	Branch direct
SetRegister(reg, value)	Set register (load immediate)
Idle(counter)	Wait for given no of cycles

Table 1. OCP-master TG instruction set.

Special Registers	Name	Usage
Interrupt Registers:		
2	IntrpMaskReg	Masks or unmasks interrupts
3	IntrpReg	Stores a backup of the program counter
5	SWIntrpReg	Sends a software interrupt from within the program
Other Registers:		
1	ThrdIDReg	Stores the ID of the current task
4	RDReg	Stores the data value returned by the Read(AddrReg) instruction
6	RtnReg	Stores a jump target location

Table 2. TG Special Registers.

stressing that, though not all interrupt-driven behaviours are represented, the applications we try to analyze here are definitely representative of a vast class of computation. The model we will propose can capture all such dynamics, given proper insight on the mechanics of the applications and the OS.

3 Support for Application Flow Replication

In this section, we describe (i) an instruction set which is capable of replicating the traffic patterns generated by an IP core, (ii) an implementation of it by means of a TG Instruction Set Simulator (ISS), and (iii) an example program written to exploit TG capabilities. The whole approach significantly extends [5] to support interrupts and task switching, enabling the modeling of real-life tasks.

The TG has an Open Core Protocol (OCP) [2] master interface, and it can emulate IP cores running one or multiple tasks with or without OS. The TG is able to issue a sequence of communication transactions separated by idle wait periods, based on the programmed flow control conditions. In order to handle interrupts and other synchronization events, it is *reactive*, *i.e.*, if necessary, it is able to switch between tasks upon notification. The TG is implemented as a non-pipelined processor with a very simple instruction set, as listed in Table 1. The processor has an instruction memory and a register file for each task, but no data memory. The instruction set consists of a group of instructions which issue OCP transactions and a group of instructions allowing the programming of conditional sequencing and parameterized waits. Within the register file, some registers are designated as special purpose for flow control management; their usage is described in Table 2. The rest are general purpose registers, and their number can be configured.

Of the interrupt-related registers, Register 2 can be used to mask critical sections of the TG program from interrupts. As seen in Section 2, different applications require different responses to interrupt events. For example, in **IO** modeling, the main task is always interruptible, while once in the OS's interrupt handling routine, additional (nested) interrupts should be disabled. Register 3 holds the base location of the interrupt handling code within the TG program. Register 5 allows the TG program to assert "software interrupts", to which the TG model will react with jumps to different parts of the program. Software (SW) interrupts are managed internally by the TG model. In contrast, hardware (HW) interrupts are routed through external wires from the interconnect, and are available on the sideband signals (`SInterrupt`) of the OCP interface. Registers 1, 4 and 6 provide support for specific flow control functions.

Within the TG ISS, by maintaining copies of the Program Counter (`PC`) and register file associated with each task, the context switching upon an interrupt event can be realized. Upon interrupt notification, the values of the `PC` and register file of the interrupted task are saved, the `PC` is updated with a value read from the special Register 3, and the register file values for the designated task are loaded. It is afterwards possible to safely exit from the interrupt routine and resume a suspended task by jumping to the backup value of the source `PC` and reloading the backup of the register file.

Let us now consider an example of a TG program. In Figure 2, a program to model the **IO** application is sketched; the interrupt handling routine is coded together with the task itself. The TG program starts with a header describing the type of core and its identifier. The next few statements express initialization of the register file. The `PC` is increasing by either one or two locations along the trace; this is because some of the opcodes in Table 1, namely `SetRegister` and `If`, require longer operands and therefore fill two program slots. The main body of the TG program is composed of sequences of bus reads and writes, interleaved with register accesses (mostly to set up transaction address and data). Flow control instructions are inserted where appropriate. The interrupt handling routine is located at `PC` 37; this base address is stored in `IntrptReg`, which is initialized at `PC` 2. Within the interrupt routine, which is the critical section of the flow, interrupts are disabled. Upon a HW interrupt event, the TG swaps the content of `IntrptReg` with that of `PC`. The TG program then executes any OS- or programmer-driven interrupt instructions, including transactions over the communication architecture. At the end of the flow, a software interrupt is triggered to restore the `PC` to the previously interrupted location (retrieved from `IntrptReg`). The flow thus mimics Figure 1(c).

