
# Design and Analysis of On-Chip Communication for Network-on-Chip Platforms

**Zhonghai Lu**

Stockholm 2007

*Department of Electronic, Computer and Software Systems*
*School of Information and Communication Technology*
*Royal Institute of Technology (KTH)*
*Sweden*

*Thesis submitted to the Royal Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Technology*

Lu, Zhonghai
Design and Analysis of On-Chip Communication for Network-on-Chip Platforms

ISBN 978-91-7178-580-0
TRITA-ICT/ECS AVH 07:02
ISSN 1653-6363
ISRN KTH/ICT/ECS AVH-07/02 -SE



Royal Institute of Technology
School of Information and Communication Technology
Department of Electronic, Computer and Software Systems
Electrum 229
S-164 40 Kista, Sweden

# Abstract

Due to the interplay between increasing chip capacity and complex applications, System-on-Chip (SoC) development is confronted by severe challenges, such as managing deep submicron effects, scaling communication architectures and bridging the productivity gap. Network-on-Chip (NoC) has been a rapidly developed concept in recent years to tackle the crisis with focus on network-based communication. NoC problems spread in the whole SoC spectrum ranging from specification, design, implementation to validation, from design methodology to tool support. In the thesis, we formulate and address problems in three key NoC areas, namely, *on-chip network architectures*, *NoC network performance analysis*, and *NoC communication refinement*.

Quality and cost are major constraints for micro-electronic products, particularly, in high-volume application domains. We have developed a number of techniques to facilitate the design of systems with low area, high and predictable performance. From flit admission and ejection perspective, we investigate the area optimization for a classical wormhole architecture. The proposals are simple but effective. Not only offering unicast services, on-chip networks should also provide effective support for multicast. We suggest a connection-oriented multicasting protocol which can dynamically establish multicast groups with quality-of-service awareness. Based on the concept of a logical network, we develop theorems to guide the construction of contention-free virtual circuits, and employ a back-tracking algorithm to systematically search for feasible solutions.

Network performance analysis plays a central role in the design of NoC communication architectures. Within a layered NoC simulation framework, we develop and integrate traffic generation methods in order to simulate network performance and evaluate network architectures. Using these methods, traffic patterns may be adjusted with locality parameters and be configured per pair of tasks. We propose also an algorithm-based analysis method to estimate whether a wormhole-switched network can satisfy the timing constraints of real-time messages. This method is built on traffic assumptions and based on a contention tree model that captures

direct and indirect network contentions and concurrent link usage.

In addition to NoC platform design, application design targeting such a platform is an open issue. Following the trends in SoC design, we use an abstract and formal specification as a starting point in our design flow. Based on the synchronous model of computation, we propose a top-down communication refinement approach. This approach decouples the tight global synchronization into process local synchronization, and utilizes synchronizers to achieve process synchronization consistency during refinement. Meanwhile, protocol refinement can be incorporated to satisfy design constraints such as reliability and throughput.

The thesis summarizes the major research results on the three topics.

# Table of Contents

# Acknowledgements

Studying towards Ph.D. takes five years, with 20% teaching workload. It is a long process filled with mixed feelings, pleasure and pressure, satisfaction and disappointment. The pleasure and enjoyment originate from the persistent development of innovative ideas in the frontline of the interesting research area. The pressure may undergo in the face of frequent deadlines for tasks and papers, especially during the tight finance period in the department. The satisfaction comes often for the recognition of a piece of innovative work while the disappointment may occur for the sake of mis-understanding and rejections. My life as a Ph.D. student is a colorful picture, which composes a beautiful and important part of my life.

While this picture is painted, a lot of people have contributed in various ways. It is the right time and place to acknowledge their help. Professor Axel Jantsch is the one I thank most. I thank him far more just because of the fact that he is my supervisor. He is a very respectable person for his personality, knowledge and creativity. I am lucky to be one of his students. In reality, he treats his students very equally. He is not only a supervisor but also a colleague and a collaborator. I am indebted to Dr. Ingo Sander. He has been acting as the co-supervisor for my Ph.D. study, and helping me in all the ways possible. He is concerned with not only my progress in research but also my office and health. We have had many and many small talks and discussions, which are sources of friendship and inspiration. I thank Professor Shashi Kumar, who was my co-supervisor before he left KTH.

I acknowledge valuable discussions and diverse help from all my colleagues in the System, Architecture and Methodology (SAM) group, particularly, the two project teams, *Nostrum* and *ForSyDe*, where I have been involved in. The Nostrum team investigates network-on-chip architectures and associated design techniques. The present and past contributors include Mikael Millberg, Rikard Thid, Erland Nilsson, Raimo Haukilahti, Johnny Öberg, Kim Petersen and Per Badlund. Particularly, I thank Rikard for his original work in the layered NoC simulation kernel. The ForSyDe team aims to develop a formal design methodology for System-on-Chip applications from modeling to implementation and verification. This team

involves Ingo Sander, Tarvo Raudvere, Ashish Kumar Singh and Jun Zhu.

I appreciate our system group, Hans Berggren and Peter Magnusson, for their active and patient support in computer and network systems. I thank secretaries Lena Beronius, Agneta Herling and Rose-Marie Lövenstig for their administrative assistance in traveling and other issues.

I thank all other colleagues in the Department of Electronic, Computer and Software Systems for their various help and for the pleasant and encouraging environment we contribute to and share. I thank Roshan Weerasekera for friendship. Special thanks should go to all my Chinese colleagues, particularly to Lirong, Jian Liu, Li Li, Bingxin, and Jinliang, for the great occasions and happiness we share.

During my Ph.D. study period, I have supervised twelve Master theses. These works have deepened my understanding on the corresponding subjects and most of them are excellent. I thank all the students for their hard and fruitful work, particularly, Bei Yin, Mingchen Zhong, Li Tong, Karl-Henrik Nielsen, Jonas Sicking and Ming Liu.

I have taken an internship in Samsung Electronics in the summer of 2005. During the three-month period, I investigated the state-of-the-art interconnect techniques. I thank Mr. Soo Kwan Eo, Dr. Cheung and Dr. Yoo and all others in the system design technology group for their arrangements and assistance.

Finally I give my deepest gratitude to my family, my wife Yanhong and daughter Lingyi. I could not count how many weekends I have spent with my computer, and how many times I have been late back home. Any piece of my achievement has an invisible part of their contribution. I thank my brothers and sisters in China for their endless concerns. I thank my parents for their permanent love and irreplaceable support.

The research presented in the dissertation is financed by the Swedish government within the SoCware program and the European Commission within the Sprint project.

Zhonghai Lu

December 2006, Stockholm

# List of Publications

**Part A. Papers included in the thesis:**

- **NoC Network Architectures**

1. Zhonghai Lu and Axel Jantsch. Flit admission in on-chip wormhole-switched networks with virtual channels. In *Proceedings of the International Symposium on System-on-Chip*, pages 21-24, Tampere, Finland, November 2004.

2. Zhonghai Lu and Axel Jantsch. Flit ejection in on-chip wormhole-switched networks with virtual channels. In *Proceedings of the IEEE NorChip Conference*, pages 273-276, Oslo, Norway, November 2004.

3. Zhonghai Lu, Bei Yin, and Axel Jantsch. Connection-oriented multicasting in wormhole-switched networks on chip. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'06)*, pages 205-210, Karlsruhe, Germany, March 2006.

4. Zhonghai Lu and Axel Jantsch. TDM virtual-circuit configuration in network-on-chip using logical networks. In submission to *IEEE Transactions on Very Large Scale Integration Systems*.

- **NoC Network Performance Analysis**

5. Zhonghai Lu and Axel Jantsch. Traffic configuration for evaluating networks on chip. In *Proceedings of the 5th International Workshop on System-on-Chip for Real-time Applications*, pages 535-540, Alberta, Canada, July 2005.

6. Zhonghai Lu, Mingchen Zhong, and Axel Jantsch. Evaluation of on-chip networks using deflection routing. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI'06)*, pages 296-301, Philadelphia, USA, May 2006.

7. Zhonghai Lu, Axel Jantsch and Ingo Sander. Feasibility analysis of messages for on-chip networks using wormhole routing. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'05)*, pages 960-964, Shanghai, China, January 2005.

- **NoC Communication Refinement**

8. Zhonghai Lu, Ingo Sander, and Axel Jantsch. Refining synchronous communication onto network-on-chip best-effort services. In Alain Vachoux, editor, *Applications of Specification and Design Languages for SoCs - Selected papers from FDL 2005*, Chapter 2, pages 23-38, Springer, 2006.

9. Zhonghai Lu, Ingo Sander, and Axel Jantsch. Towards performance-oriented pattern-based refinement of synchronous models onto NoC communication. In *Proceedings of the 9th Euromicro Conference on Digital System Design (DSD'06)*, pages 37-44, Dubrovnik, Croatia, August 2006.

**Part B. Publications not included in the thesis:**

10. Zhonghai Lu, Ingo Sander, and Axel Jantsch. Refinement of a perfectly synchronous communication model onto Nostrum NoC best-effort communication service. In *Proceedings of the Forum on Specification and Design Languages (FDL'05)*, Lausanne, Switzerland, September 2005.

11. Zhonghai Lu, Li Tong, Bei Yin, and Axel Jantsch. A power-efficient flit-admission scheme for wormhole-switched networks on chip. In *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, Florida, U.S.A., July 2005.

12. Zhonghai Lu, Rikard Thid, Mikael Millberg, Erland Nilsson, and Axel Jantsch. NNSE: Nostrum network-on-chip simulation environment. In *Proceedings of Swedish System-on-Chip Conference*, Stockholm, Sweden, April 2005.

13. Zhonghai Lu, Rikard Thid, Mikael Millberg, Erland Nilsson, and Axel Jantsch. NNSE: Nostrum network-on-chip simulation environment. In *The University Booth Tool-Demonstration Program of the Design Automation and Test in Europe Conference*, Munich, Germany, March 2005.

14. Ingo Sander, Axel Jantsch, and Zhonghai Lu. Development and application of design transformations in ForSyDe. *IEE Proceedings - Computers & Digital Techniques*, 150(5):313-320, September 2003.

15. Zhonghai Lu and Axel Jantsch. Network-on-chip assembler language (version 0.1). Technical Report TRITA-IMIT-LECS R 03:02, ISSN 1651-4661, ISRN KTH/IMIT/LECS/R-03/02-SE, Royal Institute of Technology, Stockholm, Sweden, June 2003.

16. Ingo Sander, Axel Jantsch, and Zhonghai Lu. Development and application of design transformations in ForSyDe. In *Proceedings of Design, Automation and Test in Europe Conference*, pages 364-369, Munich, Germany, March 2003.

17. Zhonghai Lu and Raimo Haukilahti. NoC application programming interfaces. In Axel Jantsch and Hannu Tenhunen, editors, *Networks on Chip*, Chapter 12, pages 239-260. Kluwer Academic Publishers, February 2003.

18. Zhonghai Lu, Ingo Sander, and Axel Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS'02)*, pages 86-91, Kyoto, Japan, October 2002.

19. Zhonghai Lu and Axel Jantsch. Admitting and Ejecting Flits in Wormhole-switched On-chip Networks. In submission to *Journal of Systems Architectures* (under the second round review).

# List of Figures

# Abbreviations

| | |
|---|---|
| ASIC | Application-Specific Integrated Circuit |
| AXI | Advanced eXtensible Interface |
| ALI | Application-Level Interface |
| CAD | Computer Aided Design |
| CLI | Core-Level Interface |
| CMOS | Complementary Metal Oxide Semiconductor |
| CT | Contention Tree |
| DSM | Deep SubMicron |
| DTL | Device Transaction Level |
| ESL | Electronic System Level |
| FIFO | First In First Out |
| ForSyDe | FORmal SYstem DEsign |
| FPGA | Field-Programmable Gate Array |
| GUI | Graphical User Interface |
| ITRS | International Technology Roadmap for Semiconductors |
| IP | Intellectual Property |
| LN | Logical Network |
| MoC | Model of Computation |
| NNSE | Nostrum Network-on-Chip Simulation Environment |
| NoC | Network on Chip |
| OCP | Open Core Protocol |
| PC | Physical Channel |
| PE | Processing Element |
| QoS | Quality of Service |
| RNI | Resource Network Interface |
| RT | Real Time |
| RTL | Register Transfer Level |
| SEMLA | Simulation EnvironMent for Layered Architecture |
| SoC | System on Chip |
| TDM | Time-Division Multiplexing |
| ULSI | Ultra Large Scale Integration |
| VC | Virtual Channel / Virtual Circuit |
| VCI | Virtual Component Interface |

# Chapter 1

# Introduction

*This chapter highlights System-on-Chip design challenges and introduces the Network-on-Chip concept. We also give an overview of the research presented in the thesis and outline the author's contributions to the enclosed papers.*

## 1.1 Network-on-Chip (NoC)

### 1.1.1 System-on-Chip (SoC) Design Challenges

Our life has been largely shaped by the exciting developments of modern electronic technologies, such as pervasive and ubiquitous computing, ambient intelligence, communication, and Internet. Today micro-electronic products are influencing the ways of communication, learning and entertainment. The key driving force for the developments during decades is the System-on-Chip (SoC) technologies, where complex applications are integrated onto single ULSI chips. Not only functionally enriched, these products such as mobile phones, notebooks and personal handheld sets are becoming faster, smaller-in-size, larger-in-capacity, lighter-in-weight, lower-in-power-consumption and cheaper. One could favorably think that this trend will persistently continue. Following this trend, we could integrate more and more complex applications and even systems onto a single chip. However, our current methodologies for SoC design and integration do not evenly advance due to the big challenges confronted.

- *Deep SubMicron (DSM) effects* [43, 80, 134]: In early days of VLSI design, signal integrity effects such as interconnect delay, crosstalk, inter-symbol interference, substrate coupling, transmission-line effects, etc. were negligible

due to relatively slow clock speed and low integration density. Chip interconnect was reliable and robust. At the scale of 250 $nm$ with aluminum and 180 $nm$ with copper and below, interconnect started to become a dominating factor for chip performance and robustness. As the transistor density is increased, wires are getting neither fast nor reliable [43]. More noise sources due to inductive fringing, crosstalk and transmission line effects are coupled to other circuit nodes globally on the chip via the substrate, common return ground and electromagnetic interference. More and more aggressive use of high-speed circuit families, for example, domino circuitry, scaling of power supply and threshold voltages, and mixed-signal integration combine to make the chips more noise-sensitive. Third, higher device densities and faster switching frequencies cause larger switching-currents to flow in the power and ground networks. Consequently, power supply is plagued with excessive IR voltage drops as wells as inductive voltage drops over the power distribution network and package pins. Power supply noise degrades not only the driving capability of gates but also causes possible false switching of logical gates. Today signal and power integrity analysis is as important as timing, area and power analysis.

- *Global synchrony* [3, 47]: Predominating digital IC designs have been following a globally synchronous design style where a global clock tree is distributed on the chip, and logic blocks function synchronously. However, this style is unlikely to survive with future wire interconnect. The reason is that technology scaling does not treat wire delay and gate delay equally. While gate delay (transistor switching time) has been getting dramatically smaller in proportion to the gate length, wires have slowed down. As the chip becomes communication-bound at 130 $nm$, multiple cycles are required to transmit a signal across its diameter. As estimated in [3], with the process technology of 35 $nm$ in year 2014, the latency across the chip in a top-level metal wire will be 12 to 32 cycles depending on the clock rate assuming best transmission conditions such as very low-permittivity dielectrics, resistivity of pure copper, high aspect ratio (ratio of wire height to wire width) wires and optimally placed repeaters. Moreover, a clock tree is consuming larger portions of power and area budget and clock skew is claiming an ever larger portion of the total cycle time [94]. Even if we have an unlimited number of transistors on a chip, chip design is to be constrained by communication rather than capacity. A future chip is likely to be partitioned into locally synchronous regions but global communication is asynchronous, so called GALS (Globally Asynchronous Locally Synchronous).

- *Communication architecture* [9, 19]: Most current SoCs have a bus-based architecture, such as simple, hierarchical or crossbar-type buses. In contrast to the scaling of chip capacity, buses do not scale well with the system size in terms of bandwidth, clocking frequency and power. First, a bus system has very limited concurrent communication capability since only one device can drive a bus segment at a time. Current SoCs integrate fewer than five processors and, rarely, more than 10 bus masters. Second, as the number of clients grows, the intrinsic resistance and capacitance of the bus also increase. This means that the bus speed is inherently difficult to scale up. Third, a bus is inefficient in energy since every data transfer is broadcast. The entire bus wire has to be switched on and off. This means that the data must reach each receiver at great energy cost. Although improvements such as split-transaction protocols and advanced arbitration schemes for buses have been proposed, these incremental techniques can not overcome the fundamental problems. To explore the future chip capacity, for high-throughput and low-power applications, hundreds of processor-sized resources must be integrated. A bus-based architecture would become a critical performance and power bottleneck due to the scalability problem. Novel on-chip communication architectures are desired.

- *Power and thermal management* [85, 105]: As circuits run with higher and higher frequencies, lowering power consumption is becoming extremely important. Power is a design constraint, which is no more subordinate to performance. Despite process and circuit improvements, power consumption shows rapid growth. Equally alarming is the growth in power density on the chip die, which increases linearly. In face of DSM effects, reducing power consumption is becoming even more challenging. As devices shrink to submicron dimensions, the supply voltage must be reduced to avoid damaging electric fields. This development, in turn, requires a reduced threshold voltage. However, leakage current increases exponentially with a decrease in the threshold voltage. In fact, a 10% to 15% reduction can cause a two-fold increase in leakage current. In increasingly smaller devices, leakage will become the dominant source of power consumption. Further, leakage occurs as long as power flows through the circuit. This constant current can produce an increase in the chip temperature, which in turn causes an increase in the thermal voltage, leading to a further increase in leakage current.

- *Verification* [107, 111]: Today SoC design teams are struggling with the complexity of multimillion gate designs. System verification runs through

the whole design process from specification to implementation, typically with formal methods or simulation-based validation. As the system has become extremely complex, the verification or validation consumes an increasing portion of the product development time. The verification effort has reached as high as 70% of engineering efforts.

- *Productivity gap* [4, 111]: Simply put, productivity gap is the gap between what we are capable of building and what we are capable of designing. In line with Moore's law [83], the logic capacity of a single chip has increased at the rate of 58% per annum compounded. Soon the complexity of the chip enters the billion-transistor era. The complexity of developing SoCs is increasing continuously in order to exploit the potential of the chip capacity. However, the productivity of hardware and software design is not growing at a comparable pace. The hardware design productivity is increased at a rate in the range 20% to 25% per annum compounded. Even worse, the software design productivity improves at a rate in the range from 8% to 10% per annum compounded. As a consequence, the costs of developing advanced SoCs are increasing at an alarming pace and time-to-market is negatively affected. The design team size is increased by more than 20% per year. This huge investment is becoming a serious threshold for new product developments and is slowing down the innovation in the semiconductor industry. As stated in the ITRS roadmap [4], cost of design is the greatest threat to continuation of the semiconductor roadmap.

### 1.1.2 Network-on-Chip as a SoC Platform

Innovations occur where challenges are present. Network-on-Chip (NoC) was proposed in face of those challenges in around year 2001 in the SoC community [9, 27, 37, 41, 109, 119]. In March 2000, packet-switched networks were proposed in SPIN [37] as a global and scalable SoC interconnection. The term *Network-on-Chip* appeared initially in November 2000 [41] where NoC was proposed as a platform to cope with the productivity gap. In June 2001, Dally and Towles proposed NoC as a structured way of communication to connect IP modules [27]. The GigaScale Research Center (GSRC) suggested NoC to address interconnection woes [119]. In October 2001, researchers from the Philips Research presented a router architecture supporting both best-effort and guaranteed-throughput traffic for Network-on-Silicon [109]. In January 2002, Luca and De Micheli formulated NoC as a new SoC paradigm [9]. While network-on-chip is still in its infancy, the

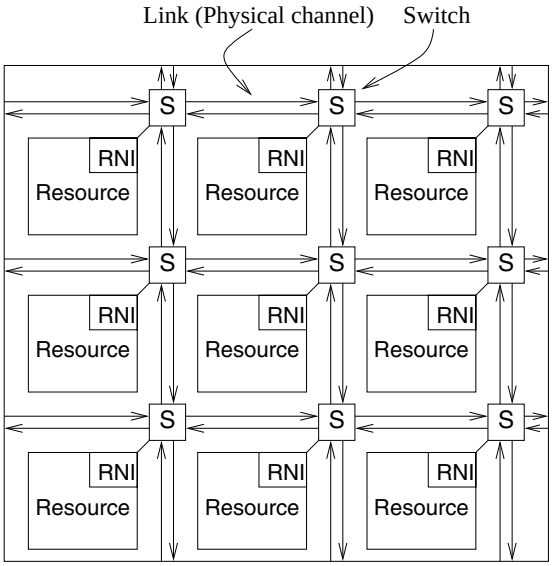



**Figure 1.1.** A mesh NoC with 9 nodes

concept has spread and been accepted in academia very rapidly. Some big companies, for instance, NXP semiconductors (former part of Philips Semiconductors) and ST Micro-electronics, are also very active in this field [34, 52]. A comprehensive survey on current research and practices of NoC can be found in [13].

Aimed to be a systematic approach, NoC proposes networks as a scalable, reusable and global communication architecture to address the SoC design challenges. As an instance, *Nostrum* [81, 90] is the name of the Network-on-Chip concept developed at the Royal Institute of Technology (KTH), Sweden. It features a mesh structure composed of switches with each resource connected to exactly one switch, as shown in Figure 1.1. A resource can be a processor, memory, ASIC, FPGA, IP block or a bus-based subsystem. The resources are placed on the slots formed by the switches. The maximal resource area is defined by the maximal synchronous region of a technology. The resources perform their own computational, storage and/or I/O processing functionalities, and are equipped with Resource-Network-Interfaces (RNIs) to communicate with each other by routing packets instead of driving dedicated wires.

Communication network is a well-known concept developed in the context of telephony, computer communication as well as parallel machines. On-chip networks share many characteristics with these networks, but also have significant differences. For clear presentation, throughout the thesis, we also call an on-chip

network a *micro-network*, a parallel-machine network a *macro-network*, a telephony or computer network a *tele-network*. On-chip networks are developed on a single chip and designed for closed systems targeting perhaps heterogeneous applications. Parallel-machine networks are developed on distributed boards and designed for a particular application which typically executes specific algorithms. Computer networks are geographically distributed and designed for open systems running diverse applications from client-server, peer-to-peer and multicast applications. Telephony networks are also geographically distributed but are designed mainly for the purposes of communicating voice, video and data. The design of a closed system allows for customization in which the network properties including the network-level, link-level and physical-level properties can be propagated to the application level and both communication and computation may be efficiently optimized. Since a micro-network is built on a single chip, it can have wide parallel wires and allows high rate synchronous clocking. On the other hand, it has more stringent constraints in performance, area and power, which are typical trade-off considerations for SoC designs. As communication is to transfer data, timing is the first-level citizen. On-chip networks have the strictest requirement on delay and jitter. The time scale is measured in nano seconds. This requirement precludes many of the software-based sophisticated arbitration, routing and flow-control algorithms. Cost is a major concern for on-chip networks since most SoCs target high-volume markets. The buffering in an on-chip network has very limited space and is expensive in comparison with board-level, local-area and wide-area networks. This means that a NoC allows a limited count of routing tables and virtual-channel buffers in network nodes. Power consumption is important for all kinds of networks. However, on-chip networks are developed also for embedded applications with battery-driven devices. Such applications require extremely low power which is not comparable to large-scale networks. As we also mentioned, on-chip network designs are confronted by the DSM effects. Taming bad physical effects is as important as network design itself. Furthermore, many SoC networks are developed as a platform for multiple use cases, not only for a single use case. Therefore designing micro-networks also need to take reconfigurability into account.

As we view it, Network-on-Chip is a revolutionary rather than evolutionary approach to address the SoC design crisis. It shifts our focus from computation to communication. It should take interconnect into early consideration in the design process, and might favor a meet-in-the middle (platform-based) design methodology against a top-down or bottom-up approach. NoC has the following features:

- *Interconnect-aware* [93]: As the technology scales, the reachable region in one clock cycle diminishes [3]. Consequently, chip design is increasingly

becoming communication-bound rather than capacity-bound. Since the size of a single module is limited by the reachable region in one cycle, to exploit the huge chip capacity, the entire chip has to be partitioned into multiple regions. A good partitioning should be regular, making it easier to manage the properties of long wires including middle-layer and top-layer wires. Each module is situated in one partitioned region and maintains its own synchronous region. In this way, the reliance on global synchrony and use of global wires can be alleviated. To guarantee correct operation, registers may be used in wire segments to make the design latency-insensitive [17]. Besides, each IP may be attached to a switch. Switches are in turn connected with each other to route packets in the network. The signal and power integrity issues may be addressed at the physical, link and higher layers. For example, redundancy in time, space and information can be incorporated in transmission to achieve reliability. By physically structuring the communication and successfully suppressing the DSM effects, the design robustness and reliability can be improved.

- *Communication-centric* [10]: Networking distributed IP modules in a partitioned chip results in a naturally parallel communication infrastructure. As long as the chip capacity is not exceeded, the number of cores which can be integrated on a single chip is scalable. The inter-core communications share the total network bandwidth with a high degree of concurrency. The network can be dimensioned to suit the bandwidth need of the application under interest. The parallel architecture allows concurrent processing in computation and communication. This helps to leverage performance and reduce power in comparison with a sequential architecture permitting only sequentialized processing. A protocol stack is typically built to abstract the network-based communication. Each layer has well-defined functionalities, protocols and interfaces. The design space at each layer has to be sufficiently explored. The tradeoffs between performance and cost should be considered in the design, analysis and implementation of the communication architecture. Quality-of-Service (QoS) and system-wide performance analysis are central issues to address predictability.

- *Platform-based* [50, 87]: Since the cost of design is the major obstacle for innovative and complex SoCs [46], developing a programmable, reconfigurable and extensible communication platform is essential for SoC designs. To this end, NoC shall serve as a communication and integration platform

providing a hardware communication architecture, an associated interconnect interface, as well as a high-level interface for integrating hardware IPs, custom logic and for software programming. This enables the architecture-level reuse. One challenge is to address the balance between generality and optimality. A platform must serve not only one application but also many applications within an application domain. On the other hand, customization to enhance performance and efficiency is needed to make designs competitive. Providing well-defined interfaces at least at the network level and the application level is important, because it enables IPs and functional blocks to be reusable. Interface standardization is one major concern to make IPs from different vendors exchangeable. It must be efficient and also addresses legacy IPs. The concept of interface-based design has been shown successful for IP plug-and-play in the history of software and hardware developments, for example, instruction sets and various interconnect buses or protocols such as Peripheral Component Interface (PCI) and Universal Serial Bus (USB). A NoC design methodology should also favor communication interfaces for the greatest possible IP reuse and integration [112, 129]. Using validated components and architectures in a design flow shrinks verification effort, reduces time-to-market and guarantees product quality, thus enhancing design productivity.

As such, NoC research does not deal with only several aspects of SoC design but creates a new area [50]. The term NoC is used today mostly in a very broad meaning. It encompasses the hardware communication infra-structure, the middleware and operating system, application programming interfaces [64, 101], the design methodology and its associated tool chain. The challenges for NoC research have thus been distributed in all aspects of SoC design from architecture to performance analysis, from traffic characterization to application design.

### 1.1.3 On-Chip Communication Model

On-chip communication is to provide a means to enable interprocess communication with a set of constraints and properties satisfied. A good view of network-based process-to-process communication is to follow the ISO's OSI model [135]. The seven-layer model was proposed to interconnect open systems, which are heterogeneous and distributed. The layered structure decomposes the communication problem into more manageable components at different hierarchical layers. Rather than a monolithic structure, several layers are designed, each of which solves one part of the problem. Besides, layering provides a more modular design. At each

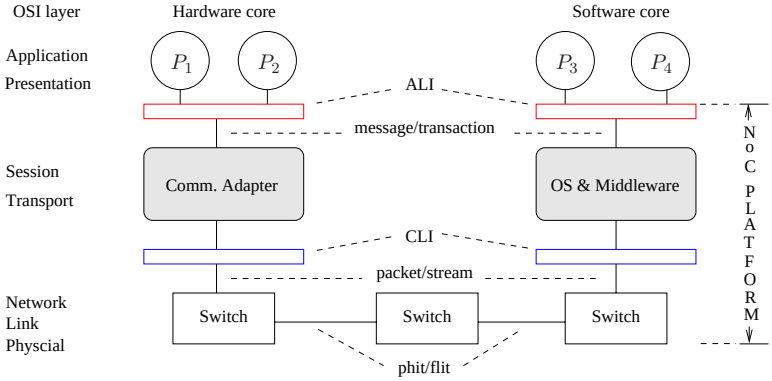



**Figure 1.2.** On-chip communication layers with Application Level Interface (ALI) and Core Level Interface (CLI)

layer, protocols and services, which are implementation-independent, are well-defined. Peer entities at the same layer can thus communicate with each other transparently. Adding new services to one layer may only need to modify the functionality at one layer, reusing the functions provided at all the other layers. Due to these advantages, several NoC groups [9, 82, 119] have followed this model and adapted it to build a protocol stack for on-chip communication.

As a platform, NoC shall provide well-defined interfaces for application programming and IP integration. Two levels of interfaces can be identified. One is the *Core-Level Interface (CLI)*, which is used to connect hardware cores. At this level, IPs and processors implement interfaces such as AXI [6], OCP [95], VCI [130], CoreConnect [45] and DTL [104]. The other level interface is for integrating hardware logic via a communication adapter and for programming embedded software. The Operating System (OS) [92] and middleware can be part of the platform. This level of interface is *Application-Level Interface (ALI)*. A recent proposal of the two-level interfaces for multiprocessors on chip can be found in [129].

An on-chip communication model combines the two views: *abstract layered communication* and *interface-based communication*. Although having been discussed separately, the two views are coherent, as shown in Figure 1.2. As can be seen, hardware and software processes (illustrated as $P_1$, $P_2$, $P_3$, $P_4$ in Figure 1.2) representing the application layer use the ALI. The hardware communication adapter for integrating hardware cores and operating system & middleware for integrating software cores realize the session and transport layers, and connect to the CLI. The CLI encapsulates the network. It is worth noting that bypassing one layer is possible, as long as the interfaces match. For example, if a hardware IP imple-

ments the CLI, it can be directly connected to the CLI instead of connecting to the ALI, bypassing the communication adapter.

## 1.2 Research Overview

We have been orienting our NoC research towards three key issues: *on-chip network architectures*, *network performance analysis* and *application design methodology*. The network communication architectures deal with the design of on-chip networks. The performance analysis evaluates the network performance and helps to uncover the impact of network parameters on performance. The design methodology is concerned with how to design applications on a NoC platform. Specifically, we deal with communication refinement that synthesizes the communication in a system model into on-chip communication. Essentially these topics deal with the design and analysis of on-chip communication for NoC platforms.

We have identified and formulated problems related to the three aspects mentioned above. The thesis is based on the research results from these studies. In the following, we give a brief sketch of the main results:

- *NoC network architectures*: We have proposed cost-effective switch architectures, a connection-oriented multicasting scheme, as well as a TDM (Time Division Multiplexing) virtual-circuit configuration method using logical networks. After studying wormhole switch micro-architectures, we propose flit admission and ejection schemes, which are cost-effective with minimal performance penalty. Our multicasting mechanism is also proposed for wormhole-switched networks. It is connection-oriented, and a connection can be established dynamically. Based on the concept of a logical network, we have developed theorems and used a back-tracking algorithm to configure contention-free TDM virtual-circuits.

- *NoC network performance analysis*: We have investigated traffic configuration, carried out network simulation and made feasibility analysis. We propose how to configure synthetic traffic patterns using distribution with controllable locality or channel-by-channel customization. This traffic configuration method has been integrated into our Nostrum NoC Simulation Environment (NNSE). A case study on the deflection networks shows that our simulator enables to explore the architectural design space and helps to make proper decisions on topology, routing schemes and deflection policies. The feasibility analysis aids designers with information about whether the application can fulfill the timing requirements of messages on the network and

how efficient network resources can be utilized. It allows one to evaluate the network using algorithm instead of simulation. Hence, it is more efficient but less accurate. This feasibility analysis is performed on wormhole-switched networks.

- *NoC communication refinement*: Based on a synchronous system model, we have proposed a communication refinement approach that refines the abstract communication into network-based communication. During the refinement, synchronization consistency is maintained in order to be correct-by-construction and protocol refinement can be incorporated to satisfy performance constraints.

Next, we summarize the author's contributions in each of the enclosed papers.

## 1.3 Author's Contributions

The thesis is based on a collection of papers, which are all peer-reviewed except Paper 4 that is under review. The papers are grouped into three blocks, namely, *NoC network architectures*, *NoC network performance analysis*, and *NoC communication refinement*. Each block is dedicated to one chapter in the thesis and we concentrate on introducing the author's contributions in each chapter. The detailed materials, experiments, results and other related work are referred to the papers. In the following, we summarize the enclosed papers highlighting the author's contributions. These papers are also listed in the references.

- **NoC Network Architectures**

  **Paper 1 [66].** Zhonghai Lu and Axel Jantsch. Flit admission in on-chip wormhole-switched networks with virtual channels. In *Proceedings of the International Symposium on System-on-Chip*, pages 21-24, Tampere, Finland, November 2004.

  This paper discusses the flit admission problem in input-buffering and output-buffering wormhole switches. Particularly it presents a novel cost-effective coupling scheme that binds flit admission queues with output physical channels in a one-to-one correspondence manner. The experiments suggest that the network performance is equivalent to the base line scheme which connects a flit admission queue to all the output physical channels.

  *Author's contributions*: The author contributed with the problem formulation, conducted experiments and wrote the manuscript.

**Paper 2 [67].** Zhonghai Lu and Axel Jantsch. Flit ejection in on-chip wormhole-switched networks with virtual channels. In *Proceedings of the IEEE NorChip Conference*, pages 273-276, Oslo, Norway, November 2004.

This paper studies flit ejection models in a wormhole virtual channel switch. Instead of the costly ideal flit-ejection model, two alternatives which largely reduce the buffering cost are proposed. Experiments show that the $p$-sink model achieves nearly equivalent performance with the ideal sink model if the network is not overloaded.

*Author's contributions*: The author formulated the flit-ejection problem, proposed solutions, conducted experiments and wrote the manuscript.

**Paper 3 [75].** Zhonghai Lu, Bei Yin, and Axel Jantsch. Connection-oriented multicasting in wormhole-switched networks on chip. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'06)*, pages 205-210, Karlsruhe, Germany, March 2006.

This paper presents a connection-oriented multicast scheme in wormhole-switched NoCs. In this scheme, a multicast procedure consists of establishment, communication and release phases. A multicast group can request to reserve virtual channels during establishment and has priority on arbitration of link bandwidth. This multicasting method has been effectively implemented in a mesh network with deadlock freedom. Our experiments show that the multicast technique improves throughput, and does not exhibit significant impact on unicast performance in a network with mixed unicast and multicast traffic.

*Author's contributions*: The author contributed with the idea and protocol design, suggested experimentation methods, and wrote the manuscript. The implementation and experiments were conducted by Bei Yin.

**Paper 4 [65].** Zhonghai Lu and Axel Jantsch. TDM virtual-circuit configuration in network-on-chip using logical networks. In submission to *IEEE Transactions on Very Large Scale Integration Systems*.

Configuring Time-Division-Multiplexing (TDM) Virtual Circuits (VCs) on network-on-chip must guarantee conflict freedom for VCs besides allocating sufficient time slots to them. Using the generalized concept of logical networks, we develop and prove theorems that constitute sufficient and necessary conditions to establish conflict-free VCs. Moreover, we give a formulation of the multi-node VC configuration prob-

lem and suggest a back-tracking algorithm to find solutions by constructively searching the solution space.

*Author's contributions*: The author developed and proved the theorems, formulated the problem, wrote the program, conducted experiments and wrote the manuscript.

- **NoC Network Performance Analysis**

**Paper 5 [68].** Zhonghai Lu and Axel Jantsch. Traffic configuration for evaluating networks on chip. In *Proceedings of the 5th International Workshop on System-on-Chip for Real-time Applications*, pages 535-540, Alberta, Canada, July 2005.

This paper details the traffic configuration methods developed for NNSE. It presents a unified expression to configure both uniform and locality traffic and proposes application-oriented traffic configuration for on-chip network evaluation.

*Author's contributions*: The author formulated the unified expression for the regular traffic patterns and defined application-oriented traffic, integrated the methods in NNSE, conducted experiments and wrote the manuscript.

**Paper 6 [76].** Zhonghai Lu, Mingchen Zhong, and Axel Jantsch. Evaluation of on-chip networks using deflection routing. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI'06)*, pages 296-301, Philadelphia, USA, May 2006.

This paper evaluates the performance of deflection networks with different topologies such as *mesh*, *torus* and *Manhattan Street Network*, different routing algorithms such as *random*, *dimension XY*, *delta XY* and *minimum deflection*, as well as different deflection policies such as *non-priority*, *weighted priority* and *straight-through* policies. The results suggest that the performance of a deflection network is more sensitive to its topology than the other two parameters. It is less sensitive to its routing algorithm, but a routing algorithm should be minimal. A priority-based deflection policy that only uses global and history-related criterion can achieve both better average-case and worst-case performance than a non-priority or priority policy that uses local and stateless criterion. These findings may be used as guidelines by designers to make right decisions on the deflection network architecture.

*Author's contributions*: The author formulated the problem, proposed solution schemes, and wrote the manuscript. The implementation and experiments were conducted by Mingchen Zhong.

**Paper 7 [69].** Zhonghai Lu, Axel Jantsch and Ingo Sander. Feasibility analysis of messages for on-chip networks using wormhole routing. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 960-964, Shanghai, China, January 2005.

The paper proposes a method for investigating the feasibility of delivering mixed real-time and nonreal-time messages in wormhole-switched networks. Particularly it describes a contention tree model for the estimation of worst-case performance for delivering real-time messages.

*Author's contributions*: The author formulated the contention tree model, developed the algorithm, wrote the program, performed experiments, and wrote the manuscript.

- **NoC Communication Refinement**

**Paper 8 [71].** Zhonghai Lu, Ingo Sander, and Axel Jantsch. Refining synchronous communication onto network-on-chip best-effort services. In Alain Vachoux, editor, *Applications of Specification and Design Languages for SoCs - Selected papers from FDL 2005*. Springer, Chapter 2, pages 23-38, 2006.

The paper proposes a top-down design flow to refine synchronous communication onto NoC best-effort services. It consists of three steps, namely, *channel refinement*, *process refinement*, and *communication mapping*. In channel refinement, synchronous channels are replaced with stochastic channels abstracting the best-effort service. In process refinement, processes are refined in terms of interfaces and synchronization properties. Particularly, we use *synchronizers* to maintain local synchronization of processes and thus achieve *synchronization consistency*, which is a key requirement while mapping a synchronous model onto an asynchronous architecture. Within communication mapping, the refined processes and channels are mapped onto a NoC platform. A digital equalizer is used as a tutorial example and implemented in the *Nostrum* NoC platform to illustrate the feasibility of our concepts.

*Author's contributions*: The author proposed the design flow for the communication refinement, developed solutions for the synchronization problem, conducted the case study, and wrote the manuscript.

**Paper 9 [72].** Zhonghai Lu, Ingo Sander, and Axel Jantsch. Towards performance-oriented pattern-based refinement of synchronous models onto NoC communication. In *Proceedings of the 9th Euromicro Conference on Digital System Design (DSD'06)*, pages 37-44, Dubrovnik, Croatia, August 2006.

This paper is complementary to Paper 8, which mainly discusses how to maintain synchronization consistency while refining the synchronous communication on asynchronous NoC architectures. This paper focuses on how to achieve performance-oriented refinement. Specifically, it deals with *protocol refinement* and *channel mapping* issues. In protocol refinement, we show how to refine communication towards application requirements such as reliability and throughput. In channel mapping, we discuss channel-convergence and channel-merge to make efficient use of shared network resources.

*Author's contributions*: The author developed the idea, implemented the proposed techniques, conducted experiments, and wrote the manuscript.

The remainder of the thesis is structured as follows. Chapter 2 summarizes our research results on NoC network architectures. In Chapter 3, we describe our work on NoC network performance analysis. We present our NoC communication refinement approach in Chapter 4. Finally we summarize the thesis in Chapter 5.

# Chapter 2

# NoC Network Architectures

*This chapter summarizes our research on NoC network architectures, particularly, cost-effective switch architectures [Paper 1, 2], connection-oriented multicasting [Paper 3], as well as TDM (Time Division Multiplexing) virtual-circuit configuration [Paper 4].*

## 2.1 Introduction

### 2.1.1 On-Chip Communication Network

#### A. On-chip network characteristics

As with macro- and tele-networks, on-chip micro-networks share the same characteristics in *topology*, *switching*, *routing*, and *flow control*. Additionally, a micro-network has to provide high and predictable performance with small area overhead and low power consumption. As noted in [27], a micro-network should appear as logical wires for network clients. Quality of Service (QoS) is thus a crucial aspect to distinguish one micro-network from another. Moreover, the design of on-chip systems should take advantage of well-validated legacy or third-party IP cores to shorten time-to-market and to guarantee product quality. To this end, IP reuse, exchange and integration are other critical issues. Addressing these issues demands a standardized hardware interface. The interface wrapping a micro-network can therefore be a distinguishing feature of a NoC proposal.

In the following, we describe the micro-network characteristics, namely, *topology*, *switching*, *routing*, *flow control*, *Quality of Service* and *Interface*, highlighting present NoC practices in these regards.

### B. Topology

The *topology* refers to the physical structure of the network graph, i.e., how network nodes (switches or routers) are physically connected. It defines the connectivity (the routing possibility) between nodes, thus having a fundamental impact on the network performance as well as the switch structure, for example, the number of ports and port width. The tradeoff between generality and customization is an important issue when determining a network topology. The generality facilitates the re-usability and scalability of the communication platform. The customization is aimed for performance and resource optimality. Both regular and irregular topologies have been advocated for NoCs. Regular topologies such as k-ary 2-cube meshes [56] and tori [27] are popular ones because their layouts on a two-dimensional chip plane use symmetric-length of wires. The significance of the regularity lies in its potential of managing wire delay and wire-related DSM effects. The k-ary tree and k-ary n-dimensional fat tree [1] are two alternative regular NoC topologies. With a regular topology, the network area and power consumption scale predictably with the size of the topology. The arguments for using irregular topologies are that specific applications require flexible and optimal topology. In [123], the number of ports in switches can be synthesized according to the requirement of connectivity. However, the area and power consumption of an irregular network topology may not scale predictably with the topology size. Other topologies in between regular and irregular ones are also proposed for NoCs. For example, an interesting NoC topology is the Octagon NoC [52] in which a ring of 8 nodes connected by 12 bi-directional chords. Traveling between any pair of nodes takes at maximum two hops. In [99], a butterfly fat-tree topology was proposed in which IPs are placed at the leaves and switches placed at the vertexes. Moreover, regular topology may be customized by introducing application-specific long-range links to improve performance with a small area penalty [96].

### C. Switching strategy

The *switching strategy* determines how a message traverses its route. There are two main switching strategies: *circuit switching* and *packet switching*. Circuit switching reserves a dedicated end-to-end path from the source to the destination before starting to transmit the data. The path can be a real or virtual circuit. After the transmission is done, the path reservation with associated resources is released. Circuit-switching is *connection-oriented*, meaning that there is an explicit connection establishment. In contrast to circuit-switching, packet-switching segments the message into a sequence of packets. A packet typically consists of a

header, payload and a tail. The header carries the routing and sequencing information. The payload is the actual data to be transmitted. The tail is the end of the packet and usually contains error-checking code. Packet-switching can be either *connection-oriented* or *connection-less*. Connection-oriented communication preserves resources while connection-less communication does not. Connection-oriented communication can typically provide a certain degree of commitment for message delivery bounds. With connection-less communication, packets are routed individually in the network in a best-effort manner. The message delivery is subject to dynamic contention scenarios in the network, thus is difficult to provide bounds. However, the network resources can be better utilized. Typical packet switching techniques[1] include *store-and-forward*, *virtual cut-through* [53], and *wormhole switching* [2].

- *Store-and-forward*: A network node must receive an entire packet before forwarding it to the next downstream node. Both link bandwidth and buffers are allocated at the packet-level. The non-contentional latency $T$ for transmitting $L$ flits is expressed by Equation 2.1. Flit is the smallest unit for the link-level flow control, which is the minimum unit of information that can be transferred across a link.

$$T = (L/BW + R) * H \tag{2.1}$$

  where $BW$ is the link bandwidth in flits per cycle; $R$ is the routing delay per hop; Hop is the basic communication action from switch to switch. $H$ is the number of hops from the source node to the destination node.

- *Virtual cut-through*: Like store-and-forward, virtual cut-through allocates both link bandwidth and buffers in units of packets. However, in virtual cut-through, a network node does not wait for the reception of an entire packet. It receives a portion of the packet, and then forwards it downstream if the buffer space in the next switch is available. The downstream node must have enough buffers to hold the entire packet. In case of blocking, the entire packet is shunt into the buffers allocated. By transmitting packets as soon as possible, virtual cut-through reduces the non-contentional latency $T$ for transmitting $L$ flits to

$$T = L/BW + R * H \tag{2.2}$$

[1] Both store-and-forward and virtual cut-through do not divide packets into flits. We show the division here for a consistent presentation of the switching techniques.