4 Coding TG Programs

Depending on IP model availability to the designer, different ways exist to write TG programs which best represent the desired type of traffic.

```
MASTER[<coreID>]                      ; Initializations
  REGISTER IntrpMaskReg 0             ; INTRP Mask
  REGISTER IntrptReg 0                ; INTRP Save PC
..
BEGIN                                 ; Comments              PC
  SetRegister(IntrpMaskReg, 1)        ; Unmask HW INTRP       0
  SetRegister(IntrptReg, 37)          ; IRC at PC 37          2
  idle(10)                            ; idle for 10 cycles    4
  ..
  SetRegister(AddrReg, 2)             ; normal flow           10
  SetRegister(DataReg, 1)             ;                       12
  Write(AddrReg, DataReg)             ;                       14
  ..
  jump(myPRGM)                        ; jump to PC 58         36
```

```
 ; Interrupt Handling Routine at PC 37
IHR SetRegister(IntrpMaskReg, 0)      ; Mask HW INTRP         37
  SetRegister(AddrReg, 23)            ;                       39
  SetRegister(DataReg, 1)             ;                       41
  Write(AddrReg, DataReg)             ;                       43
  ..
  SetRegister(IntrpMaskReg, 1)        ; Unmask HW INTRP       54
  SetRegister(SWIntrpReg, 1)          ; Trigger SW INTRP      56
; End Interrupt Handling
```

```
myPRGM SetRegister(AddrReg, 11)       ; Continue normal flow  58
  Read(AddrReg)                       ;                       60
  ..
END                                   ;                       124
```

Figure 2. IO TG Program.

4.1 Trace Parsing

In this scenario, availability of a pre-existing model for the IP under study is assumed. In this case, the approach for TG program generation goes through two steps. First, a reference simulation is performed by using the available IP model, even plugged into a different SoC platform from the target one. An execution trace is collected. Second, the trace is parsed with an off-line tool. The output of the tool is the desired TG program.

In [5], we applied the above methology and showed that the replacement of a pre-existing ISS with a TG device can speed up subsequent simulations, which is valuable in any design space exploration stage. The TG provides a quick functional yet cycle-accurate port of the IP model to a SoC's interconnect platform; this is useful in the case where pre-existing IP models are not directly or immediately available (due to licensing or technical issues) for the next co-exploration phase.

4.2 Trace Parsing and Editing

In a related scenario, an IP model might be available, but it may differ under some respect from the IP that will eventually be deployed in the SoC device. The designer may then follow a route similar to the one outlined above, but with an additional step: editing the reference trace so that it more closely resembles that of the target IP. Some examples of the editing steps which are possible include:

- Removing or adding bus transactions to approximate a different cache subsystem and/or a target Instruction Set Architecture (ISA) behaviour
- Altering the spacing among bus transactions to reflect different pipeline designs or timing properties
- Grouping or ungrouping bus accesses to reflect write-back *vs.* write-through cache policies

The effort required to automate these kinds of trace alterations is expected to be quite low. It is certainly reasonable to expect that the TG program coding time will be substantially less than that required to develop or refine the target IP model, thus allowing for earlier exploration of the interconnect design space.

In this scenario, overall cycle accuracy with respect to the eventual system is of course not guaranteed. However, the TG will still be able to react with cycle accuracy to any optimization in the SoC interconnect. Provided that the transaction patterns are kept close to the ones of the target IP core, the approach will result in valuable guidelines.

4.3 Direct Development

Of course, TG programs can be written from scratch. In this case, the flexible TG instruction set allows for a full-featured traffic generation system. The availability of built-in flow control management lets the designer implement the same synchronization patterns which are present in real world applications (see Section 3 and [5]). Additionally, the application chunks enclosed within synchronization points can quickly be rendered by exploiting the flexible loop structures provided by the TG ISS, thus providing periodic traffic generation capabilities at least on par with those of traditional TG implementations.

5 Test Case of Trace-Based TG Deployment

To test TG accuracy and viability, we set up a validation flow following the outline described in Section 4.1. The TG model was integrated into MPARM [4], a homogeneous multiprocessor SoC simulation platform, which provides a bit- and cycle-true SoC reference simulation environment. MPARM also contains a port of RTEMS [1] - a real-time OS. The use of the OCP 2.0 [2] protocol at the interfaces between the cores and the interconnect allows for an easy exchange of IP cores for TGs. After performing a reference simulation, where execution traces were collected, we processed them to derive suitable TG programs. The off-line tool for trace to TG program conversion, explained in [5], was significantly expanded to capture fundamental application flow properties and synchronization patterns like those described in Section 2. The trace to TG program conversion process is fully automated and the time taken for this process is nominal. The validation of the TG flow was achieved by coupling the TG with the same interconnect used for tracing with IP cores, and checking the accuracy of the resulting IP core emulation. Experimental results will be shown in Section 6.

It is worth stressing that the complexity of application modeling in presence of interrupt handling is not trivial. However, the algorithm in the automated flow is capable of detecting and capturing many synchronization behaviours of Section 2, without the need for the designer to handle

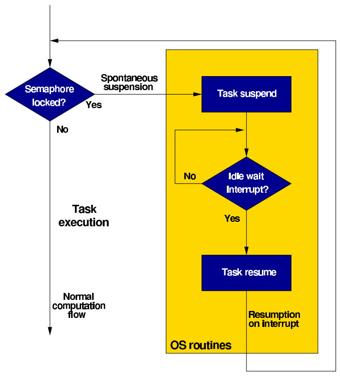


Figure 3. Application flow of pipe.

them manually, as explained next. Depending on the target application, one or more of the following pieces of information can be extracted about interrupt handling from the trace file to help the translator tool:

- the time when interrupt events occur,
- the end of an interrupt handling routine,
- the spontaneous suspension waiting for an interrupt in idle state.

The amount of annotations that can be extracted reflects the degree of access the programmer has to the interrupt routine and to the OS internals. In the **IO** test case, the interrupt handling is likely to be part of the functionality of a custom device driver, and thus we assume that the programmer has full access to both the code of the application and of the interrupt handler. Therefore, trace files contain the time of occurrence of the interrupt event; custom markers (*i.e.* dummy memory accesses to specific locations) can be appended by the programmer at the end of the interrupt handling routine. The transactions within these bounds can be detected as interrupt handling code and be encapsulated as such in the TG program.

In the **pipe** scenario, the task is interacting with the OS internals by voluntarily suspending should certain conditions be true (*i.e.* finding a semaphore locked). Additionally, the task negotiates with the OS to be resumed upon interrupt receipt. The task may also want to ignore an interrupt in the following condition: it is possible that the upstream producer, or the downstream consumer, notifies availability of data or buffer space before the actual need for such resources, because the current task is still busy with previous internal processing. Despite the complex interaction, usually the synchronization functionality required by **pipe** can be achieved by properly using OS APIs, without direct access to the interrupt handler code, whose exit point is therefore assumed to be not accessible by the programmer. As a result, the only annotations of significance within

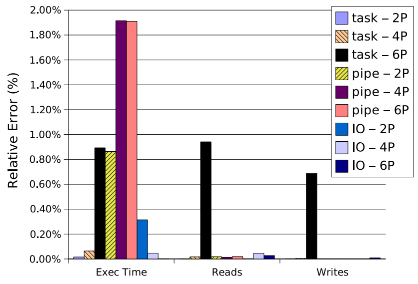


Figure 4. Accuracy of the execution on TGs *vs.* the original ARM cores.

the trace file are the synchronization points (semaphore checks) and the interrupt arrival time. A TG program can thus mimic the flow shown in Figure 1(b), first by reading the semaphore location, and then by choosing to continue or suspend depending on the lock. Upon resumption by HW interrupt, a final (re-)check of the semaphore unlock can be done to ensure safe task operation. Figure 3 shows the equivalent flow. In the TG program, HW interrupts are used to wake up from the suspension state within OS routines, while SW interrupts redirect the execution flow towards the main task. Note that `IntrpMaskReg` is set to the masked state for the regular program and OS execution, and is only unmasked within the suspended state.

In the **task** benchmark, the interrupt handler is typically completely out of the programmer's control, as it is tied to the OS scheduling code. The tasks are not explicitly notified upon the receipt of an interrupt, and are just suspended and resumed by the OS. Therefore, trace files are annotated only with the time of occurrence of interrupt events. The TG execution toggles among tasks upon these interrupts. This is not much different from **IO**, but, since it is assumed for the programmer to be impossible to explicitly tag the handler exit point with a custom flag, the interrupt handling routine is merged with a stage of the next scheduled task because the off-line tool has no way to detect this jump. Additionally, control is never spontaneously released by means of SW interrupts: the previously active task is only resumed upon arrival of a HW interrupt. The TG ISS automatically supports context switching, as described in Section 3, with multiple register sets.

6 Experimental Results

We coded the three test cases mentioned in Section 2 as tasks running on top of an operating system and we simulated them within the MPARM framework. Each was tested with two (2P), four (4P) and six (6P) system processors. For **task** and **IO**, we devoted one of the system cores to the generation of interrupts, emulating the role of a timer or an IO device; this processor is not generating any other traffic on the bus, and is just idling between interrupt generation events. The **pipe** benchmark does not need this, since interrupts are directly triggered by the same tasks which perform the computation. Figure 4 depicts the accuracy of our modeling scheme, by plotting the mismatch among the original execution on ARM cores and the execution on TGs (after applying the flow described in Section 5).