[2] In the literature, *wormhole switching*, *wormhole routing* and *wormhole flow control* have been used. In this thesis, we tend to use *wormhole switching*.

- *Wormhole switching*: A packet is decomposed into flits. Operating like virtual cut-through, wormhole switching delivers flits in a pipelined fashion. Due to the pipelined transmission, the non-contentional latency $T$ of transmitting $L$ flits is the same as that for virtual cut-through. Wormhole-switching and virtual cut-through are both *cut-through* switching techniques. They mainly differ in how they handle packet blocking. With wormhole switching, link bandwidth and buffers are allocated to flits rather than packets. The switch buffering capacity is a multiple of a flit. If a packet is blocked, flits of the packet are *stalled* in place. With virtual cut-through, a switch, at which a packet is blocked, must *receive* and *store* all flits of the blocked packet. This enforces that the buffering capacity in switches must be a multiple of a packet. Virtual cut-through utilizes the network's bandwidth more efficiently, achieving higher throughput than wormhole switching but requiring higher buffering capacity.

Circuit-switching for on-chip networks is proposed in [132] to satisfy applications with hard real-time constraints. The majority of on-chip networks is based on packet-switching, and combined packet-switching and circuit-switching. For example, TDM virtual-circuits [34, 81], which preserves time slots to switch packets in a contention-free manner, can be viewed as a circuit-switching technique implemented in a packet-switched network.

### D. Routing algorithm

The *routing algorithm* determines the routing paths the packets may follow through the network graph. It usually restricts the set of possible paths to a smaller set of valid paths. In terms of path diversity and adaptivity, routing algorithm can be classified into three categories, namely, *deterministic routing*, *oblivious routing* and *adaptive routing* [28]. Deterministic routing chooses always the same path given the source node and the destination node. It ignores the network path diversity and is not sensitive to the network state. This may cause load imbalances in the network but it is simple and inexpensive to implement. Besides, it is often a simple way to provide the ordering of packets. Oblivious routing, which includes deterministic algorithms as a subset, considers all possible multiple paths from the source node to the destination node, for example, a random algorithm that uniformly distributes traffic across all of the paths. But oblivious algorithms do not take the network state into account when making the routing decisions. The third category is adaptive routing, which distributes traffic dynamically in response to the network state. The network state may include the status of a node or link, the length of queues,

and historical network load information. A routing algorithm is termed *minimal* if it only routes packets along shortest paths to their destinations, i.e., every hop must reduce the distance to the destination. Otherwise, it is non-minimal. Both *table-based* and *algorithmic* routing mechanics can be used to realize the routing algorithms [28]. The table-based routing mechanism uses routing tables either at the source or at each hop along the route. Instead of storing the routing relation in a table, the algorithmic routing mechanism computes it. For speed, it is usually implemented as a combinational logic circuit. The algorithmic routing is usually restricted to simple routing algorithms and regular topologies, sacrificing the generality of table-based routing.

In comparison with adaptive routing, deterministic or oblivious minimal routing results in relatively simple switch designs because a routing decision is made independent of the dynamic network state. Though a routing algorithm has different properties in design complexity, adaptivity and load balancing, the performance of a routing algorithm is also topology and application dependent [88]. An interesting extreme case of non-minimal adaptive routing is *deflection routing* [16], also called *hot-potato* routing. Its distinguishing feature is that it does not buffer packets. Instead, packets are always on the run cycle-by-cycle. A deflection policy prioritizes packets on the use of favored links. If there is no contention, packets are delivered via shortest paths. Upon contending for shared links, packets with a higher priority win arbitration and use the favored links while packets with a lower priority are mis-routed to non-minimal routes. Deflection routing has been used in optical networks where buffering optical signals is too expensive [106]. Because of simplicity and adaptivity, it is adopted and implemented in communication networks embedded in massively parallel machines such as the Connection machine [42]. For the same reasons, it has also been proposed for on-chip networks in the Nostrum NoC [81, 91]. Using deflection routing results in faster and smaller switch designs. As projected in [91], a deflection switch with an arity of five can run 2.38 GHz with a gate count of 19370 in 65 $nm$ technology. Deadlock and livelock are the primary concern when designing a routing algorithm in order to ensure correct network operation [30]. As shown in [97], application knowledge can be effectively utilized to avoid deadlock. In [16, 49], maximum delivery bounds are derived for deflection networks. Thus the networks are livelock free.

### E. Network flow control

The *network flow control* governs how packets are forwarded in the network, concerning shared resource allocation and contention resolution. The shared resources are buffers and links (physical channels). Essentially a flow control mechanism

deals with the coordination of sending and receiving packets for the correct delivery of packets. Due to limited buffers and link bandwidth, packets may be blocked due to contention. Whenever two or more packets attempt to use the same network resource (e.g., a link or buffer) at the same time, one of the packets could be stalled in placed, shunted into buffers, detoured to an unfavored link, or simply dropped. For packet-switched networks, there exist *bufferless flow control* and *buffered flow control* [28].

- *Bufferless flow control* is the simplest form of flow control. Since there is no buffering in switches, the resource to be allocated is link bandwidth. It relies on an arbitration to resolve contentions between contending packets. After the arbitration, the winning packet advances over the link. The other packets are either dropped or misrouted since there are no buffers. The deflection routing uses bufferless flow control. In fact, deflection routing includes an orthogonal concern of routing algorithm and deflection policy. While a routing algorithm determines the favored links for packets, a deflection policy resolves contentions for shared links by forwarding the packet with the highest priority to its favored link and misrouting other packet(s) with a lower priority to unfavored links. As deflection routing does not buffer packets, the switch design can be simpler and thus cheaper because it has no buffer and flow management. Moreover, since the routing paths of packets are fully adaptive to the network state, deflection routing has higher link utilization and offers the potential to allow resilience for link and switch faults.

- *Buffered flow control* stores blocked packets while they wait to acquire network resources. Store-and-forward, virtual cut-through and wormhole switching techniques adopt buffered flow control. The granularity of resource allocation for different buffered flow control techniques may be different. Store-and-forward switching and virtual cut-through switching allocate link bandwidth and buffers in units of packets. Wormhole switching allocates both link bandwidth and buffers in units of flits. Buffered flow control requires a means to communicate the availability of buffers at the downstream switches. The upstream switches can then determine when a buffer is available to hold the next flit to be transmitted. If all of the downstream buffers are full, the upstream switches must be informed to stop transmitting (assuming drop-less delivery). This phenomenon is called *back pressure*. Link-level flow control mechanisms, in which the buffer availability information is passed and propagated between switches, are introduced to provide such

back-pressure. Today, there are three types of link-level flow control techniques in common use: credit-based, on/off, and ack/nack [28].

The flow control scheme of a network may be coupled with its switching strategy. For instance, both store-and-forward and virtual cut-through switching use the packet-buffer flow control, and wormhole switching uses the flit-buffer flow control. It is worthwhile to discuss them separately because a flow control scheme emphasizes the movement of packet flows instead of switching individual packets.

### F. Quality of Service

Generally speaking, Quality-of-Service (QoS) defines the level of commitment for packet delivery. Such a commitment can be correctness of the result, completion of the transaction, and bounds on the performance [33]. But, mostly, QoS has a direct association with bounds in bandwidth, delay and jitter, since correctness and completion are often the basic requirements for on-chip message delivery. Correctness is concerned with packet integrity (corrupt-less) and packet ordering. It can be achieved through different means at different levels. For example, error-correction at the link layer or re-transmission at the upper layers can be used to ensure packet integrity. A network-layer service may secure that the packets are delivered in order. Alternatively, if a network-layer service cannot promise in-order delivery, a transport-layer service may compensate to do the re-ordering. Completion requires that a flow control method does not drop packets. In case of a shortage of resources, packets can be mis-routed or buffered. In addition, the network must ensure deadlock and livelock freedom.

Roughly classified, NoC researchers have proposed *best-effort*, *guaranteed*, and *differentiated* services for on-chip packet-switched communication. A best-effort service is connectionless. The network delivers packets as fast as it can. Packets are routed in the network, resulting in dynamic contentions for shared buffers and links. A packet-admission policy is usually desired to avoid network saturation. Below the saturation point, the network exhibits good average performance but the worst-case can be more than an order of magnitude worse than the average case. A guaranteed service is typically connection-oriented. It avoids network contentions by establishing a virtual circuit. Such a virtual circuit may be implemented by time slots, virtual channels, parallel switch fabrics and so on. The Æthereal NoC [34] implements a contention-free TDM virtual-circuit in a network employing buffered flow control. The Nostrum NoC [81] also realizes TDM virtual-circuit but in a network using bufferless flow control. Both Æthereal and Nostrum networks operate synchronously. The MANGO network [14] is clockless.

Since the network switches do not share the same notion of time, it uses sequences of virtual channels to set up virtual end-to-end connections. In contrast to TDM, SDM (Space-Division-Multiplexing)-based QoS is achieved by allocating individual wires on the link for different connections [61]. For the guaranteed services, if a virtual circuit is set up dynamically, the setup procedure has to use best-effort packets. This phase is somewhat unpredictable due to the best-effort nature. A differentiated service prioritizes traffic according to different categories, and the network switches employ priority-based scheduling and allocation policies. For instance, the QNoC [15] network distinguishes four traffic classes, i.e., signaling, real-time traffic, read-write and block transfer. The signaling class has the highest and the block-transfer class the lowest priority. Priority-based approaches allow for higher utilization of resources but cannot provide strong guarantees like guaranteed services. To improve resource usage, a best-effort service may be mixed with a guaranteed service using, for example, slack-time aware routing [5].

### G. Interface

Wrapping on-chip networks with an interface is essential for NoC designs. The interface is preferably standardized, but domain-specific customization is necessary for optimal and dedicated solutions. An interface-based design approach [112, 129] separates computation from communication. It gives the interface users an abstraction that makes only the relevant information visible. It facilitates the exchange and reuse of IPs as long as the IPs conform to the same interface.

To make a huge number of legacy IPs reusable and integrable, an on-chip network interface has to follow standard interfaces. Current network interconnects implement interfaces such as AXI [6], OCP [95], VCI [130] and DTL [104]. AXI (Advanced eXtensible Interface) is AMBA's highest performance interface developed by ARM to support ARM11 processors. The configurable AXI interconnection is optimized for the processor-memory backplane and has advanced features such as split transactions (address and data buses are decoupled), multiple outstanding transactions, and out-of-order data. OCP (Open Core Protocol) is a plug-and-play interface for a core having both master and slave interfaces. The OCP specification defines a flexible family of memory-mapped, core-centric protocols for use as a native core interface in on-chip systems. OCP addresses both data-flow signaling and side-band control-flow signaling for error, interrupt, flag, status and test. The VCI (Virtual Component Interface) specification includes three variants: PVCI (peripheral), BVCI (basic) and AVCI (advanced). The DTL (Device Transaction Level) interconnection interface is developed by Phillips Semiconductors to

interface existing SoC IPs. It allows easy extension to other future interconnection standards. The Æthereal NoC [113] provides a shared-memory abstraction to the cores and is compatible to standard interfaces such as AXI, DTL and OCP. The SPIN [37] and Proteo [122] NoCs support the VCI interface. The OCP interface is used in the MANGO NoC [14]. Nonetheless, the cost of adopting standard socket-type interfaces is nontrivial. The HERMES NoC [84] demonstrates that the introduction of OCP makes the transactions up to 50% slower than the native core interface. Therefore domain-specific interfaces will be an option for optimization.

Next, in Section 2.2, we investigate the design complexity of a canonical wormhole switch from the perspective of admitting and ejecting flits, proposing the *coupled admission* model for flit admission (Paper 1) and the $p$*-sink* model for flit ejection (Paper 2). Section 2.3 suggests a multicasting service (Paper 3). In Section 2.4, we discuss the construction of TDM virtual-circuits using logical networks (Paper 4). Since both Section 2.2 and Section 2.3 consider wormhole switching, we introduce wormhole switching further in the next subsection.

### 2.1.2 Wormhole Switching

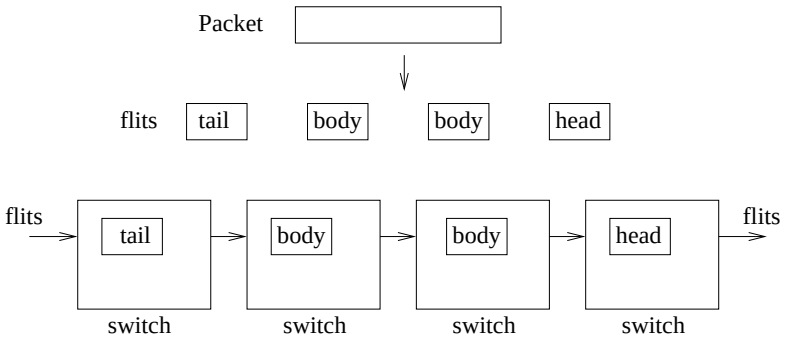



**Figure 2.1.** Flits delivered in a pipeline

Wormhole switching [26] allocates buffers and physical channels (PCs, links) to flits instead of packets. A packet is decomposed into a head flit, body flit(s), and a tail flit. A single-packet flit is also possible. We call this decomposition *flitization*. Flitization is named following packetization, i.e., encapsulate a message into one or more packets. A flit, the smallest unit on which flow control is performed, can advance once buffering in the next switch is available to hold the flit. This results in that the flits of a packet are delivered in a pipeline fashion. As illustrated in Figure

2.1, a packet is segmented into four flits, with one head flit leading two body flits and one tail flit, and then the four flits are transmitted in a pipeline via switches. For the same amount of storage, it achieves lower latency and greater throughput.

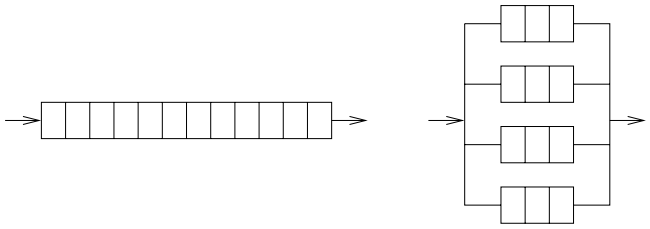



**Figure 2.2.** Virtual channels (lanes)

However, wormhole switching uses physical channels (PCs) inefficiently because a PC is held for the duration of a packet. If a packet is blocked, all PCs held by this packet are left idle. To mitigate this problem, wormhole switching adopts *virtual channels* (*lanes*) to make efficient use of the PCs [25]. Several parallel lanes, each of which is a flit buffer queue, share a PC (Figure 2.2). Therefore, if a packet is blocked, other packets can still traverse the PC via other lanes, leading to higher throughput. Because of these advantages, namely, *better performance*, *smaller buffering requirement* and *greater throughput*, wormhole switching with virtual-channel flow control is the prevailing switching scheme advocated for on-chip networks [1, 24, 44, 110]. In addition, virtual-channel has a versatile use in optimizing link utilization, improving throughput, avoiding deadlock [26], increasing fault tolerance [18] and providing guaranteed services [14]. Nonetheless, in order to maximize its utilization, the procedure to allocate virtual channels is critical in designing routing algorithms [128].

Note that wormhole switching is not without problems. First, it incurs flit-type overhead to distinguish head, body, tail, and single-packet flits. Second, the flits of a packet may be distributed in flit buffers of multiple switches. The intermediate buffers between a head flit and a tail flit may be under-utilized, resulting in lower buffer utilization [120]. Third, due to the flit distribution, wormhole switching is more prone to deadlock.

## 2.2 Flit Admission and Ejection

This section summarizes the research in Paper 1 (Flit admission) and Paper 2 (Flit ejection).

### 2.2.1 Problem Description

Despite the aforementioned advantages, using wormhole switching for on-chip networks has to minimize the switch design complexity. First, since an on-chip network is an interconnect using shared wires instead of dedicated wires to pass signals, its cost must be reasonable [27]. Second, embedded applications often have very stringent requirements on power. For future complex SoC integration, the communication bandwidth is achievable but the energy consumption will probably be the bottleneck that has to trade off the performance [98]. Therefore, in order to shrink energy dissipation, it is important to reduce the switch design complexity so as to decrease the number of gates and switching capacitance.

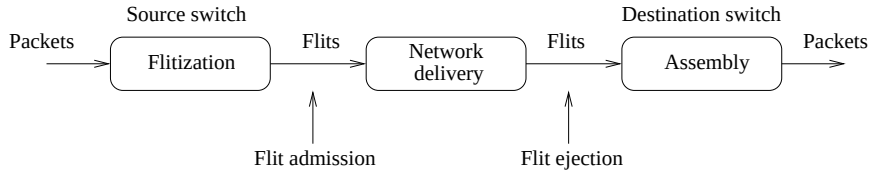



**Figure 2.3.** Flit admission and ejection

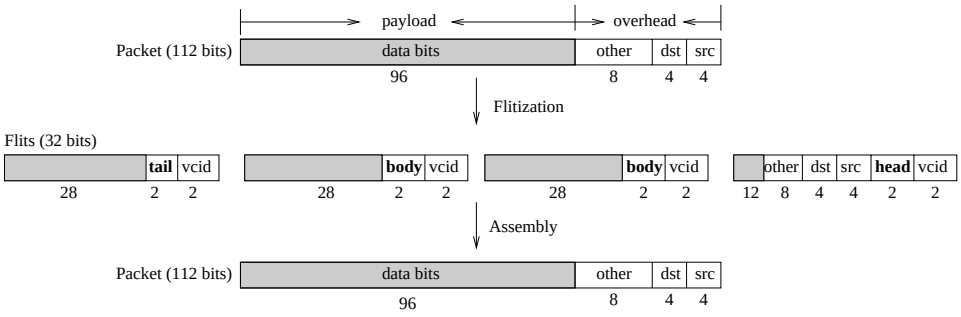



**Figure 2.4.** Flitization and assembly

We examine the problem of flit-admission and flit-ejection in a wormhole-switched network. As depicted in Figure 2.3, the delivery of packets passes through three stages: *flitization*, *network delivery*, and *assembly*. The flitization is performed at a source node, and the assembly, which decapsulates flits into packets, is conducted by a destination node. Figure 2.4 illustrates the flitization and assembly of a packet. As can be seen, the packet is encapsulated into four flits (one head flit, two body flits and one tail flit), where vcid is the identity number of the lane the flit occupies. We assume 4 lanes per port in a switch, thus vcid takes 2 bits.

Since flits are both the workload of switches and the source of network contentions for shared Virtual Channels (VCs)[3] and Physical Channels (PCs, links), their admission and ejection are as important as delivery. As the transmission time of a flit comprises admission time, delivery time plus ejection time, the network performance is the function of *flit-admission*, *flit-delivery* and *flit-ejection*. Intuitively, to achieve good network utilization and throughput, flits should be admitted as fast as possible. However, flits to be advanced (after admission) may contend not only with each other, but also with flits to be admitted. Flit-admission and flit-delivery interfere with each other. This implies that a fast admission mechanism may speed up the admission but slow down the delivery. If the network is too loaded, the overall transmission time may get worse. For the ejection process, a faster ejection frees flit buffers quicker, thus the faster the better. A slower ejection of flits may slow down the flit delivery and eventually affect the flit delivery and admission through back-pressure. However, an ideal ejection, which ejects flits immediately once they reach destinations, may over-design the switch. Finally the interplay between flit-admission and flit-ejection influences the tradeoff between performance and switch complexity. A practical cost-effective ejection model may actually tolerate a slower but simpler admission model with reasonable performance penalty.

In the rest of this section, we first explain the operation of a canonical wormhole lane switch in Section 2.2.2. Then we discuss flit-admission and flit-ejection models in Section 2.2.3 and Section 2.2.4, respectively. Particularly, we introduce the *coupled* admission and the $p$*-sink* model.

### 2.2.2 The Wormhole Switch Architecture

Figure 2.5 illustrates a canonical wormhole switch architecture with virtual channels at input ports [25, 102, 110]. It has $p$ physical channels (PCs) and $v$ lanes per PC. It employs the credit-based link-level flow control to coordinate packet delivery between switches.

A packet passes the switch through four states: *routing*, *lane allocation*, *flit scheduling*, and *switch arbitration*. In the routing state, the routing logic determines the routing path the packet advances over. Routing is only performed with the head flit of a packet and only when the head flit becomes the earliest-come flit in the FIFO lane. After routing, the packet path and output PC are determined. In the state of lane allocation, the lane allocator *associates* the lane the packet occupies with an available lane in the next switch on its routing path, i.e., to make a

[3] In Section 2.2 and Section 2.3, we use the shorthand *VC* for *Virtual Channel*.

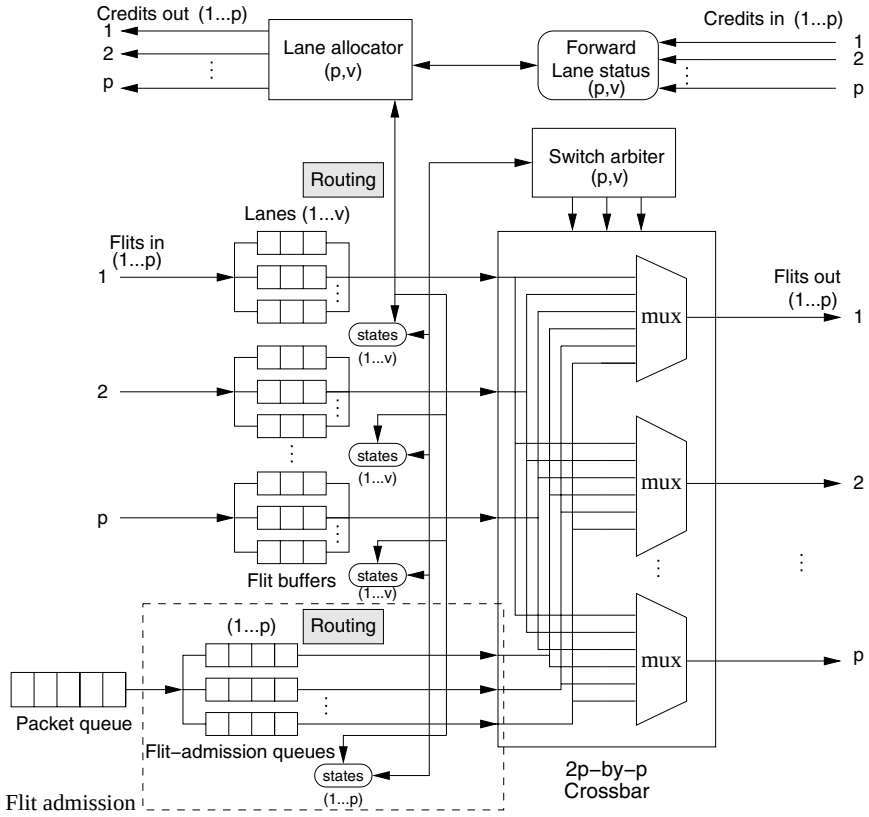



**Figure 2.5.** A canonical wormhole lane switch (ejection not shown)

*lane-to-lane* association. A lane is available if it is not currently being allocated to an upstream lane. A lane-to-lane association fails when all requested lanes are already associated to other lanes in directly connected switches, or the lane loses arbitration in case multiple lanes in the switch try to associate with the same downstream lane. Note that it is not necessarily required here that there is an empty buffer in the lane in order to make a successful association. If the lane-to-lane association succeeds, the flit vcid is determined and the packet enters the scheduling state. If there is a buffer available in the associated downstream lane, the lane enters the state of switch arbitration. This can be done with a two-level arbitration scheme. The first level of arbitration is performed on the lanes sharing the same physical channel. The second level of arbitration is for the crossbar traversal. If the lane wins the two levels of arbitration, the flit situated at the head of the lane is switched out. Otherwise, the lane stays in the arbitration state. The lane-to-lane

association is released after the tail flit is switched out. Then the allocated lane is available to be reused by other packets. Credits are passed between adjacent switches in order to keep track of the status of lanes, such as if a lane is free and a count of available buffers in the lane.

A flit differs from a packet in that (1) a flit has a smaller size; (2) only the head flit carries the routing information such as source/destination address, packet size, priority etc. As a consequence, the routing and lane allocation can only be performed with the head flit of a packet. Once a lane-to-lane association is established by the head flit of the packet, the rest of flits of the packet inherit this association. After the tail flit leaves, the lane-to-lane association is released. Thus, a lane is allocated at the packet level, i.e., *packet-by-packet* while a link is scheduled at the flit level, i.e., *flit-by-flit* since the flit scheduling as well as the switch arbitration is performed per flit. As the head flit advances, lanes are associated like a chain along the routing path of the packet, the rest of flits are pipelined along the chain path. Carrying routing information only in the head flit of a packet leaves more space for payload. However, flits belonging to *different* packets can not be interleaved in associated lane(s) since only head flits contain routing information. To guarantee this, a lane-to-lane association must be *one-to-one*, i.e., *unique* at a time.

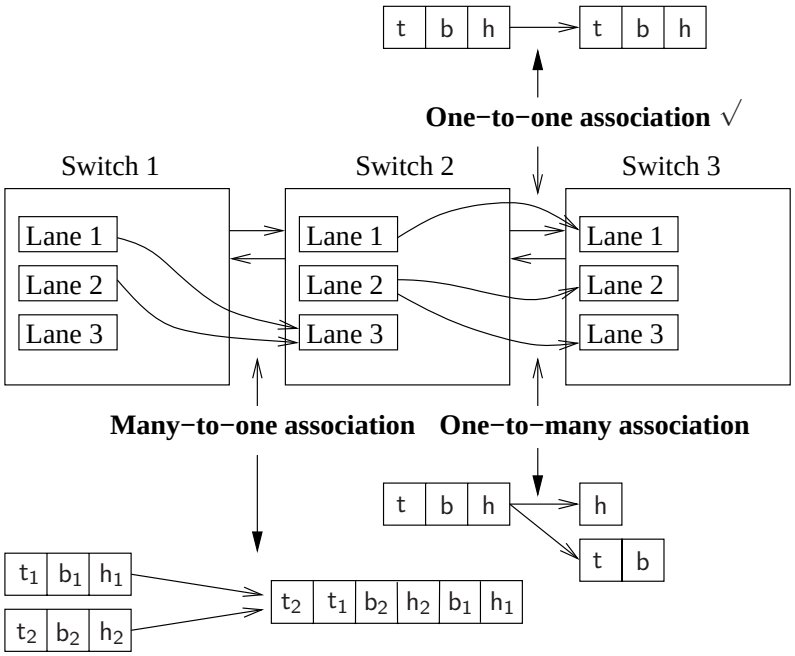



**Figure 2.6.** Lane-to-lane associations

Figure 2.6 illustrates lane-to-lane associations. The one-to-many association leads to that the flits from lane 2 in switch 2 are delivered to lane 2 and lane 3 in switch 3. The many-to-one association results in that lane 3 in switch 2 will receive flits from lane 1 and lane 2 from switch 1. Obviously the one-to-many and many-to-one associations result in that the integrity of a worm (the flit sequence of a packet) is destroyed. It becomes impossible either to route the flits of a packet or assemble the flits into a packet. Therefore, both one-to-many and many-to-one associations must be forbidden, and only one-to-one association is permissible.

### 2.2.3 Flit Admission

#### A. The decoupled admission

We assume that a switch receives packets injected via a packet FIFO. A packet is first flitized into flits that are then stored in flit FIFOs, called *flit-admission queues*, before being admitted into the network. There are various ways of organizing the packet queue and the flit-admission queues. In Figure 2.7(a), flit-admission queues are organized as a FIFO. In Figures 2.7(b) and 2.7(c), they are arranged as $p$ parallel FIFO queues ($p$ is the number of PCs). Figures 2.7(a) and 2.7(b) permit at maximum one flit to be admitted to the network at a time while Figure 2.7(c) allows up to $p$ flits to be admitted simultaneously. We adopt the organization of flit-admission queues in Figure 2.7(c) for our further discussions since it allows potentially higher performance while the other two may lead to under-utilize the network.

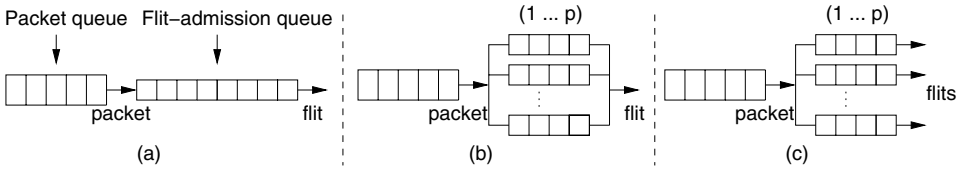



**Figure 2.7.** Organization of packet- and flit-admission queues

The organization of $p$ flit-admission queues is also illustrated in the switch architecture in Figure 2.5. Initially, packets are stored in the packet queue. When a flit-admission queue is available, a packet is split into flits which are then put into an admission queue. Similarly to a lane, a flit-admission queue transits states to inject flits into the network via the crossbar. Note that flits to be admitted (in admission) contend with flits already admitted (in delivery) for VCs in the lane-to-lane association state and PCs in the *crossbar arbitration* state. This interference

makes a flit-admission model nontrivial. Our study shows that when the network is nearly saturated, a faster admission model actually begins to slow down the delivery, worsening the network performance. Furthermore, *the routing is performed after flitization*. By this scheme, each flit-admission queue is connected to $p$ multiplexers. Flits from a flit-admission queue can be switched to anyone of the $p$ output PCs. To implement this scheme, the crossbar must be fully connected, resulting in a port size of $2p \times p$. Since the flit-admission queues are decoupled from the output PCs, we call this flit-admission scheme *decoupled admission*.

### B. The coupled admission

Although the decoupled admission allows a flit to be switched to anyone of the $p$ output ports, this may not be necessary since a flit is aimed to one and only one port after routing. Based on this observation, we propose a coupling scheme that can sharply decrease the crossbar complexity, as sketched in Figure 2.8. Just like the decoupled admission, it uses $p$ flit-admission queues, but one queue is bound to one and only one multiplexer dedicated for a particular output PC. Due to this coupling, flits from a flit-admission queue are dedicated to the output PC. Consequently, an admission queue only needs to be connected to one multiplexer instead of $p$ multiplexers. The size of the crossbar is sharply decreased from $2p \times p$ to $(p+1) \times p$, as shown in Figure 2.8. The number of control signals per multiplexer is reduced from $\lceil log(2p) \rceil$ to $\lceil log(p+1) \rceil$ for any $p > 1$ [4].

In order to support the coupling scheme, *the routing must be performed before flitization* instead. By a routing algorithm, the output physical channel that a packet requests can be determined. Hence, the corresponding admission queue is identified. One drawback due to the coupling is that the head-of-line blocking may be worse if the packet injection rate is high. Specifically, if the head packet in the packet queue is blocked due to the bounded number and size of the flit-admission queues, the packets behind the head packet are all unconditionally blocked during the head packet's blocking time. In the decoupled admission, the head-of-line blocking occurs when the four flit-admission queues are fully occupied. With the coupled admission, this blocking occurs when the flit-admission queue, which the present packet targets, is full.

As the crossbar is a power-hungry component in a switch [131], the coupled admission saves also power in comparison with the decoupled admission due to the reduction in the gate count and switching capacitance. The study on the power assumption of the flit admission schemes [74] shows that the coupled admission

[4] $\lceil x \rceil$ is the ceiling function which returns the least integer that is not less than $x$.

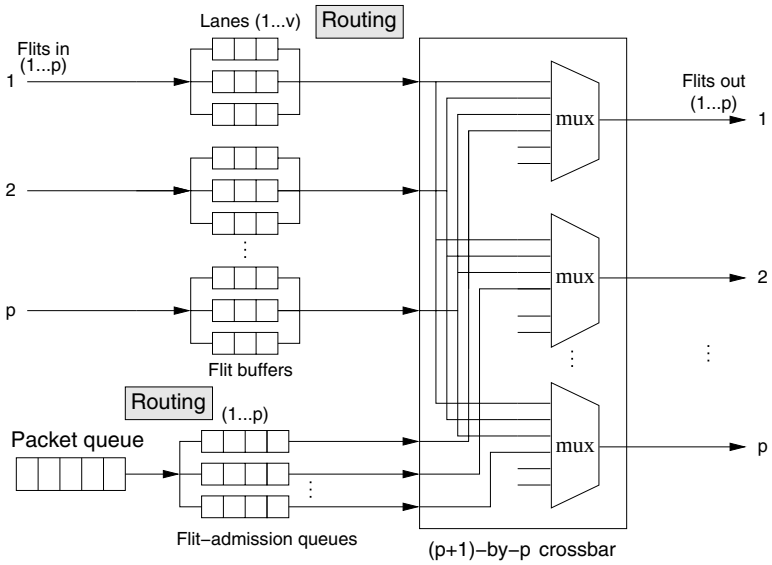



**Figure 2.8.** The coupled admission sharing a (p+1)-by-p crossbar

decreases the switch power by about 12% on average with the uniform traffic with random-bit payloads.

### 2.2.4 Flit Ejection

#### A. The ideal sink model

An ideal sink model is typically assumed for a wormhole lane switch. With such a model, flits reaching their destinations are ejected form the network immediately, emptying the lane buffers they occupy. An ideal flit-ejection model is drawn in Figure 2.9. A *flit sink* is a FIFO receiving the ejected flits. Each lane is connected to a sink and the crossbar (for packet forwarding) via a de-multiplexer.

To incorporate ejection, the lane state is extended with a *reception* state in addition to the four states. If the routing determines that the head flit of a packet reaches its destination, the lane enters the reception state immediately by establishing a *lane-to-sink* association. Since flits from different packets can not interleave in a sink queue, there must be $p \cdot v$ sink queues, each of them corresponding to a lane, in order to realize an immediate transition to the reception state. Assuming that one sink takes the flits of one packet, the depth of a sink is the maximum number of flits of a packet. After the lane transits to the reception state, the head flit bypasses the crossbar and enters its sink. The subsequent flits of the packet are

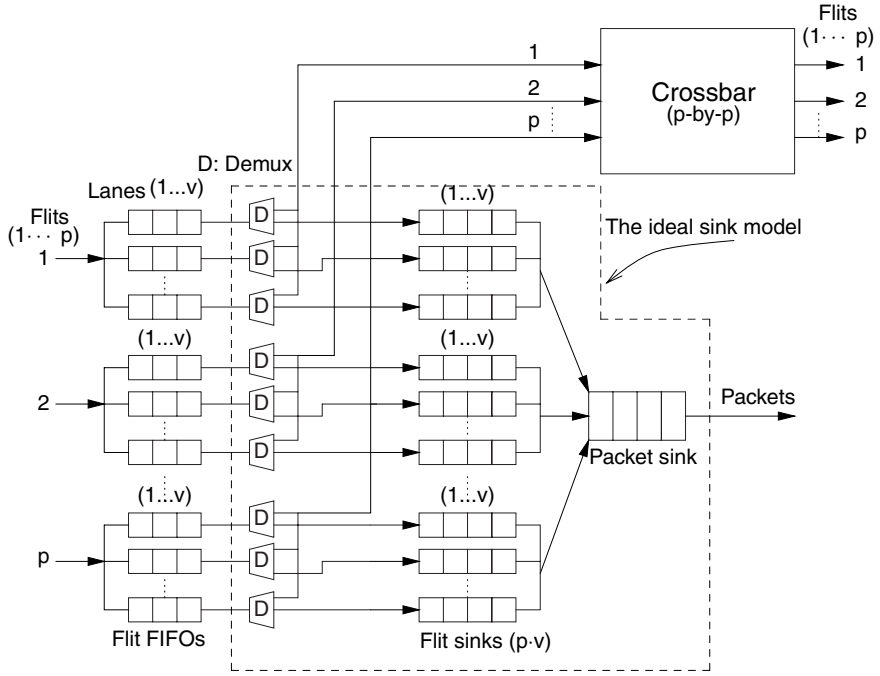



**Figure 2.9.** The ideal sink model

ejected into the sink immediately upon arriving at the switch. When the tail flit is ejected, the lane-to-sink association is freed. This model is beneficial in both time and space. Although a head flit may be blocked by flits situated in front of it in the same lane, a non-head flit neither waits to be ejected (time) nor occupies a flit buffer (space) once the lane is in the reception state. Moreover, it does not interfere with flits buffered in other lanes from advancing to next switches downstream (because the $v$ demultiplexers of a PC share one input of the crossbar, one PC allows one flit from a lane to be switched via the crossbar without interference with sinking flits from other lanes.). Upon receiving all the flits of a packet, the packet is composed and delivered into the packet sink. If the packet sink is not empty, the switch outputs one packet per cycle from it in a FIFO manner.

### B. The $p$-sink model

Implementing the ideal sink model requires $p \cdot v$ flit sinks, which can eject $p \cdot v$ flits per cycle. This may over-design the switch since there are only $p$ input ports, implying that at maximum $p$ flits can reach the switch per cycle. Since the

maximum number of flits to be ejected per switch per cycle is $p$, we can use $p$ sink queues instead of $p \cdot v$ sink queues to eject flits to avoid over-design. Moreover, in order to have a more structured design, we could connect the $p$ sink queues to the crossbar, as illustrated in the dashed box of Figure 2.10.

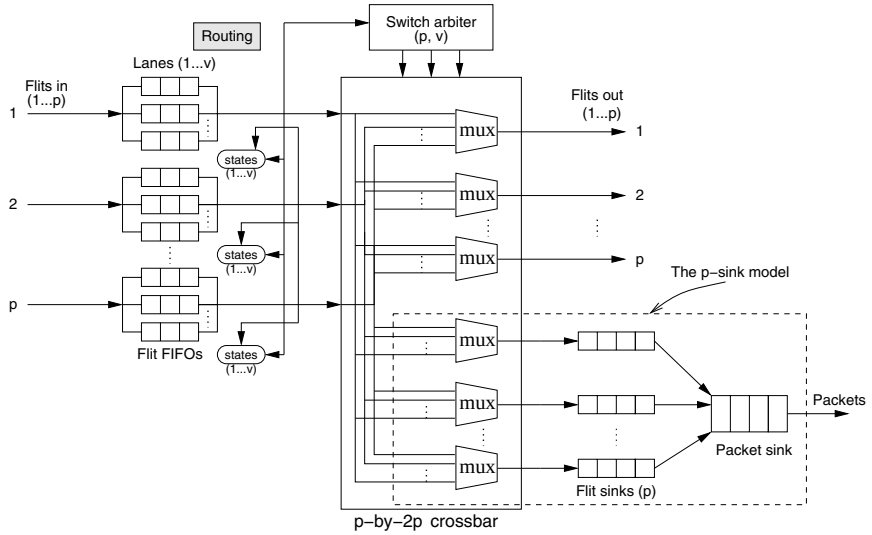



**Figure 2.10.** The $p$-sink model

To enable ejecting flits in the $p$-sink model, we now extend the four lane states with two new states: an *arriving* and a *reception* state. If a head flit reaches its destination, the lane the flit occupies transits from the routing to the arriving state. Then it will try to associate with an available sink, i.e., to establish a *lane-to-sink* association. If the association is successful, the lane enters the *reception* state. Subsequently the other flits of the packet follow this association exactly like flits advancing in the network. Upon the tail flit entering the sink, the association is torn down. If the lane-to-sink association fails (when all sinks have already been allocated), the head flit is blocked in place holding the lane buffer. To speed up flit ejection, the contentions for the crossbar input channels and crossbar traversal are arbitrated on priority. A lane in a reception state has a higher priority than a lane in a state for forwarding flits. The drawback in this sink model is the increase of blocking time when flits reach their destinations. First, the lane-to-sink association may fail since all sink queues might be in use. In contrast, an ideal sink model guarantees an exclusive sink for each lane. Second, only one lane per PC can win arbitration to an input channel of the crossbar due to sharing the input channel for

both advancing flits and ejecting flits. In case of more than one lane of a PC are in the reception state, only one lane can use the channel.

| | $p \times 1$ **Mux** | $1 \times 2$ **Demux** | **Flit sink** |
|---|---|---|---|
| **The ideal model** | - | $p \cdot v$ | $p \cdot v$ |
| **The $p$-sink model** | $p$ | - | $p$ |

**Table 2.1.** Cost of the sink models

To implement this $p$-sink model, the crossbar must double its capacity from $p$-*by*-$p$ ($p$ $p \times 1$ multiplexers) to $p$-*by*-$2p$ ($2p$ $p \times 1$ multiplexers), as illustrated in Figure 2.10. The number of control ports of the crossbar is doubled proportionally. Table 2.1 summarizes the number of each component to implement the sink models. As can be seen, the ideal sink model requires $p \cdot v$ flit sinks while the $p$-sink model uses only $p$ flit sinks. With the $p$-sink model, the number of flit-sinks becomes independent of $v$, implying that the buffering cost for flit sinks is only $1/v$ as much as the ideal ejection model.

## 2.3 Connection-oriented Multicasting

This section summarizes the research in Paper 3.

### 2.3.1 Problem Description

As discussed previously, a bus and its variants (segmented, cross-bar and hierarchical buses) do not scale well with the system size in bandwidth and clocking frequency. However, a bus is very efficient in broadcasting since all clients are directly connected to it. A unicast is in fact broadcasted to all clients in the bus segment. In a NoC, IP blocks are distributed and communicate through multi-hop connections. This allows many more concurrent transactions, but does not directly support multicast. In NoC systems, it often desires to maintain a consistent view on the system state among the distributed cores, for example, in the case of implementing cache coherency protocols, of passing global states for barrier synchronization, and of managing and configuring the network. These communication patterns involve one source but multiple recipients. This type of pattern distinguishes from one-to-one communication in that the same message from one source has to be transmitted to multiple destinations. Particularly, real-time constrained, throughput-oriented embedded applications for multi-media processing

exhibit such patterns, for instance, forking one data stream into multiple identical streams to be processed by multiple processing elements. Providing an efficient support for such one-to-many communication patterns is desirable.

Implementing multicast by sending multiple unicast messages is intuitive but neither efficient nor scalable because of excessive link and buffer consumption. Secondly, these messages are delivered in a best-effort manner without QoS. Our purpose is to provide an efficient multicast support from the network layer. We have taken a connection-oriented approach in aware of QoS. This allows dynamic multicast setup and release, thus consuming resources only if necessary. Our multicast scheme is realized in wormhole-switched networks. The resulting wormhole switch supports both unicast and multicast.

### 2.3.2 The Multicasting Mechanism

Our multicasting mechanism consists of three phases: *group setup*, *data transmission*, and *group release*. It is connection-oriented, meaning that a multicast connection must be established before one-to-many data transmission can start. The member visiting order of a multicast group is computed off-line and the multicast path is set up conforming to the unicast routing algorithm in the group setup phase. After data transmission, a multicast connection has to be explicitly released. A multicast connection means that

- There is a group master who owns the connection. The group master is the source node who has the group member information and determines the member visiting sequence. It initiates the establishment by sending a multicast setup packet using unicast. The last node in the sequence is responsible for acknowledging the establishment. In case of setup failure, a negative acknowledgment is sent from the node where the failure occurs. After data transmission, the group master sends a multicast release packet to release the connection.

- A simplex path is defined from the group master to the last member, passing other member nodes. Data transmission will deterministically follow this path from upstream to downstream. In addition to the group master, any upstream node is allowed to send multicast packets downstream.

- Each switch along the multicast delivery path has stored information about how to deal with a multicast packet (*copy*, *forward* or *sink*) and about the

connection status. The copy means that the multicast packet has to be forwarded besides being locally sunk. The record of a multicast connection includes {MultiID, GroupType, Sadr, VCID, VCID downstream, Output PC, Next member addr.}, where MultiID is the multicast group identity number, which is unique for each multicast group; GroupType is the type of the multicast group which informs the switches whether the multicast group requires reserving a lane or not; Sadr is the group master address; VCID is the identity number of the lane that the multicast packets use in the current switch; VCID downstream is the identity number of the lane allocated in the next downstream switch; OutputPC is the output physical channel over which the multicast packets are to be switched; Next member address is the address of the next member in the multicast group.

During the setup phase, multicasting can be aware of QoS in the sense that a multicast group may request to *reserve a lane*. The GroupType indicates if the group reserves a lane or not. To speed up multicasting, multicast packets enjoy higher priority than unicast packets for link bandwidth arbitration. After a connection is established, a multicast is realized by sending a single copy of multicast packets to multicast group members along the pre-established path. Multicast packets carry multiID in their headers. This results in low packet overhead and efficient use of link bandwidth. The drawback is the setup and release overhead.

The multicasting protocol is designed seamlessly with the unicasting protocol. The unicast packet format is extended to include different types of packets. In the implementation, the controller of the unicast switch is enriched to identify and act according to the different packet types. The data path of the switch remains the same. In this way, the resulting wormhole switch supports both unicast and multicast. The network allows mixed unicast and multicast traffic.

## 2.4 TDM Virtual-Circuit Configuration

This section summarizes the research in Paper 4.

### 2.4.1 Problem Description

A Virtual Circuit (VC)[5] is a set of pre-allocated resources to enable performance guarantees. Since the pre-allocation involves a setup phase, a VC is connection-oriented. A TDM (Time-Division Multiplexing) VC [34, 81] is a VC that shares

[5] In Sections 2.4 and 2.5, we use the shorthand *VC* for *Virtual Circuit*.

buffers and link bandwidth in a time-division fashion. Each node along the VC's path is equipped with a time-sliced routing table which reserves time slots for input packets to use output links. The routing table partitions link bandwidth and avoids the simultaneous use of shared links. A VC is simplex. In general, it may comprise multiple source and destination nodes (multi-node). As long as a VC is established, packets delivered on it, called *VC packets*, encounter no contention and thus have guarantees in latency and bandwidth. Unlike connection-less Best-Effort (BE) packet delivery that starts as soon as possible, VC packet delivery can not start until the VC is successfully set up. Therefore, VC configuration is an indispensable process. Moreover, well-planned VC configurations can make a better use of network resources and achieve better performance.