The plot shows a good match between ARM and TG runs. The typical relative error, both in execution time and number of bus accesses, is below 2%, resulting in a faithful reproduction of the original execution flow and traffic patterns. The near-matching amount of read and write accesses proves the role of the TG as a powerful design tool to mimic complex application behaviour in replacement of a real IP core. Additionally, the correctness of our TG program translation is validated. Some mismatches can be observed especially in the execution time for the **pipe** benchmark. These are due to minor issues in properly pinpointing single sections of internal OS code in the execution trace.

With regards to the simulation speedup, a gain of 1.37x to 2.27x was observed when running the benchmark code on TGs as opposed to ARM ISSs[1]. The **task** and **IO** benchmarks showed slightly better results than the **pipe** benchmark due the presence of an IP core which is idle for most of the time, in the time lapses between interrupt injections. In addition, the **pipe** benchmark is at a disadvantage due to a higher bus utilization (with six processors, 78% against 63% for **IO** and 38% for **task**), which shifts simulation time emphasis upon the interconnect model.

7 Previous Work

The use of traffic generators to explore NoC architectures is not new. Apart from the ineffective statistical approach presented in [3], in [6, 7], Transaction-Level Modeling (TLM) has been used for bus architecture exploration. The communication is rendered as read and write transactions, which are implemented within the bus model. Depending on the required accuracy of the simulation results, timing information such as bus arbitration delay is annotated within the bus model. In [7] intra-transaction visibility is traded off for a simulation speed gain. While modeling the entire system at a higher abstraction level *i.e.* TLM, both [6] and [7] preserve accuracy with gains in simulation speed. Such models are efficient in capturing regular communication behaviour, but the fundamental problem of capturing system reactiveness in presence of interrupts is not addressed.

Our approach is significantly different from a purely behavioural encapsulation of application code into a simulation device, in analogy with TLM modeling. The TG model we propose is aimed at faithfully replicating traffic patterns generated by a *processor running an application*, not just by the application; this includes *e.g.* accurate modeling of *cache refills* and of latencies between accesses, allowing for cycle-true simulations. In [5], we have successfully and accurately captured core-initiated system behaviour, while in this paper we have attempted to model the processor's response to unpredictable system-initiated communication events. We propose an extensive methodology, that takes into account multitasking and the impact of an underlying OS as seen in realistic applications. To the best of our knowledge, this is the first time that such a light-weight interrupt modeling flow has been adopted to represent a wide range of synchronization patterns. Additionally, we have deployed the flow in a test environment, showing it to be over 98% accurate and proving a speedup that, while nominal, favourably compares to [7].

[1]Benchmarks taken on a multiprocessor Xeon® 1.5 GHz with 12 GB of RAM, thus eliminating any disk swapping or loading effect. Time measurements were taken by averaging over multiple runs.

8 Conclusions

Experimental results proved the viability of a modeling approach which decouples simulation and optimization of IP cores and of interconnect fabrics. Even when tested under complex synchronization scenarios, including asynchronous interrupts involving OS interaction in a multiprocessor environment, the proposed instruction set is able to reproduce IP traffic with full capability to express the application flow. Multiple ways to write programs for this architecture are suggested, and a thorough analysis of one of them is presented. The accuracy of a simulation device providing an implementation of said instruction set is validated in a cycle-true environment by benchmarking multiple applications, additionally achieving a nominal but noticeable simulation speedup.

References

[1] The Real-Time Operating System for Multiprocessor Systems. http://www.rtems.com.

[2] Open Core Protocol Specification, Release 2.0, 2003.

[3] K. Lahiri, A. Raghunathan, and S. Dey. Evaluation of the traffic-performance characteristics of System-on-Chip communication architectures. In *Proceedings of the 14th International Conference on VLSI Design*, pages 29–35, 2001.

[4] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*. IEEE, 2004.

[5] S. Mahadevan, F. Angiolini, M. Storgaard, R. G. Olsen, J. Sparsø, and J. Madsen. A network traffic generator model for fast network-on-chip simulation. In *Proceedings of Design, Automation and Testing in Europe Conference 2005 (DATE05)*. IEEE, March 2005.

[6] O. Ogawa, S. B. de Noyer, P. Chauvet, K. Shinohara, Y. Watanabe, H. Niizuma, T. Sasaki, and Y. Takai. A practical approach for bus architecture optimization at transaction level. In *Proceedings of Design, Automation and Testing in Europe Conference 2003 (DATE03)*. IEEE Computer Society, March 2003.

[7] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proceedings of 38th Design Automation Conference (DAC'04)*, pages 113–118. ACM, 2004.