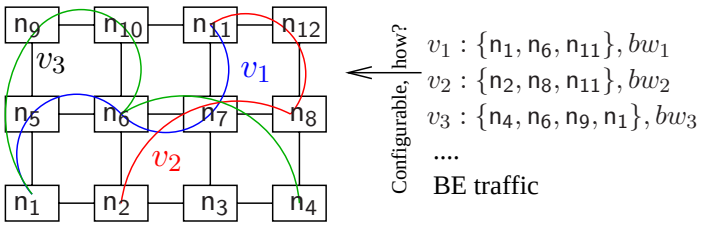



**Figure 2.11.** The virtual-circuit configuration problem

Figure 2.11 illustrates the multi-node VC configuration problem. It shows a partial mesh network and a specification of three VCs, $v_1$, $v_2$ and $v_3$, to be configured in the network. The network also delivers BE traffic. Each VC comprises multiple source and destination nodes and is associated with a bandwidth requirement. Configuring VCs involves (1) *path selection*: This has to explore the network path diversity. It turns out that there exists a huge design space to explore. Suppose that the size of a VC specification set is $m$, each VC has $p$ alternative paths, we have $p^m$ solution possibilities; (2) *slot allocation*: Since VC packets can not contend with each other, VCs must be configured so that an output link of a switch is allocated to one VC per slot. Both steps together must ensure that VCs are set up free from contention and allocated with sufficient slots. The network must be deadlock-free and livelock-free.

### 2.4.2 Logical-Network-oriented VC Configuration

#### A. TDM virtual circuits

Figure 2.12 shows two VCs, $v_1$ and $v_2$, and the respective routing tables for the switches. An output link is associated with a buffer or register. $v_1$ passes switches

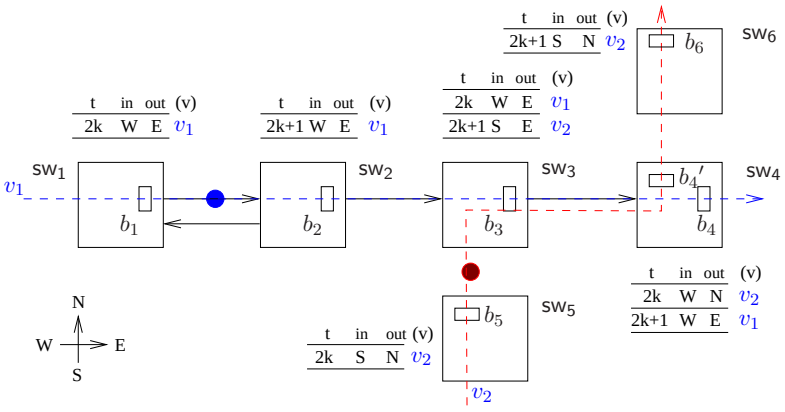



**Figure 2.12.** TDM virtual circuits

$\mathsf{sw}_1$, $\mathsf{sw}_2$, $\mathsf{sw}_3$ and $\mathsf{sw}_4$ through $\{b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4\}$; $v_2$ passes switches $\mathsf{sw}_5$, $\mathsf{sw}_3$, $\mathsf{sw}_4$ and $\mathsf{sw}_6$ through $\{b_5 \rightarrow b_3 \rightarrow b_4' \rightarrow b_6\}$. $v_1$ and $v_2$ only overlap in buffer $b_3$, i.e., $v_1 \cap v_2 = \{b_3\}$. A routing table entry $(t, in, out)$ is equivalent to a routing function $\mathcal{R}(t, in) = out$, where $t$ is time slot, $in$ an input link, and $out$ an output link. For example, $(2k, W, E)$ in $\mathsf{sw}_1$ means that $\mathsf{sw}_1$ reserves its $E$ (East) output link at slots $2k$ ($k \in \mathbb{N}$) for its $W$ (West) inport ($\mathcal{R}(2k, W) = E$). As can also be observed, $\mathsf{sw}_3$ configures its even slots $2k$ for $v_1$ and its odd slots $2k+1$ for $v_2$. As $v_1$ and $v_2$ interleavely use the shared buffer $b_3$ and its associated output link, $v_1$ and $v_2$ do not conflict.

### B. Using logical networks to avoid conflict

We draw a simplified picture of Figure 2.12 in Figure 2.13, where a bubble represents a buffer. VC packets on $v_1$ and $v_2$ are fired once every two cycles. Their bandwidth is $bw_1 = bw_2 = 1/2$ packets/cycle. Suppose that both VCs start admitting packets at slot $t = 0$. $v_1$ packets visit the shared buffer $b_3$ at even slots with an initial latency of two slots; $v_2$ packets visit $b_3$ at odd slots with an initial latency of one slot. This also means that, at even slots, $v_1$ packets hold buffers $b_1$ and $b_3$ while $v_2$ packets hold buffers $b_5$ and $b_4'$; At odd slots, $v_1$ packets hold buffers $b_2$ and $b_4$ while $v_2$ packets hold buffers $b_3$ and $b_6$. Thus $v_1$ and $v_2$ never conflict with each other. Figure 2.13 shows a snapshot of VC packets at even slots.

From the local perspective of buffer $b_3$, the alternate use of this shared buffer by $v_1$ and $v_2$ virtually partitions its time slots into two disjoint sets, the odd set and the even set. The two sets can be regularly mapped to the slot sets of other buffers on

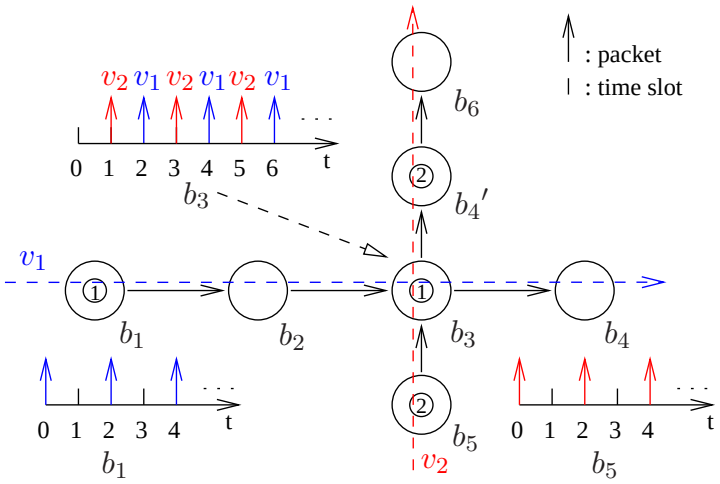



**Figure 2.13.** Using logical networks to avoid conflict

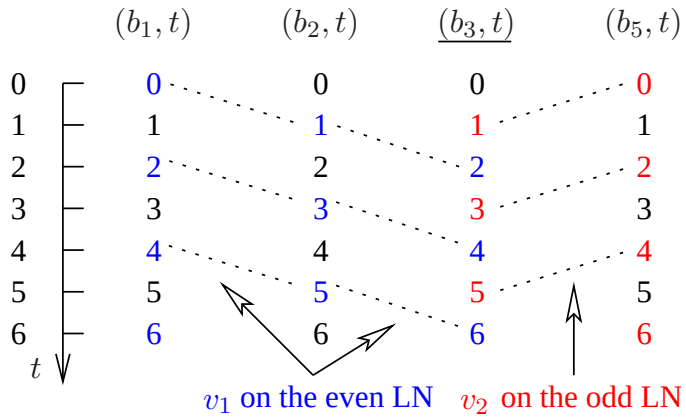



**Figure 2.14.** The view of logical networks

a VC in an unambiguous way. The reason is that, due to the synchronous network operation and contention-free VC-packet transmission, a VC packet advances one step per slot and never stalls, thus a packet visiting $b_1$ at even slots will visit $b_2$ at odd slots, visit $b_3$ at even slots, and so on. Therefore the partitioned slots are networked, as illustrated in Figure 2.14. We can view that $v_1$ and $v_2$ stay in the same physical network but in different logical networks. We define a *logical network (LN)* as a composition of associated sets of time slots in a set of buffers of a VC with respect to a *reference buffer*. We call them LNs because the logically networked slots comprise disjoint networks over time. The overlapping of $v_1$ and

$v_2$ results in two LNs, the even and odd LNs; $b_3$ is the reference buffer. If $v_1$ and $v_2$ subscribe to different LNs, they are conflict-free. In the example, $v_1$ subscribes to the even LN and $v_2$ to the odd LN.

### C. Formal conflit-free conditions

The logical-network concept generalizes the concepts of admission classes [16] and temporally disjoint networks [81]. In Paper 4, we have given formal definitions on the VC and LN. Formally, we have addressed the key questions for the LN-oriented VC configuration. Suppose that $v_1$ and $v_2$ are two overlapping VCs,

- The maximum number $T$ of LNs, which both VCs can subscribe to without conflict, equals to $GCD(D_1, D_2)$, the Greatest Common Divisor (GCD) of their admission cycles $D_1$ and $D_2$. The *admission cycle* $D$ of a VC $v$ is the length (in number of time slots) of its packet-admission pattern. The bandwidth that a LN possesses equals $1/T$ packets/cycle.
- Assigning both VCs to different logical networks is the sufficient and necessary condition to avoid conflict between them.
- If they have multiple shared buffers, these buffers must satisfy *reference consistency* in order to be free from conflict. If so, any of the shared buffers can be used as the reference buffer to construct LNs. Two shared buffers $b_1$ and $b_2$ are termed *consistent* if it is true that "$v_1$ and $v_2$ packets do not conflict in buffer $b_1$" if and only if "$v_1$ and $v_2$ packets do not conflict in buffer $b_2$". The sufficient and necessary condition for them to be consistent is that the distances of $b_1$ and $b_2$ along the two VCs, denoted $d_{b_1\vec{b}_2}(v_1)$ and $d_{b_1\vec{b}_2}(v_2)$, respectively, satisfy $d_{b_1\vec{b}_2}(v_1) - d_{b_1\vec{b}_2}(v_2) = kT,\ k \in \mathbb{Z}$. Furthermore, instead of pair-wise checking, the reference consistency can be linearly checked.

### D. The VC configuration method

We have used the theorems to guide the construction of VCs. We use a backtracking algorithm to constructively search the solution space while exploring the path diversity. The algorithm is a recursive function performing a depth-first search. The solution space in a tree structure is generated while the search is conducted. The backtracking algorithm trades runtime for memory consumption. At any time during the search, only the route from the start node to the current expansion node is saved. As a result, the memory requirement of the algorithm is $O(m)$, where $m$ is the number of VCs. This is important since the solution space organization

needs excessive memory if stored in its entirety. Whenever two VCs overlap, the assignment of LNs to VCs is performed. If they can be assigned to two different LNs with sufficient bandwidth, the assignment is done successfully. Otherwise, the assignment fails. Other path alternatives have to be considered. This VC-to-LN assignment serves as a bounding function by which, if it fails, the algorithm prunes the current expansion node's subtrees, thus making the search efficient.

With a feasible solution, for each VC $v_i$ ($1 \leq i \leq m$) with a normalized bandwidth requirement $\bar{bw}_i$, our VC configuration program returns $(\vec{P}_i, D_i, \vec{A}_i, R_i)$, where $\vec{P}_i$ is the sequence of buffers visited by $v_i$, representing its delivery path; $D_i$ is the admission cycle by which $v_i$ repeats its packet-injection pattern; $\vec{A}_i$ is the allocated slot vector whose size $|\vec{A}_i|$ is the number of slots in $\vec{A}_i$, and $\forall A_{i,j} \in \vec{A}_i$ ($1 \leq j \leq |\vec{A}_i|$), $A_{i,j} \in [0, D_i)$ and $|\vec{A}_i| \in (0, D_i]$; $R_i$ is the reference buffer for which the time slots are referred to. The LN(s) that $v_i$ subscribes to is reflected in $\vec{A}_i$ explicitly, or implicitly if $v_i$ uses only a portion of bandwidth in the allocated LN(s). In addition, they satisfy the bandwidth constraint:

$$\frac{|\vec{A}_i|}{D_i} \geq \bar{bw}_i$$

As an example, for the two VCs, $v_1$ and $v_2$, in Figure 2.13, $D_1 = D_2 = 2$. The number $T$ of logical networks is $GCD(2, 2) = 2$. The result of configuring $v_1$ is $(< b_1, b_2, b_3, b_4 >, 2, \{0\}, b_3)$, which is equivalent to $(< b_1, b_2, b_3, b_4 >, 2, \{0\}, b_1)$. This means that $v_1$ packets are fired from $b_1$ at even slots, once every two cycles. The result of configuring $v_2$ is $(< b_5, b_3, b'_4, b_6 >, 2, \{1\}, b_3)$, which is equivalent to $(< b_5, b_3, b'_4, b_6 >, 2, \{0\}, b_5)$. This means that $v_2$ packets are fired from $b_5$ at even slots, once every two cycles.

## 2.5 Future Work

NoC communication architectures need to offer various services with different guarantees and efficient support for different communication patterns such as multicast, peers and client-server, to provide robust and reliable communication, to enable re-configuration, and to reduce area and power budget. In the future, our research may be complemented along the following threads:

- *Contract-oriented virtual-circuit configuration*: While configuring virtual-circuits (VCs), the network can satisfy their requirements in two ways. One is to meet their demand on its own. The other is to eventually generate contracts through possible negotiation, one for each VC. According to the

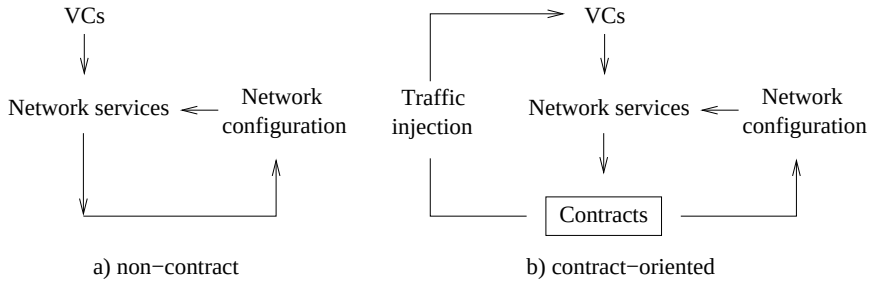



**Figure 2.15.** Virtual-circuit configuration approaches

contracts, the network nodes configure slot tables. Moreover, VC traffic is injected obligating to the contracts. We illustrate the two approaches in Figure 2.15. Figure 2.15a is non-contract oriented while Figure 2.15b is contract-oriented. As we can see, there is an additional feedback loop introduced in the contract-oriented approach. This loop defines the obligation of VCs in terms of traffic injection pattern. In this way, both the network and VCs must fulfill their obligations. Our VC configuration approach generates $(\vec{P}, D, \vec{A}, R)$ for each VC. This in fact constitutes a *contract*. The network configures slot tables along the VC delivery path using the contracts, and the VCs regularly launch packets using the allocated slots. We expect that a contract-oriented method can facilitate predictable IP integration and the formal validation of QoS guarantees in comparison with a noncontract-oriented one. These benefits as stated have not yet been substantiated.

- *Reconfigurable QoS network architectures*: SoC applications are becoming extremely functionally rich. For example, a personal handheld set does telephoning, multimedia processing, gaming, and may execute diverse web-based utilities. A network designed for such systems must be reconfigurable because different use cases require different configurations. Satisfying all use cases concurrently may over-design the network, leading to unacceptable cost. To have a dynamically re-configurable communication platform is most cost- and power-efficient since it allows us to allocate and use resources only if it is necessary. The challenge is not to sacrifice performance, efficiency and predictability when allowing adaptivity. Efficient protocols, micro-architectures and methods are in a great need to support network reconfigurability. Error-resilient and self-healing mechanisms can be further incorporated to provide fault tolerance and robustness to cope with the nano-regime uncertainties.

# Chapter 3

# NoC Network Performance Analysis

*This chapter summarizes our simulation-based and algorithm-based NoC performance analysis [Paper 5, 6, 7]. The simulation-based analysis [Paper 5, 6] is performed within the Nostrum Network-on-chip Simulation Environment (NNSE). The algorithm-based approach addresses the feasibility test of delivering real-time messages in wormhole-switched networks [Paper 7].*

## 3.1 Introduction

### 3.1.1 Performance Analysis for On-Chip Networks

#### A. On-chip network performance analysis

Network-on-chip provides a structured communication platform for complex SoC integration. However, it aggravates the complexity of on-chip communication design. From the network perspective, there exists a huge design space to explore at the network, link and physical layers. In the network layer, we need to investigate topology, switching, routing and flow control. In the link layer, we can examine the impact of link capacity and link-level flow control schemes on performance. In the physical layer, we could inspect wiring, signaling, and robustness issues. Each of the design considerations (parameters) also has a number of options to consider. From the application perspective, the network should not only be customizable but also be scalable. To design an efficient and extensible on-chip network that suits a specific application or an application domain, performance analysis is a crucial

and heavy task. The impact of the design parameters at the different layers and the performance-cost tradeoffs among these parameters must be well-understood. The customization on optimality and extensibility can sometimes be in conflict with each other. For instance, a customized irregular topology may be optimal but not easy to scale. In addition, the analysis task is very much complicated because of the un-availability of domain-specific traffic models. Due to the separation between computation and communication, a communication platform may be designed in parallel with the design of computation. The concurrent development speeds up time-to-market, but leaves the development of the communication platform without sufficiently relevant traffic knowledge. Therefore we must be able to evaluate network architectures and analyze their communication performance with various communication patterns extensively so as to make the right design decisions and trade-offs. Once a network is constructed in hardware, it is difficult, time-consuming, and expensive to make changes if performance problems are encountered.

Design decisions include both architecture-level decisions such as topology, switching, and routing algorithm, and application-level decisions such as task-to-node mapping, task scheduling and synchronization etc. While evaluating network architectures and analyzing their performance, we can embed design decisions in experiments during the evaluation and analysis process. In turn, this helps to seek for optimal network and application constructions. Making design decisions is likely to be an iterative process. The feedback information in such a process includes functional and nonfunctional measures. Functional criteria are typically bandwidth, latency, jitter, and reliability, which can be broadly classified into quality-related metrics. Nonfunctional criteria are network utilization, area and power consumption, which are all cost-related.

### B. Network performance analysis methods

We may classify network performance analysis methods (before prototyping and implementation) into three categories: *simulation-based*, *algorithm-based* and *mathematics-based*. Below, we give a brief account on them to the extent that is adequate for the introduction.

The *simulation-based* approach builds network and traffic models and then the network operation is simulated by loading the traffic into the network. The network models can be constructed in detail or in an abstract way. For example, a switch model can model all its functional components such as buffers, crossbar and control units in detail. Alternatively, a switch can only model the packet shuffling behavior without modeling each component. A switch may model the internal pipeline

stages, leading to a multiple-cycle model. Or it can be just a single-cycle model in which packet switching completes in one cycle. More abstractly, a network may be modeled without building detailed switch models. But the network behavior such as routing and arbitration is modeled so that the net effect of packet delivery such as delay and jitter is reflected in the results. In a simulation-based approach, both synthetic and realistic traffic models can be applied. Furthermore, it allows us to perform system-wide simulation where the interaction between the network and traffic sources/sinks may be captured and the performance-cost tradeoff is examined [103, 114]. The evaluation of the network performance is conducted after simulation statistics are collected. The simulation speed can be different depending on the modeling details [78].

The *algorithm-based* approach makes assumptions on network communication models. In the communication model, the network delivery characteristics and switch arbitration behavior are captured. Additional models may be created to reflect network contention. The network behavior can thus be approximated. Based on the models, an algorithm is then developed to conduct the performance evaluation without resorting to detailed simulation. An algorithm-based approach usually assumes that traffic has certain properties, for example, periodicity and independence. Examples using the algorithm-based approach can be found in [7, 40, 55, 69].

The *mathematics-based* approach builds mathematical models for network and traffic. The performance figures are calculated through formal derivation. For instance, two basic analytic tools for network performance evaluation are queuing theory and probability theory [28]. Queuing theory [36] is useful for analyzing a network in which packets spend much of their time waiting in queues. Probability theory is more useful in analyzing networks in which most contention time is due to blocking rather than queuing. Another example is the use of the network calculus [22, 23] to compute the end-to-end delivery bounds. The mathematics-based approach is most efficient but limited in capability. It can model many aspects of a network, but there are some situations that are simply too complex to express under the mathematical models. Besides, it often simplifies the real situations by making a number of approximations that may affect the accuracy of results.

The performance analysis methods, as described above, are not isolated. They can be used to validate against each other. To validate a model, we need to compare its results against known good data at a representative set of operating points and network configurations. They may be composed to take the advantages of each method. For instance, simulation and formal methods may be combined to speed up the simulation-based performance analysis [57].

### C. A comparison of network performance analysis methods

| | **Simulation** | **Algorithm** | **Mathematics** |
|---|---|---|---|
| Com. model | Detailed/Abstracted | Simplified | Accurate/Simplified |
| Evaluation | Cycle-true/Behavior sim. | Run algorithms | Formal derivation |
| Execution time | Slow/Medium | Medium | Fast |
| Accuracy | High/Medium | Medium | High/Medium |
| Capability | High | Low | Medium |

**Table 3.1.** Network performance analysis methods

All the performance analysis methods require building network and traffic models. They mainly differ in modeling details, efficiency, quality-of-result (accuracy) and capability. We compare the three methods in Table 3.1[1]. The simulation-based method can offer the highest accuracy but may be very time-consuming. Each simulation run can take considerable time and evaluates only a single network configuration, traffic pattern, and load point. It is difficult, if not impossible, to cover all the system states. Depending on the details simulated, runtime and accuracy trade off with each other. However, simulation, in contrast to emulation and implementation, is flexible and cheap. It can also model complex network designs for which mathematical or other analytical models are difficult to build. A simulation tool usually enables to explore the architectural design space and assess design quality regarding performance, cost, power and reliability etc. The algorithm-based scheme does not run network simulation, but the network behavior is captured in an algorithm. It is generally faster than simulation-based schemes, but only approximates the simulated results. The mathematical analysis [2, 35] is the most efficient one. It provides approximate performance numbers with a minimum amount of effort and gives insight into how different factors affect performance. It also allows an entire family of networks with varying parameters and configuration to be evaluated at once by deriving a set of equations that predict the performance of the entire family. The accuracy of results depends on the accuracy of the mathematical models for the traffic and network. It can be rough but gives an initial and quick estimation. A performance bound may be also tight enough.

[1] Note that the qualitative assessments on run-time, accuracy and capability emphasize the differences between the methods. They are relative, and should not be considered absolute.

As simulation is most powerful, once it is verified, it is typically used to validate the algorithm-based and formalism-based approaches. In the next subsection, we outline current practices of NoC simulation.

### 3.1.2 Practices of NoC Simulation

NoC researchers have used general-purpose network simulators and NoC-specific simulators to simulate the network behavior. OPNET is a commercial network simulator used in [15, 133]. It provides a tool for hierarchical modeling and includes processes, network topology description and supports different traffic scenarios. However, to simulate an on-chip network, it has to be adapted by explicitly modeling synchronous operations and distribution [133]. OMNET is an open-source C++-based network simulation engine. It is used in [89] to validate a network contention model proposed in [69]. As with OPNET, additional modules are needed to model synchronous network operations in OMNET. Semla [125, 126] is a dedicated NoC simulator written in SystemC [20]. It implements five layers of the OSI seven-layer model (without the presentation and session layers), and is equipped with transaction-level primitives to communicate messages between application processes. The SystemC kernel provides the concurrent and synchronous operation semantics, thus a SystemC-based network simulator can take this advantage. In [8], a VHDL-based RTL model is created for evaluating power and performance of NoC architectures. It can model dynamic and leakage power at the system-level. The Orion [131] performance-power simulator models only the dynamic power consumption.

OCCN (On Chip Communication Network) [21] models on-chip network communication in SystemC using high-level modeling concepts such as transactions and channels. In [77], an on-chip communication network is treated as a *communication processor* to reflect servicing demands. The network is modeled using allocators, schedulers and synchronizers. The allocator decides the resource requirements such as bandwidth and buffers along a message's path while minimizing resource conflict. The scheduler executes the message transfer accordingly, minimizing the resource occupation. The synchronizer performs synchronization according to dependencies among messages while allowing concurrency. In [32], network communication is defined as a multiport blackbox communication structure. A message can be transmitted from an arbitrary port to another but the actual implementation of the NoC may not be considered.

Next, we present our NoC simulation tool NNSE in Section 3.2. In Section 3.3, we present our algorithm-based network performance analysis, focusing on the fea-

sibility test of delivering real-time messages, i.e., whether their timing constraints can be met or not.

## 3.2 NNSE: Nostrum NoC Simulation Environment

### 3.2.1 Overview

NNSE stands for Nostrum NoC Simulation Environment in which Nostrum is the name of our NoC concept [81]. NNSE is aimed to be a tool for full NoC system simulation so that designers can use it to explore the architecture-level and application-level design space. Currently, it is capable of

- constructing network-based communication platforms [125],
- generating synthetic and semi-synthetic Traffic Patterns (TPs) [68],
- simulating the communication behavior with the various TPs [76], and
- mapping application tasks onto the platform [71].

The first three functions have been automated and the last function is so far a manual step. The automation is achieved through parameterizing network and synthetic traffic configurations. One can configure these parameters, recompile the program if the parameters are compile-time, and invoke simulations with the specified network and traffic configurations. This procedure can be conducted in a Graphical User Interface (GUI). With the GUI, the tool allows us to easily explore different network architectures and different traffic settings. Network architectures can thus be efficiently and extensively evaluated. In addition to using synthetic traffic, the manual application mapping creates realistic traffic scenarios in the communication platform. The evaluation may be iterative by applying the configured or created traffic on the configured networks, as illustrated in Figure 3.1. The evaluation criteria can be performance, power and cost. The current version evaluates only the network performance in terms of packet latency, link utilization and throughput. Since it simulates the network behavior at the flit-level cycle-by-cycle, the performance estimates are accurate.

NNSE logically comprises a NoC simulation kernel [124, 125] wrapped with a GUI. The kernel is developed in SystemC and the GUI written in Python. Following the ISO's OSI seven-layer model [135], the simulation kernel called *Semla* (Simulation EnviornMent for Layered Architecture) implements five of the seven layers except for the representation and session layers. The simulation tool presently

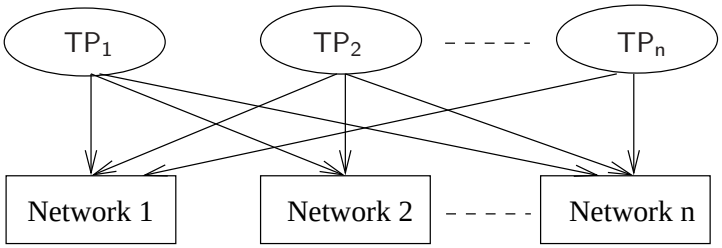



**Figure 3.1.** Network evaluation

supports the configuration of the network and application layers in the GUI. The configuration of the application layer refers to the traffic configuration. In the GUI, all the network and traffic configurations can be stored and thus reusable. To facilitate data exchange, they are stored as eXtensible Markup Language (XML) files. The simulation results can be shown graphically or in a text format.

### 3.2.2 The Simulation Kernel

The simulation kernel Semla [124, 125] implements the five communication layers, namely, the physical layer (PL), the data link layer (LL), the network layer (NL), the transport layer (TL) and the application layer (AL). The upper three layers are shown in Figure 3.2, where TG/S stands for Traffic Generator/Sink, and Glue is the TL component which does packetization/packet-assembly, message queuing, multiplexing, de-multiplexing and so on.

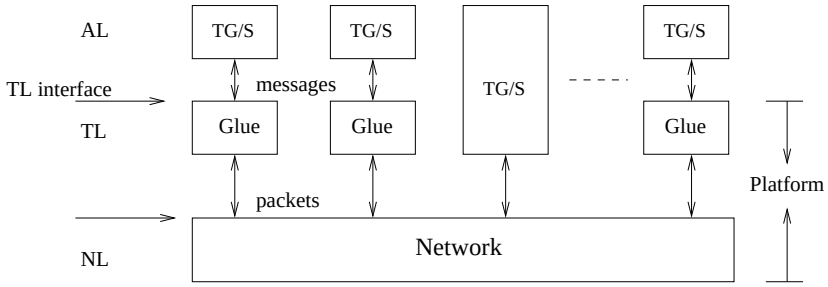



**Figure 3.2.** The communication layers in Semla

The transport layer provides transaction-level communication primitives as an interface to enable communication via *channels* between application processes. A channel, similar to a SystemC channel [20], is a transaction-level modeling entity which allows simplex communication from a source process to a destination

process. In Semla, a compact set of message passing primitives for using the best-effort service is defined and implemented:

- *int open_channel(int src_pid, int dst_pid)*: it opens a simplex channel from a source process *src_pid* to a destination process *dst_pid*. The method returns a positive integer as a unique channel identity number *cid* upon successfully opening the channel. Otherwise, it returns a negative integer for various reasons of failure, such as invalid source and destination processes. The current implementation opens channels statically during compile time and the opened channels are never closed through simulation.
- *bool nb_write(int cid, void msg)*: it writes *msg* to channel *cid*. The size of messages is finite. It returns the status of the write. The write is nonblocking.
- *bool nb_read(int cid, void *msg)*: it reads channel *cid* and writes the received protocol data unit to the address starting at *msg*. It returns the status of the read. The read is nonblocking.

Application tasks use the set of communication primitives to communicate messages with each other. While mapping tasks onto the NoC platform, the network topology is visible. The communication part of the tasks must be written in or adapted to the communication primitives. The interaction between the tasks creates realistic workload in the platform, and the system behavior can be simulated.

Thanks to the layering, one can design and implement different structures and protocols in a layer without modifying other layers as long as one complies with the interfaces. For instance, Semla originally developed the network layer for deflection routing. In order to perform experiments on flit-admission and flit-ejection schemes in Chapter 2, the network layer for wormhole switching was developed and integrated into the simulator. The physical layer was skipped because the interest was on the flit-level not the phit-level activities. While the compilation and simulation were invoked, only the network layer entity was replaced while the upper layers remained the same.

### 3.2.3 Network Configuration

We parameterize a network according to topology, switching mode and routing algorithm. The network configuration is thus straightforward, as illustrated by the tree in Figure 3.3. The topology is for a 2D regular structure, which can be dimensioned along the number of nodes on the X axis, the number of nodes on the Y axis. The structure may be chosen from one of the options (mesh, torus, tree,

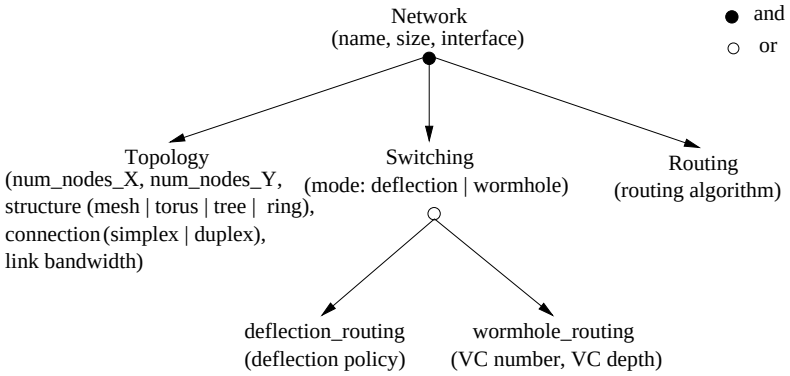



**Figure 3.3.** Network configuration tree

ring). The link connection may be simplex or duplex with different data width. The network parameters may be further elaborated resulting in the next level in the tree since each of them has a number of choices on its own. As can be seen, with the deflection scheme, different deflection polices may be chosen; with the wormhole scheme, the number and depth of virtual channels (VCs) may be specified.

### 3.2.4 Traffic Configuration

This subsection summarizes the research in Paper 5.

#### A. Traffic configuration approaches

Network evaluation typically employs *application-driven* and *synthetic* traffic [28]. Application-driven traffic models the network and its clients simultaneously. This is based on full system simulation and communication traces. Full system simulation requires building the client models. Application-driven traffic can be too cumbersome to develop and control. In NNSE, application-driven traffic is created by mapping application tasks onto the communication platform. Synthetic traffic captures the prominent aspects of the application-driven workload but can also be easily designed and manipulated. Because of this, synthetic traffic is widely used for network evaluation.

In NNSE, two types of traffic can be configured. One is purely *synthetic traffic*, the other *application-oriented traffic*. For synthetic traffic, we proposed a unified formal expression for both uniform and locality traffic. With this expression, we can control the locality of traffic distribution by setting locality factors for the traffic. The application-oriented traffic is semi-synthetic, which can be viewed as a

traffic type between application-driven traffic and synthetic traffic. It statically defines the spatial distribution of traffic on a per-channel basis according to application, and the temporal and size distributions of each channel may be synthetic or extracted from communication traces.

### B. The traffic configuration tree

Traffic can be characterized and constructed via its distributions over three dimensions: *spatial distribution*, *temporal characteristics*, and *message size specification*. The spatial distribution defines the communication patterns between sources and destinations. The temporal characteristics describe the message generation probability over time. The size specification gives the length of generated messages. We use a traffic configuration tree to express the elements and their attributes in Figure 3.4.

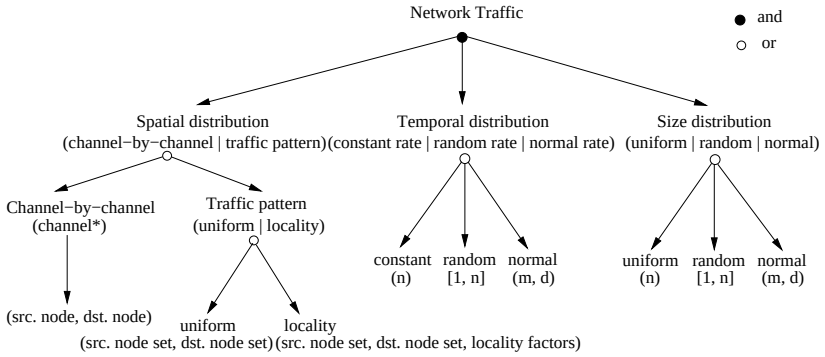



**Figure 3.4.** The traffic configuration tree

By the spatial distribution, traffic is broadly classified into two categories: *traffic pattern* and *channel-by-channel* traffic. In a traffic pattern, all the channels share the same temporal and size parameters. In contrast, channel-by-channel traffic consists of a set of channels, and each channel can define its own temporal and size parameters. The temporal distribution has a list of candidates such as constant rate (periodic), random rate, and normal rate etc. The size distribution has a list of choices such as uniform, random, and normal. As can be observed, these lists are just examples of possible distributions. Other useful distributions can be integrated into the tree with their associated parameters. According to the tree, configuring a traffic pattern is to select a set of parameters on the three axes. Note that the axes may not be independent. For instance, scale-invariant burstiness traffic and scale-variant burstiness (self-similar) traffic [100] involve the variation in the time scale

and message size, thus requiring the synergy between the temporal distribution and the size distribution.

### C. Representation of traffic patterns

As shown in Figure 3.4, the traffic patterns consist of *uniform* and *locality* traffic. They can be uniformly expressed in a formal representation.

We first define *communication distribution probability* $DP$ from node $i$ to node $j$ $DP_{i\rightarrow j}$ as the probability of distributing messages from node $i$ to node $j$ while node $i$ sends messages to the network. Suppose, there are $N$ nodes in the network, Equation 3.1 means that all messages from node $i$ are aimed to the $N$ destination nodes.

$$\sum_{j=1}^{N} DP_{i\rightarrow j} = 1 \tag{3.1}$$

Next, we relate $DP$ to the minimal distance between nodes. Let the shortest distance between a source node $i$ and a destination node $j$ be $d$, we define communication distribution probability $DP_{i\rightarrow j}$ as a relative probability to a common probability factor $P_c$ ($0 \leq P_c \leq 1$) in Equation 3.2.

$$\begin{aligned} DP_{i\rightarrow j} &= coef(\alpha, d) \cdot P_c \\ where\ coef(\alpha, d) &= 1 + \frac{\alpha}{d+1} \end{aligned} \tag{3.2}$$

In the equation, *coef* is the *distribution coefficient* and $\alpha$ called *locality factor*. Since $DP_{i\rightarrow j} \geq 0$, $\alpha \geq -(d+1)$. Particularly when $\alpha = -(d+1)$, $DP_{i\rightarrow j} = 0$; when $\alpha = 0$, $DP_{i\rightarrow j} = P_c$. Besides, when $-(d+1) < \alpha < 0$, $DP_{i\rightarrow j}$ is proportional to distance $d$; When $\alpha > 0$, $DP_{i\rightarrow j}$ is inversely proportional to distance $d$. In addition, $\alpha(d)$ can be defined for each possible value of distance $d$.

Using the traffic expression, the locality of traffic distribution can be easily controlled by setting $\alpha(d)$ for each possible distance value $d$. For instance, if $\alpha(d) = -(d+1)$, $coef(-(d+1), d) = 0$ meaning that no traffic is generated between sources and destinations if their shortest distance is $d$; if $\alpha(0) = -1$ for $d = 0$, $coef(-1, 0) = 0$, meaning that no self-loop traffic is created. If we set "$\alpha(d) = 0$" for all possible values of $d$ in the network, their distribution coefficients $coef(0, d) = 1$. Then for any source node $i$, it has an equal probability to distribute traffic to any node $j$. In this case, the traffic distribution is independent of distance $d$, meaning that the traffic is uniform. After setting $\alpha(d)$, we can calculate $coef(\alpha, d)$ and $P_c$ using Equations 3.1 and 3.2. Then $DP(d)$ can be derived. An example of the calculation is given in Paper 5.

**D. Channel-by-channel traffic**

For the traffic patterns, we control the traffic generation and locality by setting a locality factor $\alpha$ for each possible distance $d$. Since one distance may cover a number of pairs of source and destination nodes, we avoid specifying the communication distribution probabilities for each source node to each and every possible destination node. For channel-by-channel traffic, as the name suggests, we set traffic parameters for each individual channel. The set of traffic parameters of a channel is $\{s\_proc, d\_proc, \mathcal{T}, \mathcal{S}\}$, where $s\_proc$ represents the source process, $d\_proc$ the destination process, $\mathcal{T}$ its temporal characteristics, and $\mathcal{S}$ its message size specification. For each channel, we can determine the source node for $s\_proc$ and the destination node for $d\_proc$ after the application task graph is mapped onto the network nodes. The temporal characteristics $\mathcal{T}$ and the message size specification $\mathcal{S}$ can be synthetically configured using the same set of options in the tree or approximated using analysis or communication traces [68].

Channel-by-channel traffic differs from the traffic patterns mainly in that the traffic's spatial pattern is statically built on a per-channel basis according to an application task graph. Since the communication pattern in the task graph is captured, this type of traffic is used to construct application-oriented workloads.

### 3.2.5 An Evaluation Case Study

As a case study (Paper 6), we have evaluated deflection networks in NNSE [76]. A deflection-routed network (see Section 2.1 of Chapter 2) has three orthogonal characteristics: *topology*, *routing algorithm* and *deflection policy*. It is crucial to explore the alternatives of the three aspects since the decisions on these aspects may be hardwired and may not be dynamically configurable or too costly to permit dynamic configuration. Therefore identifying the significance of each factor and evaluating their alternatives play a vital role in the decision-making.

In the evaluation, we have considered 2D regular topologies such as *mesh*, *torus* and *Manhattan Street Network*, different routing algorithms such as *random*, *dimension XY, delta XY* and *minimum deflection*, as well as different deflection policies such as *non-priority*, *weighted priority* and *straight-through* policies [76]. Our results suggest that the performance of a deflection network is more sensitive to its topology than the other two parameters. It is less sensitive to its routing algorithm, but a routing algorithm should be minimal. A priority-based deflection policy that uses global and history-related criterion can achieve both better average-case and worst-case performance than a non-priority or priority policy that uses only local and stateless criterion. These findings are important since they can

guide designers to make right decisions on the network architecture, for instance, selecting a routing algorithm or deflection policy which has potentially low cost and high speed for hardware implementation.

## 3.3 Feasibility Analysis of On-Chip Messaging

This section summarizes the research in Paper 7.

### 3.3.1 Problem Description

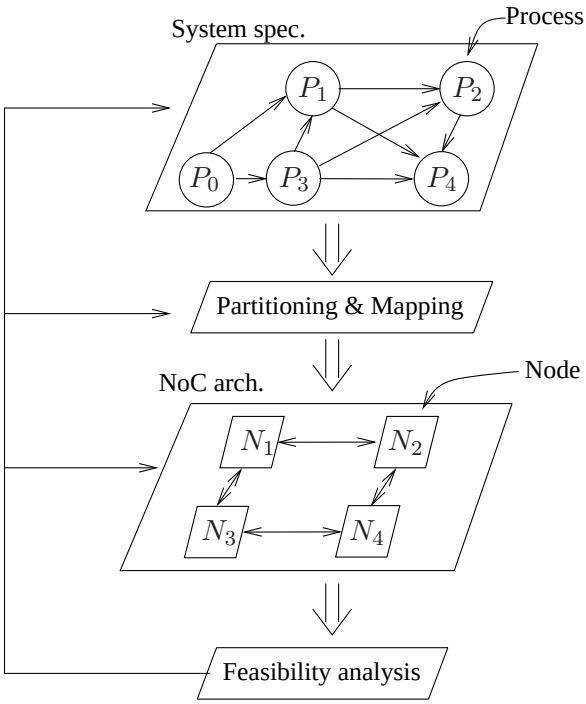



**Figure 3.5.** Feasibility analysis in a NoC design flow

As illustrated in Figure 3.5, NoC design starts with a system specification which can be expressed as a set of communicating tasks. The second step is to partition and map these tasks onto the resources of a NoC. With a mapping, application tasks running on these resources load the network with messages, and

impose timing constraints for delivering messages. The feasibility analysis is performed on the resulting NoC instance. Feasibility analysis could, on its own, cover a wide range of evaluation criteria such as performance, power and cost. In our context, we concentrate on the timely delivery of messages, which is essential for performance and predictability.

Following [110], we distinguish *real-time* and *nonreal-time* messages in on-chip networks. Messages with a deterministic performance bound, which must be delivered predictably even under worst case scenarios, are *real-time* (RT) messages. Messages with a probabilistic bound, which ask for an average response time, are *nonreal-time* messages. Our focus in the thesis is on the feasibility analysis of delivering RT messages in a wormhole-switched network. We follow the feasibility definition in [7]: *Given a set of already scheduled messages, a message is termed* feasible *if its own timing property is satisfied irrespective of any arrival orders of the messages in the set, and it does not prevent any message in the set from meeting its timing property.* We resort to an algorithm-based instead of simulation-based approach in the analysis to avoid cycle-by-cycle simulations. Since it is the network contention that makes the message delivery non-deterministic, we formulate a *contention tree* model that captures direct and indirect network contentions and reflects concurrency in link usage. Based on this model, we investigate message scheduling to estimate the worst-case performance for RT messages and develop an algorithm to conduct the feasibility analysis. The analysis returns the pass ratio, i.e., the percentage of feasible messages, and the network utilization of the feasible messages.

In the following, we first describe the contention tree model, message scheduling on a contention tree, and then the feasibility analysis flow.

### 3.3.2 The Network Contention Model

#### A. The real-time communication model

Wormhole switching divides a message into a number of flits for transmission[2]. During the delivery, it manages two types of resources, the lanes and the link bandwidth. Lanes are flit buffers organized into several independent FIFOs instead of a single FIFO. Lane allocation is made at the message level while link bandwidth is assigned at the flit level. In conventional wormhole switches, the shared lanes are arbitrated on First-Come-First-Serve (FCFS), and the shared link bandwidth are multiplexed by the lanes. Messages are not associated with a priority and they are equally treated. This model is fair and produces average-case performance results.

[2]The effect of packetization is not considered here.

It is suitable to deliver nonreal-time messages, which do not require guarantees. But, it can not directly support real-time messages because there is no promise that messages are delivered before deadlines. In order to enable guarantees, real-time messages must be served with other disciplines, for instance, priority-based arbitrations [62].

We assume a conventional wormhole switch architecture and a priority-based delivery model for RT messages. Special RT communication services generally require special architectural support which may potentially complicate the switch design. All messages are globally prioritized, and priority ties are resolved randomly. This model arbitrates shared lanes and link bandwidth on priority. The priority, which may be assigned according to rate, deadline or laxity [40, 62], takes a small number of flits. With this RT communication model, the worst-case latency $T^{rt}$ of delivering a message of $L$ flits is given by :

$$T^{rt} = (L + L_{pri})/B^{rt} + HR + \tau = T + \tau \tag{3.3}$$

where $B^{rt}$ is the link bandwidth allocated to the RT message along its route; $H$ is the number of hops from the source node to the destination node; $R$ is the routing delay per hop; $L_{pri}$ is the number of flits used to express the message priority. The routing delay $R$ is assumed to be the same for head flits and body/tail flits. The first term counts for the transmission time of all the message flits; the sum of the first two terms is the non-contentional or base latency $T$, which is the lower bound on $T^{rt}$; the last term $\tau$ is the worst-case blocking time due to network contention.