[8] W. Wolf. *Computers as Components:Principles of Embedded Computing System Design*, chapter 3. Morgan Kaufmann, 2001.

Bibliography

[1] A. Baghdadi and N-E. Zergainoh. Design Space Exploration for Hardware/Software Codesign of Multiprocessor Systems. In *Proceedings of the 11th International Workshop on Rapid System Prototyping (RSP)*, pages 8–13. IEEE, June 2000.

[2] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70–78, January 2002.

[3] A. Bobrek, J. J. Pieper, J. E. Nelson, J. M. Paul, and D. E. Thomas. Modeling shared resource contention using a hybrid simulation/analytical approach. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*, pages 1144–1149. IEEE, Febuary 2004.

[4] Lukai Cai and Daniel Gajski. Transaction level modeling in system level design. CECS technical report 03-10, Center for Embedded Computer Systems, Information and Computer Science, University of California, Irvine, March 2003.

[5] Jon Connell. Arm system-level modeling. Available from ARM website (http:// www.arm.com), June 2003.

[6] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 684–689. IEEE, June 2001.

[7] Franco Fummi, Paolo Gallo, Stefano Martini, Giovanni Perbellini, Massimo Poncino, and Fabio Ricciato. A timing-accurate modeling and simulation environment for networked embedded systems. In *Proceedings of the 42th Design Automation Conference (DAC)*, pages 42–47, June 2003.

[8] Franco Fummi, Stefano Martini, Giovanni Perbellini, Massimo Poncino, Fabio Ricciato, and Maura Turolla. Heterogeneous co-simulation of networked embedded systems. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, Febuary 2004.

[9] Paolo Gai, Luca Abeni, and Giorgio Buttazzo. Multiprocessor DSP Scheduling in System-on-a-chip Architectures. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 231–238. IEEE, June 2002.

[10] A. Gerstlauer, H. Yu, and D.D. Gajski. RTOS modeling for system level design. In *Proceedings of Design, Automation and Test in Europe, DATE'03*, pages 130–135, March 2003.

[11] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[12] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. In *IEE Proceedings - Computers and Digital Techniques*, March 2005.

[13] Jon Jonsson. *The Impact of Application and Architecture Properties on Real-Time Multiprocessor Scheduling*. PhD thesis, School of Electrical and Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, August 1997. Ph.D. Thesis No. 311.

[14] K. Lahiri, A. Raghunathan, and S. Dey. Design space exploration for optimizing on-chip communication architectures. In *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2004.

[15] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*, pages 752–757. IEEE, Febuary 2004.

[16] G. De Micheli, R. Ernst, and W. Wolf. *Readings in Hardware/Software Co-Design*. Morgan Kaufmann, 2001. 1st edition.

[17] OCPIP. Open Core Protocol (OCP) Specification, Release 1.0, 2001.

[18] OCPIP. The importance of sockets in SoC design. White paper downloadable from http://www.ocpip.org, 2003.

[19] Osamu Ogawa, Sylvain Bayon de Noyer, Pascal Chauvet, Katsuya Shinohara, Yoshiharu Watanabe, Hiroshi Niizuma, Takayuki Sasaki, and Yuji Takai. A practical approach for bus architecture optimization at transaction level. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, March 2003.

[20] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proceedings of 38th Design Automation Conference (DAC)*, pages 113–118. ACM, 2004.

[21] P. Pop, P. Eles, and Z. Peng. Analysis and optimization of heterogeneous multiprocessor SoC. In *IEE Proceedings - Computers and Digital Techniques*, March 2005.

[22] K. Richter, M. Jersak, and R. Ernst. A formal approach to mpsoc performance verification. *IEEE Computer*, 36(4):60 – 67, April 2003.

[23] James A. Rowson and Alblerto Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the 34th Design Automation Conference (DAC'97)*, pages 178–183, June 1997.

[24] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems.* Kluwer Academic Publishers, 2004.

[25] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and Alblerto Sangiovanni-Vincentelli. Addressing the system-on-chip interconnect woes through communication-based design. pages 667 – 672.

[26] J. Sifakis. Modeling real-time systems - challenges and work directions. In *EMSOFT, Lecture Notes in Computer Science Vol. 2211*, pages 373–389. October 2001.

[27] Andreas Wieferink, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and Achim Nohl. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*, pages 1256–1261. IEEE Computer Society, Febuary 2004.

[28] Daniel Wiklund. *Development and Performance Evaluation of Networks on Chip.* PhD thesis, Department of Electrical Engineering, Linkoping University, 2005. Dissertation No. 932.