**B. Network contention**

To estimate the worst-case latency $T^{rt}$ of an RT message $M$, we have to estimate the worst-case blocking time $\tau$. To this end, we first determine all the contentions the message may meet.

In flit-buffered networks, the flits of a message $M_i$ are pipelined along its routing path. The message advances when it receives the link bandwidth along the path. The message may directly and/or indirectly contend with other messages for shared lanes and link bandwidth. $M_i$ has a higher priority set $S_i$ that consists of a *direct contention* set $S_{D_i}$ and an *indirect contention* set $S_{I_i}$, $S_i = S_{D_i} + S_{I_i}$. $S_{D_i}$ includes the higher priority messages that share at least one link with $M_i$. Messages in $S_{D_i}$ directly contend with $M_i$. $S_{I_i}$ includes the higher priority messages that do not share a link with $M_i$, but share at least one link with a message in $S_{D_i}$, and $S_{I_i} \cap S_{D_i} = \emptyset$. Messages in $S_{I_i}$ indirectly contend with $M_i$. As an example, Fig. 3.6a shows a fraction of a network with four nodes and four messages. The messages $M_1$, $M_2$, $M_3$ and $M_4$ pass the links AB, BC, AB→BC→CD, and CD,

respectively. A lower message index denotes a higher priority. The message $M_1$ has the highest priority, thus $S_1 = \emptyset$. For the message $M_2$, it directly contends with $M_3$, but it has a higher priority, thus $S_2 = \emptyset$. The message $M_3$ has a higher priority message set $S_3 = S_{D_3} = \{M_1, M_2\}$, $S_{I_3} = \emptyset$. For the message $M_4$, $S_{D_4} = \{M_3\}$ and $S_{I_4} = \{M_1, M_2\}$ because $M_1$ or $M_2$ may block $M_3$ which in turn blocks $M_4$.

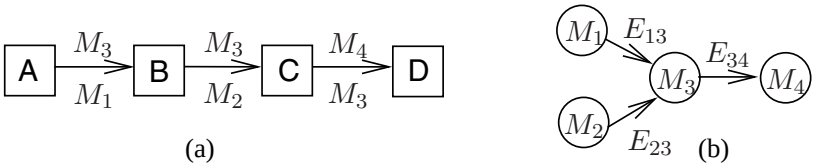



**Figure 3.6.** Network contention and contention tree

### C. The contention tree

To capture both direct and indirect contentions and to reflect concurrent scheduling on disjoint links, we have formulated a *contention tree* model that is defined as a directed graph $G : M \times E$. A message $M_i$ is represented as a node $M_i$ in the tree. An edge $E_{ij}(i < j)$ directs from node $M_i$ to node $M_j$, representing the direct contention between $M_i$ and $M_j$. $M_i$ is called *parent*, $M_j$ *child*. Given a set $n$ of RT messages, after mapping the messages to the target network, we can build a contention tree with the following three steps:

**Step 1.** Sort the message set in descending priority sequence with a chosen priority assignment policy.

**Step 2.** Determine the routing path for each of the messages.

**Step 3.** Construct a tree, starting with the highest priority message $M_1$, and then $M_2 ... M_n$. If $M_i$ shares at least one link with $M_j$ where $i < j \leq n$, an edge $E_{ij}$ is created between them. Each node in the tree only maintains a list of its parent nodes.

In a contention tree, a direct contention is represented by a directed edge while an indirect contention is implied by a *walk* via parent node(s). A walk is a path following directed edges in the tree. The contention tree for Fig. 3.6a is shown in Fig. 3.6b, where the three direct contentions are represented by the three edges $E_{13}$, $E_{23}$ and $E_{34}$, and the two indirect contentions for $M_4$ are implied by the two walks $E_{13} \rightarrow E_{34}$ and $E_{23} \rightarrow E_{34}$ via $M_4$'s parent node $M_3$. Since determining the

routing path is a priori, creating a contention tree is more suitable for deterministic routing. For adaptive routing, it is difficult to figure out the worst-case routing path.

**D. Assumptions and simplifications**

The estimation of latency bounds are based on messages' schedules on links. A schedule is a timing sequence where a time slot is occupied by a message or left empty. The latency bound of a message is the earliest possible completion time for delivery under the worst case. Before introducing schedules of messages, we list the assumptions, limitations and simplifications as follows:

- The messages we consider are periodic and independent. There is no data dependency among messages so that each message can be periodically fired or activated, meaning that the messages are sent to the network and start to compete for shared resources, i.e., buffers and links.
- We focus on link contentions. Similarly to [7, 40], we assume that there is a sufficient number of Virtual Channels (VCs) so that priority inversion due to VC unavailability does not occur. Priority inversion happens when a message with a lower priority holds shared resources, leading to blocking messages with a higher priority. As discussed in [7, 40], this problem can be alleviated by packetization.
- In this communication model, messages are allocated with time slots depending on their priorities and contentions. Whenever there is a contention for a link, a message with a higher priority will be scheduled first. In addition, a higher priority message can preempt a lower priority message.
- The worst case is assumed to occur when all the messages are fired into the network at the same time.
- The bandwidth of a link is assumed to transmit one flit in one time slot. The routing delay per hop takes one time slot. We simplify the pipeline latency on links so that the flits of a message are available to compete *all* the link bandwidth along the message's path simultaneously for the duration of its communication time. To explain this, we illustrate a message transmission in Figure 3.7, where $M_2$ passes through three hops (A, B, C) and two links (AB, BC). $M_2$ contains four flits (one head h flit, two body b flits and one tail t flit). It has a base latency of 7 ($1 \cdot 3 + 4$). If $M_2$ fires at time instant 0, by the assumption, it will compete for *both* links AB and BC from slot 1 to 7, i.e., for its entire base latency period.

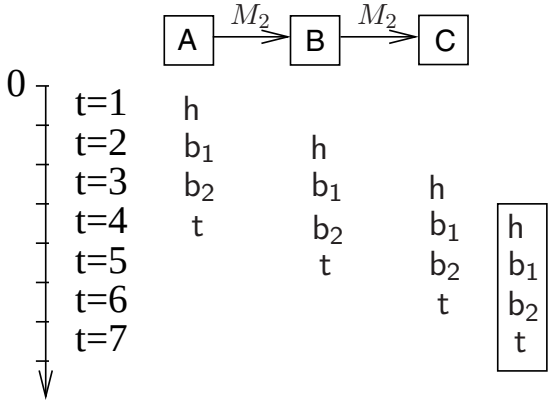



**Figure 3.7.** Message contention for links simultaneously

- We assume that a message advances only if it simultaneously receives all the link bandwidth along its path. This means that the flits are delivered either concurrently via the links or blocked in place. As a result, a message competes for links only for its base latency period. It does not happen that a flit advances via a link while another flit is blocked in place. As shown in Figure 3.8, at time slot 3, the head flit h has advanced from node B to C but the first body flit $b_1$ is blocked in node A. As a consequence, the pipeline latency is increased by one slot. According to our assumption, this scenario in time slot 3 is avoided and thus not considered. Apparently, if flits are individually routed via links, the contention period may become larger than its base latency and unpredictable.

### E. Scheduling on the contention tree: an example

Table 3.2 shows an example of message parameters for Fig. 3.6, where the priority is assigned by rate, and the deadline $D$ equals period $p$. The worst-case schedules for the three links are illustrated separately in Fig. 3.9a. Initially, all messages are fired. $M_1$ is allocated 7 slots on link AB. $M_2$ is allocated 3 slots on link BC. $M_3$ is blocked by $M_1$ and $M_2$. $M_4$ is blocked by $M_3$. After $M_1$ and $M_2$ complete transmission, $M_3$ is allocated 3 slots concurrently on link AB, BC and CD. At time slot 10, $M_1$ fires again and holds slots $[11, 17]$ on link AB, preempting $M_3$. At time slot 15, $M_2$ fires the second time and holds slots $[16, 18]$ on link BC. After $M_1$ and $M_2$ complete their second transmission, $M_3$ continues its first transmission by holding slots $[19, 20]$. After $M_3$ finishes its first delivery, $M_4$ is allocated

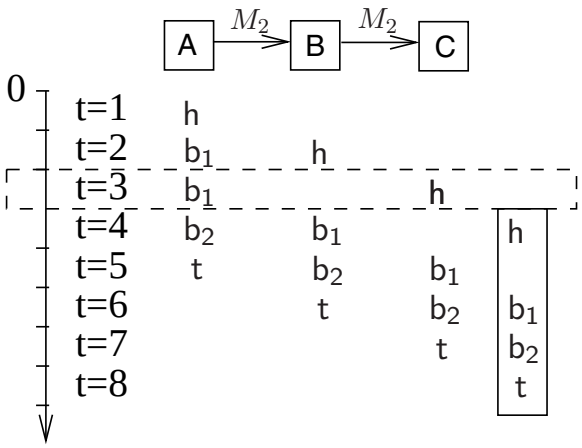



**Figure 3.8.** Avoided flit-delivery scenario

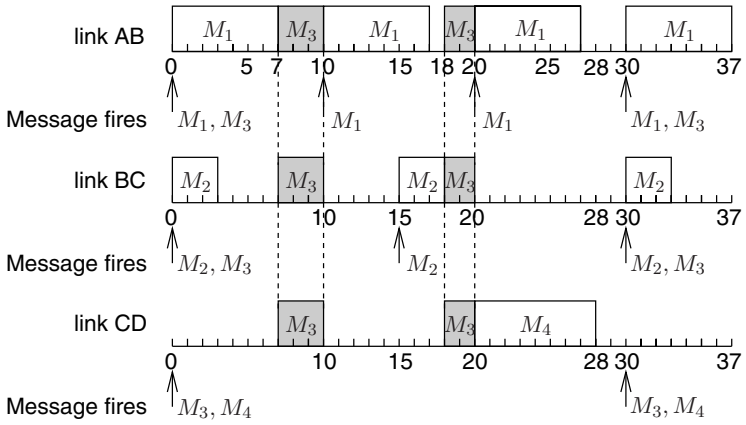
**Table 3.2.** Message parameters and latency bounds

| **Message** | **Period** $p$ | **Deadline** $D$ | **Base latency** $T$ | **Latency bound** $T^{rt}$ |
|---|---|---|---|---|
| $M_1$ | 10 | 10 | 7 | 7 |
| $M_2$ | 15 | 15 | 3 | 3 |
| $M_3$ | 30 | 30 | 5 | 20 |
| $M_4$ | 30 | 30 | 8 | 28 |

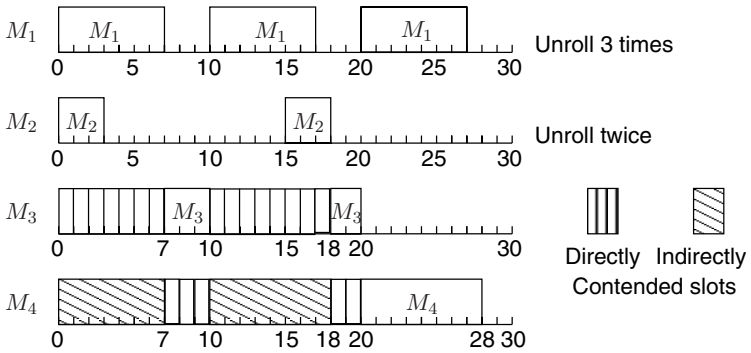
slots $[21, 28]$ on link CD. $M_1$ starts its third round and holds slots $[21, 27]$ on link AB. Since the four messages have a Least Common Multiple (LCM) period of 30, the four messages are scheduled in the same way at each LCM period. From the schedules, we can find that the latency bounds for $M_1$, $M_2$, $M_3$, $M_4$ are 7, 3, 20, 28, respectively. Equivalently, the worst-case blocking times for the four messages are 0, 0, 15, 20. The latency bounds for the four messages are also listed in Table 3.2. We can see that all the four messages are feasible.

Looking into the schedules, we can observe that

(1) $M_1$ and $M_2$ are scheduled in parallel. This concurrency is in fact reflected by the *disjoint* nodes in the tree. We call two nodes *disjoint* if no single walk can pass through both nodes. For instance, $M_1$ and $M_2$ in Fig. 3.6b are disjoint, therefore their schedules do not interfere with each other.




(a) Link schedules of the messages




(b) Global schedules of the messages

**Figure 3.9.** Message scheduling

(2) $M_3$ is scheduled on the overlapped empty time slots [8, 10] and [19, 20] left after scheduling $M_1$ and $M_2$. This is implied in the tree where $M_3$ has two parents, $M_1$ and $M_2$. The *contended slots* [1,7] and [11,18] are occupied by $M_1$ or $M_2$. A contended slot is a time slot occupied by a higher priority message when the contention occurs. A contention occurs only when two competing messages are fired.

(3) $M_4$ is scheduled only after $M_3$ completes transmission at time 20. The indirect contentions from $M_1$ and $M_2$, which are reflected via slots [1,7] and [11,18], *propagate* via its parent node $M_3$. For $M_3$, these slots are directly contended

slots. For $M_4$, they become indirectly contended slots.

The four message schedules are individually depicted in Fig. 3.9b. If the direction contention is not distinguished from the indirect contention as the lumped-link model [7] does, $M_3$ and $M_4$ would be considered infeasible since $M_2$ would occupy the slots [8, 10] and [18, 20], leaving only three slots [28, 30] for $M_3$ and $M_4$. If the concurrent use of the two links, AB by $M_1$ and BC by $M_2$, was not properly captured as the blocking-dependency model [55] does, $M_3$ and $M_4$ would also be considered infeasible since $M_2$ would occupy the slots [8, 10] and [18, 20] before slot 30.

In a contention tree, all levels of indirect contentions propagate via the intermediate node(s). This is pessimistic since many of them are not likely to occur at the same time. Also, a lower priority message can actually use the link bandwidth if a competing message with a higher priority is blocked elsewhere.

The validation of the contention-tree model as well as the comparisons with other proposed contention models are provided in [89].

### 3.3.3 The Feasibility Test

#### A. The feasibility test algorithm

Based on the contention tree and priority-based message scheduling, each feasible message obtains a global schedule. A message schedule is based on its parents' schedules. If a node has no parent or feasible parent, it is scheduled whenever it fires, thus it is always feasible. If a node has feasible parent(s), we must first mark the contended slots as occupied and then schedule the node.

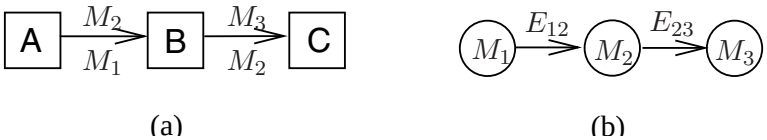



**Figure 3.10.** A three-node contention tree

Note that a slot occupied by a higher priority message is not necessarily a *contended* slot. Consider the contention tree in Figure 3.10 where the three messages use the parameters in Table 3.2. The message schedules are depicted in Figure 3.11. $M_1$ has the highest priority and schedules whenever it fires. Consider the LCM period for the three messages, which is 30 in this case, $M_1$ fires three times and occupies slots [1, 7], [11, 17] and [21, 27]. $M_2$ fires twice at time 0 and 15.

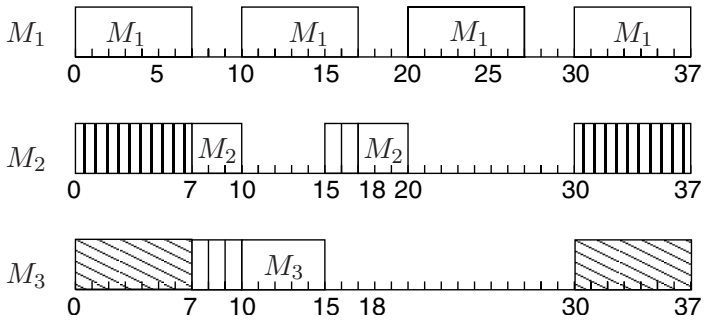



**Figure 3.11.** Message scheduling and contended slots

Although the slots [10, 15] and [21, 27] are occupied by $M_1$, $M_1$ does not contend with $M_2$ during these time slots since $M_2$ is not fired or has already been scheduled. The directly contended slots with $M_1$ are slots [1, 7] and [16, 17], implying that $M_2$ can not be scheduled on these slots. Hence, $M_2$ schedules on slots [8, 10] and [18, 20]. $M_3$ fires once at time 0. The contended slots are [1, 7] (indirectly with $M_1$) and [8, 10] (directly with $M_2$). Hence, $M_3$ is scheduled on slots [11, 15]. In summary, a slot is regarded as a contended slot only if two conditions are true: (1) it is occupied by a higher-priority message; (2) competing messages must fire at the time slot. Particularly, for indirectly contended slots, the intermediate message(s) must also fire in order to pass the contention downwards; otherwise, the slots are not contended. As illustrated in Figure 3.11, for $M_3$, slots [11, 15] are occupied by $M_1$ but not contended slots, since $M_2$ are not fired during these slots. Therefore $M_3$ is scheduled on these slots.

The indirect contentions propagate via parent nodes. Disjoint nodes are scheduled concurrently. If a node $M$ has $k$ feasible parents, $M$ can only be scheduled on the overlapped empty or free (non-contended) slots of the $k$ parents' schedules. The feasibility of a message can be determined by comparing the number $N$ of empty slots available for scheduling $M$ with its non-contentional or base latency $T$. We distinguish messages with a deadline constraint $D$ or a jitter constraint $J$. For a deadline constrained message, its latency bound $T^{rt}$ must satisfy $T^{rt} \leq D$; For a jitter constrained message, its latency bound $T^{rt}$ must satisfy $D - J \leq T^{rt} \leq D$. For a message $M$ with a base latency $T$, we denote that the number of available slots for scheduling $M$ before its jitter range $D - J$ and before its deadline $D$ is $N_J$ and $N_D$, respectively. If $M$ is deadline-constrained and $T \leq N_D$, $M$ is feasible (feasible($M$)=1); otherwise, $M$ is infeasible (feasible($M$)=0). If $M$ is jitter-constrained and $N_J \leq T \leq N_D$, $M$ is feasible; otherwise, $M$ is infeasible.

**Algorithm 1** Contention-Tree-based Feasibility Test for Real-Time Messages

**Input**: A sorted set of $n$ messages and a contention tree for the messages;
**Output**: Feasible$(M_i) = 1/0$, for $i = 1, 2, \cdots n$;

```
1   Find the LCM for the periods of all n messages;
2   For a message M_i, initially i = 1, do {
3     Feasible(M_i) = 0;
4     find M_i's feasible parent(s) F_P;
5     if F_P = φ
6       fire M_i and schedule it to the length of LCM; Feasible(M_i) = 1;
7     else
8       do {
9         fire M_i once;
10        mark M_i's contended slots as occupied and the rest as empty within
        M_i's deadline D_i;
11        compute the length N_Ji and N_Di, which are the overlapped empty slots
        on F_P's schedules within M_i's jitter range D_i − J_i and deadline D_i,
        respectively;
12        if (M_i is jitter-constrained and N_Ji ≤ T_i ≤ N_Di)
        or (M_i is deadline-constrained  and T_i ≤ N_Di), Feasible(M_i)=1;
        and schedule M_i on these free time slots;
13        else Feasible(M_i) = 0; release the scheduled slots for M_i;
14        } while (M_i fires not reaching LCM) and (Feasible(M_i) = 1);
15    i = i + 1;
16  } while (i <= n);
```

We formulate this contention-tree-based feasibility test in Algorithm 1. The input to the algorithm is a sorted set of messages with parameters and constraints, and a contention tree for these messages. The output is the feasibility for each of the $n$ messages, either *pass* (feasible, Feasible$(M_i) = 1$) or *miss* (infeasible, Feasible$(M_i) = 0$). After obtaining the feasible messages, we can further estimate the link utilization of the feasible messages. Finding the LCM of the messages' periods is the necessary and sufficient condition in order to terminate the algorithm since the rest of a feasible schedule can be repeated after the LCM.

### B. The feasibility analysis flow

Using the feasibility test, we can efficiently conduct feasibility analysis by exploring the application-level, partitioning/mapping-stage and architecture-level design

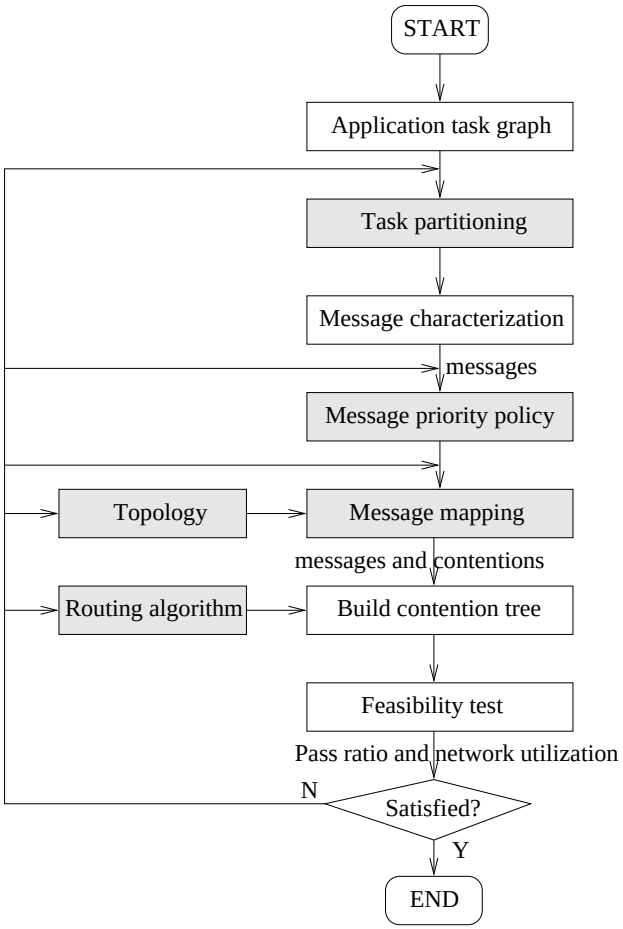



**Figure 3.12.** A feasibility analysis flow

space. Figure 3.12 shows a feasibility analysis flow. First, we partition the tasks and then characterize the messages from the application task graph. Then we build a contention tree. Since the contention tree is affected by several design decisions such as task partitioning, priority policy, message mapping strategy and the routing algorithm etc., we can build different contention trees by exploring these possibilities. After creating a contention tree, the feasibility test algorithm can perform the analysis. The outcome of the test is the pass ratio and network utilization of feasible messages. These two measures may serve as the criteria to calibrate the design decisions. Clearly, this procedure is iterative until satisfaction.

## 3.4 Future Work

NNSE has been demonstrated in the University Booth EDA (Electronic Design Automation) Tool Program of DATE 2005 [73]. After publicity, it has been requested for research use by a number of NoC research groups in Europe, U.S.A. and Asia. In the future, we plan to improve it in the following directions:

- *Parameterize more layers*: Current tunable parameters include topology, routing, and switching schemes. Each of the parameters may be extended with more options. These are all network-layer parameters. In NNSE, the layered structure allows us to orthogonally consider other layers' parameters. In the physical layer, we can build wire, noise and signaling models to examine the reliability and robustness issues. We may consider the link layer parameters such as the link capacity, link-level flow control schemes etc. The upper layer like the transport layer allows us to investigate buffer dimensioning and buffer sharing schemes, as well as end-to-end flow control methods.

- *Configure dependent traffic*: We have so far configured independent traffic, both synthetic and semi-synthetic. This means that traffic from different channels is independent from each other. This is easy to control and generate, but realistic traffic exhibits dependency and correlation. The way to generate traffic with various dependencies such as data, control, time, causality etc. is worth investigating. For example, traffic with the requirement of lip-synchronization shows correlated delivery requirements on video and audio traffic streams.

- *Support Quality-of-Service (QoS)*: This requires the implementation of QoS in the communication platform, and accordingly QoS generators and sinks. Monitoring service may be necessary to collect statistics on whether the performance constraints of a traffic stream have been satisfied or not.

- *Integrate application mapping*: A tool that only explores communication performance is not sufficient. System performance is the result of interactive involvement of both communication and computation. Therefore, supporting application-mapping onto NoC platforms is surely desirable. To this end, we need to build and/or integrate resources models for cores, memories and I/O modules.

- *Incorporate power estimation*: As power is as sensible as performance for a quality SoC/NoC product, NNSE should incorporate the estimation of power

consumption so that the performance and power tradeoffs can be better investigated and understood.

Extending further the traffic generation for performance evaluation ends up with benchmarking different on-chip networks. The diverse NoC proposals necessitate standard sets of NoC benchmarks and associated evaluation methods to fairly compare them.

# Chapter 4

# NoC Communication Refinement

*This chapter presents our NoC communication refinement approach [Paper 8, 9]. We start with a system model specified in the synchronous model of computation. Through a top-down procedure, we refine the communication in the system model into NoC communication via the communication interface of a NoC platform.*

## 4.1 Introduction

### 4.1.1 Electronic System Level (ESL) Design

The rapid advancement of technology constantly fuels the SoC revolution [79]. As we mentioned previously, the state-of-the-art SoC design methodologies cannot sufficiently exploit the abundant transistor capacity. An on-going trend to shrink the productivity gap is Electronic System Level (ESL) design. This trend is mixed with the platform-based design concept [54] and the promotion of using formal models for system specification and verification.

Traditional Register Transfer Level (RTL) for hardware design, which was introduced in the 90s, allows synthesized standard cell design. A synthesizable RTL description is presently often the starting point for an ASIC/FPGA design flow. The design productivity cannot keep pace with the exponential expansion of the number of transistors on a chip. Traditional C-based design for embedded software development shows even slower enhancement in design productivity. To shrink the gap between the design capability and the chip capacity, raising the design abstraction-level is an essential step forward. The current activities in Electronic System Level (ESL) [29] is consistent with this direction. The ITRS [46] defined ESL to be a

level above RTL, that consists of "a behavioral (before hardware/software partitioning) and architectural level (after)". The ESL raises the abstraction level in which systems are expressed. A system-level design allows larger function-architecture co-exploration [63], which is more than traditional hardware-software codesign. The final implementation can benefit in performance and cost. Using system-level models, hardware and software design can be developed in parallel. This breaks the sequential flow of hardware-first-software-second, thus compressing the design cycle. Besides, the benefits of ESL include enabling new levels of design reuse and offering design chain integration across tool flows and abstraction levels. Using formal models is also advocated for system-level design [54, 116, 118]. As noted in [54], using formal models and transformations in system design is promoted so that verification and synthesis can be applied to advantage in the design methodology. Verification, which is a key design activity, is effective if complexity is handled by formalization, abstraction and decomposition. Besides, the concept of synthesis can be applied only if the precise mathematical meaning of a system specification is defined.

A formal model is associated with Models of Computation (MoCs). As defined in [118], a MoC refers to mathematical models that specify the semantics of computation and of concurrency. Loosely defined, MoC specifies the operational semantics governing how processes interact with each other. There are a variety of MoCs that exist for embedded system design, such as finite state machines [39], Petri nets [86], Kahn process networks [51], and synchronous models [11, 12] etc. A comprehensive digest of the various models can be found in [31, 118]. The tagged-signal model [59] defines a denotational, semantic framework of signals and processes within which models of computation can be studied and compared. In [48], a formal classification and description of these models is presented comprehensively. Essentially, how time and concurrency are expressed distinguishes one MoC from another.

### 4.1.2 Communication Refinement

Communication refinement is a key step in a system-level design approach. It is a top-down process of synthesizing abstract communication in the system model into concrete communication in the system implementation architecture [31, 54]. Abstraction defines the type of information present in a model. Unlike hierarchy, abstraction is not concerned with the amount of information visible, but with the semantic principles of a model. In general, the movement from high to low abstraction levels involves a decision-making process. By making design decisions and increasing information about implementation details, we replace more abstract

models with less abstract models, until the system is manufacturable. Through the refinement process, system properties and application constraints must be incorporated and satisfied.

Communication refinement may be conducted in the functional domain or in the implementation domain and usually comprises well-defined steps. A system model, after steps of refinement, is derived into a refined model. The three key issues for refinement are *correctness, constraint and property satisfaction* and *efficiency*. As the refined model is an elaborate version of the original model, they must be functionally equivalent. This is achieved by preserving semantics during refinement, i.e., a refinement step should not introduce semantic deviation. The second requirement means that the refined, correct model must satisfy design constraints for performance and ensure properties to achieve design objectives. The third one here refers to resource consumption in the system implementation architecture. It can be very specific, depending on whether our application is aimed for low power or low cost.

In the NoC case, the communication architecture is preferably predefined as a platform and the Application Level Interface (ALI), which provides primitives for inter-process communication, is the only way to access the communication services. The NoC communication refinement is therefore to refine the abstract communication in a system specification onto the NoC platform via the ALI. We have proposed a three-step top-down procedure to refine the communication of a system model specified in the synchronous MoC into NoC communication. Before we present the refinement steps, we introduce the synchronous MoC.

### 4.1.3 Synchronous Model of Computation (MoC)

#### A. Synchronous modeling paradigm

The synchronous modeling paradigm [11, 12] is based on an elegant and simple mathematical model, which has been shown successful and is the ground of synchronous languages [38] such as Esterel, Signal, Argos and Lustre. The basis is the perfect synchrony hypothesis, i.e., both computation and communication take non-observable time. The critical requirement from specification to implementation is that the implementation has to be *fast enough* both in communication and computation. This means that the implementation phase has to take worst-case into account. Synchronous MoC was initially introduced for reactive and safety-critical embedded control systems where reasoning about the functional correctness is supreme. It has to verify that at each tick over time the system works properly.

In a synchronous MoC, a system is modeled as a set of fully concurrent communicating processes via signals. Processes use ideal data types and assume infinite buffers. By following the tagged-signal model [59], a signal can be defined as a set of ordered events, with each event taking a value and a tag. The value is the informative data to be communicated, and the tag indicates a time slot. This means that each event is conceptually and explicitly accompanied by a time slot to convey data. If the data contains a useful value, the event is $present$ and called a $token$; otherwise, the event is $absent$ and modeled as a $\sqcup$ [1] representing a clock tick. With the introduction of $\sqcup$, multi-rate systems can be modeled since every $n$th event in one signal aligns with the events in another. A synchronous MoC is a timed MoC where events are globally and totally ordered. Each signal can be related to the time slots of another signal in an unambiguous way. The output events of a process occur in the same time slot as the corresponding input events. Moreover, they are instantaneously distributed in the entire system and are available to all other processes in the same slot. Receiving processes in turn consume the events and emit output events again in the same time slot. Processes can thus be viewed as communicating events via an *ideal channel*, which is delay-free. In addition, the ideal channel is buffer-less and has unlimited bandwidth because any type of event values passes through it instantaneously. This communication channel is in contrast to that of other MoCs. For example, the Kahn and dataflow process networks [58] assume unbounded FIFO channels between actors (processes).

Two events are *synchronous* if they have the same tag. Two signals are *synchronous* if each event in one signal synchronous with an event in the other signal and vice versa. A process is *synchronous* if every signal of the process is synchronous with every other signal of the process. A system is *synchronous* if all processes are synchronous locally and globally (synchronous with each other). A system specified in the synchronous paradigm is a synchronous system. For feedback loops, the perfect synchrony leads to cyclic dependency between an input signal and an output signal. If such cyclic communication is allowed in system behavior, some mechanism must be used to resolve it. One possibility is to introduce a delay in the output signal. Another possibility is to use fixed-point semantics, where the system behavior is defined as a set of events that satisfy all processes. The third possibility is to leave the results undefined, resulting in nondeterminism or infinite computation within one tick. If only one precise result is defined for a feedback loop using the delayed time tag, a synchronous model is determin-

[1] In Paper 9, we used $\perp$ (pronounced "bottom") to represent *absent*. Since $\perp$ has been used in dataflow process networks to represent *don't-care* [58], we later used $\sqcup$ in Paper 8 to represent *absent* in order to distinguish it from $\perp$. This notation is also consistent with [48].

istic, i.e., given the same input sequence of events, it generates the same output sequences of events.

### B. The ForSyDe methodology

ForSyDe stands for FORmal SYstem DEsign. It is a system-level design methodology for SoC applications developed in the Royal Institute of Technology, Sweden. The ForSyDe methodology [116] is based on the synchronous MoC. It uses *process constructors* to cleanly separate communication from computation. Communication is captured by the process constructors and computation by the function of the processes. It employs transformations in the functional domain to refine a system model into a less-abstract model optimized for implementation [115, 117]. The transformations, which are conducted step by step, can be either semantic-preserving or a design decision. Semantic-preserving transformations are correct by construction while design decision is not. But, formal verification of design decisions is possible by defining an appropriate notion of equivalence [108]. After refinement, the refined model is partitioned into hardware and software and mapped onto the implementation architecture [70]. In ForSyDe, the zero-delay feedback is forbidden. A delay is introduced in the feedback loop. ForSyDe uses the functional language Haskell [127] to express its system models. The models are executable.

Our refinement approach has been conducted in the ForSyDe framework in order to experiment and validate our concepts with executable models, but our refinement approach applies also to a general synchronous model.

### C. Modeling NoC applications with the synchronous MoC

The reason to start our refinement from a synchronous model is two fold. One is to adopt a formal MoC for the specification of system function. A synchronous model is formal and purely functional. This highest abstraction level leaves the greatest design space to explore, and the advantage of formalism can be used for well-defined refinement, synthesis and verification. The second reason lies in the appropriateness of the synchronous MoC. To model a NoC application, one can ask which MoC is more appropriate? In general the answer depends on which kind of NoC applications to be designed. Considering the strength and weakness of various MoCs, most probably there is no such a one-size-fits-all MoC but different MoCs find their own roles for different applications. However, as the days of cheap communication are gone, expressing communication in a system specification is necessary. A model for a NoC application has to capture communication

properly. Besides, as NoC is a concurrent-processing platform, capturing concurrency in the system model is also necessary. We believe that the synchronous MoC is a good candidate to specify some NoC applications because it captures concurrent computation and communication, and explicitly expresses them in the simplest form possible.

In the following, we first formulate and analyze the communication refinement problem in order to identify the exact sub-problems to be addressed, and then we summarize our solutions presented in Papers 8 and 9.

## 4.2 The Communication Refinement Approach

### 4.2.1 Problem Description and Analysis

Our task is to refine synchronous communication into on-chip communication. Specifically, the problem can be formulated as follows: *Given is a synchronous system model, refine the system communication onto a network-based communication platform. During the refinement, design constraints should be satisfied and the network should be efficiently utilized.*

A synchronous model provides globally synchronous, concurrent and instant communication for inter-process communication. The properties of synchronous communication can be summarized as follows:

- *Global synchrony*: There is logically a global clock triggering the consumption and generation of events. Since computation takes non-observable time, input and output events are distributed synchronously in each and every tick.

- *Instancy*: Events pass via an ideal communication channel. The channel provides zero-delay (instantaneous), unlimited bandwidth, ordered, and lossless delivery. The unlimited bandwidth is due to that a signal can have a value type of any kind, such as any primitive and compound types, demanding an arbitrary communication bandwidth. Consequently, if an output signal of a process is connected to an input signal of another process (as long as they have the same value type), the two signals are identical. Thus we can use a single signal to represent both. Together with the global synchrony, ideal channels maintain a global event order.

- *Full-scale concurrency*: In the topology of a process network, all events are communicated in parallel. Each communication channel is point-to-point

and dedicated. No serialization on the use of the channels is necessary. Full-scale communication concurrency makes the full-scale process computation concurrency possible because input events are available each tick and processes can evaluate the input events and generate output events each tick.

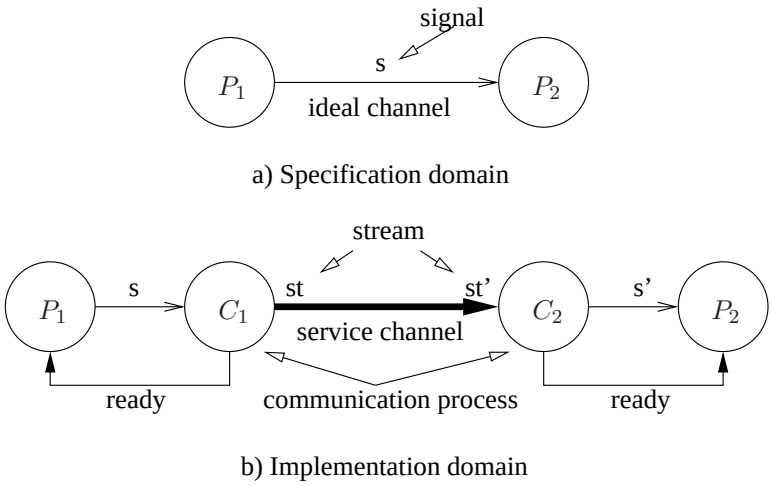



**Figure 4.1.** Computation and communication elements

The computation elements in the specification domain are *processes*, which produce and/or consume events. The communication elements are *signals*, which are ordered sequences of events, and conceptually *ideal channels*. We illustrate these elements with two processes, $P_1$ and $P_2$, in Figure 4.1a.

In the implementation domain, NoC has a very different communication model and associated properties. We consider a message-passing NoC platform where each core has its own local memory. As we discussed in Chapter 1, NoC communication can be represented using the layered and interfaced model. The inter-process communication is offered by the Application-Level Interface (ALI). The application processes use communication primitives such as *open_channel(), read()* and *write()* to communicate messages. Logically we can view that process-to-process communication is conducted through a dedicated, point-to-point *service channel*. The service is mapped directly to an underlying network communication service. We simplify the consideration of the session/transport layer, which performs packetization/de-packetization, interleaving for using shared buffers and bandwidth, and re-ordering for maintaining the message causality, if necessary. We assume that the net effect of the session/transport layer is the addition of delay

and in-order message delivery. This delay contributes to the delay in the service channel model.

The service-channel communication model differs drastically from the synchronous communication model.

- *Multiple clock domain communication*: There is no a global clock triggering system computation and communication. Instead the cores and the network reside in different clock domains. We assume that the network itself is clocked by a single clock, which has the same phase as the core clocks. The core frequencies can be different from each other. The communication behavior of the cores and network can be modeled in their own clock domains following the synchronous model. But the cross-domain time structures must be arbitrated.

- *Bandwidth-limited and delay-variant channel*: Although we can abstract the inter-process communication as logically a point-to-point service channel at the application layer, these channels share physical communication resources such as buffers and links in the session/transport and network layers. The service channel has a capacity limitation and in general introduces delay and delay variation (jitter). It provides in-order message delivery within a service channel, but there is no message ordering between service channels.

- *Conditional concurrency*: As a service channel is bandwidth-limited, it is impossible to send and receive arbitrary amount of data (any kind of data structure) within a fixed-length time window. The communication concurrency is restricted by available bandwidth. This limitation leads to conditional computation concurrency, i.e., computation concurrency is communication dependent.

In the implementation domain, we must introduce a communication process in order to glue a signal to a service channel. The communication elements in the implementation domain are *communication processes*, *streams* and *service channels*. Streams are ordered sequences of messages. The elementary communication processes either generate messages by consuming events or produce events by consuming messages. Service channels are where the streams are transported. The computation processes must be stallable if the required input tokens for computation are not available. We illustrate the three communication elements with the two computation processes, $P_1$ and $P_2$, in Figure 4.1b.

As we can observe from the above analysis, the ideal communication in the synchronous model does not exist at all in the implementation domain. The immediate questions to answer while refining the synchronous communication into NoC communication are:

- How to compromise global synchrony into multiple-clock synchrony?
- How to refine ideal communication into shared communication?
- How to refine fully concurrent computation and communication into conditionally concurrent computation and communication?
- How to satisfy performance constraints and communication properties?
- How to make a good utilization of network resources during the refinement?

These questions may not be addressed in isolation because they are inherently correlated. For example, refining ideal communication into shared communication results in reducing the concurrency level in the system model. Throughout the communication refinement, the system behavior can not be changed. Maintaining the correct system behavior during the refinement is the first concern because of the violation of the ideal communication assumption in the implementation domain. This task is burdened with the requirements of satisfying constraints such as processing $n$ samples/second and of not over-dimensioning the underlying network.

### 4.2.2 Refinement Overview

In Papers 8 and 9, we have proposed a three-step approach for the NoC communication refinement problem. Paper 8 mainly focuses on refinement for correctness and Paper 9 for performance and efficiency. In this section, we unify the concepts presented in the two papers in order to give a coherent view of the proposed communication refinement approach.

Our refinement approach consists of three steps, namely, *channel refinement*, *process refinement* and *communication mapping*, as illustrated in Figure 4.2.

Step 1. *Channel refinement* (Section 4.2.3): An ideal channel is refined into a service channel. The service channel models the characteristics of the underlying network communication service. We also model the interfaces between different clock domains assuming that the network resides in a clock domain different from cores.

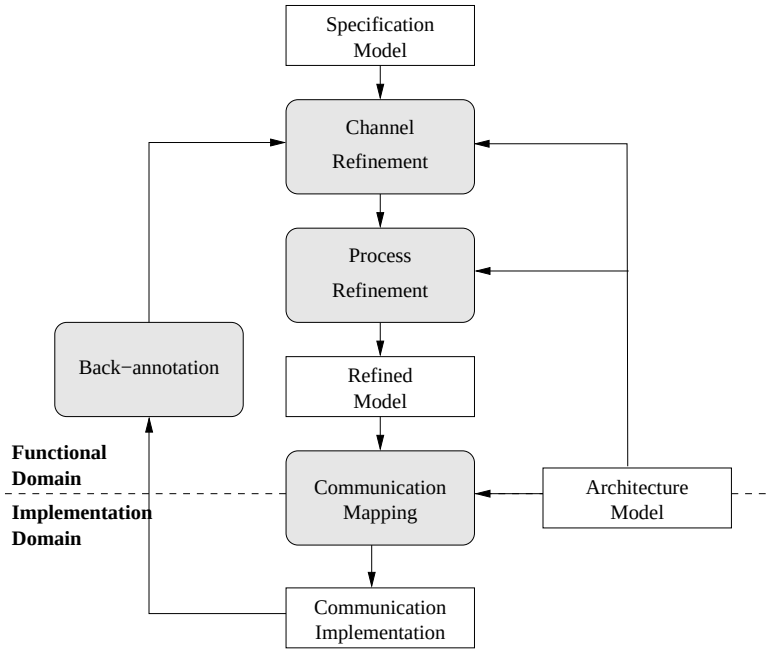



**Figure 4.2.** NoC communication refinement

Step 2. *Process refinement* (Section 4.2.4): In the specification model, a process may be viewed as only performing functionality since communication is ideal (unlimited bandwidth and zero-delay). This will not be the case once the ideal channel is replaced by a service channel. To reuse the original computational process, we leave it untouched. But we need to encapsulate the process with a *communication process*. This communication process (a) interfaces with the service channel; (b) fulfills the computation synchronization requirement of the process, which we call *process synchronization property*. This synchronization property must be consistent during the refinement. With the introduction of the service channel, the process cannot fire automatically with the clock. Instead, it requires additionally a *synchronization ready* signal from the communication process to control its execution; (c) satisfies communication property and performance constraints by refining communication protocols, for example, performing end-to-end flow control for reliability and overlapping computation with communication to hide the communication latency; (d) deals with feedback loops, if any. A zero-delay feedback loop is resolved by introducing an initial delay in the

loop to break the cyclic dependency in our specification model. However, in implementation, it results in excessive delay when the feedback is looped through service channels. If the process sticks to this synchronization point, the feedback loop becomes a serious performance bottleneck.

Step 3. *Communication mapping* (Section 4.2.5): After the above two steps, we obtain a refined model. To further optimize for the use of shared network resources, two or multiple service channels (a) may be converged to share one implementation channel and (b) may be merged into one service channel so as to use one implementation channel. After the optimization, we move from the functional domain to the implementation domain. With a process-to-core mapping plan, we map the service channels and the communication processes in the refined and optimized model onto the NoC platform.

In Step 1, ideal channels are replaced by service channels, which involve multiple clock-domain communication. We use communication processes in Step 2a and Step 2b to refine the ideal communication into shared communication and to refine full-scale concurrency into conditional concurrency. Particularly, we have focused on Step 2b of synchronization consistency, which is proposed for correctness. Step 2c aims to satisfy communication property and performance requirements by communication protocol refinement. Step 2d deals with the feedback problem. It aims also to enhance system performance. In Step 3a and Step 3b, we consider channel-convergence and channel-merge to make efficient use of network resources. In summary, we have taken into account correctness, performance as well as resource utilization during the refinement.

### 4.2.3 Channel Refinement

#### A. The clock-domain interface

A synchronous model is very simplified in the sense that a single clock drives the system computation and communication. We assume that NoC communication is partitioned into multiple clock domains. While each clock domain is synchronous, the time structures of cross-domain communications have to be correctly arbitrated. Our assumption is that the cores and the network have their own clock domains. In addition, we assume that all clock phases are aligned.

To refine a single clock domain into multiple clock domains, we introduce clock sub-domains into the system's main domain. Each sub-domain is modeled synchronously, and a clock domain interface arbitrates the time structures of different clock domains. Introducing a synchronous sub-domain into the system model

was presented in [115] where the event rate of the sub-domain is $\frac{1}{n}$ ($n$ is a positive integer) of the main domain. The main domain is interfaced to the sub-domain by a single *down-sampling* process $P_{dn}(n)$. The sub-domain is interfaced back to the main domain by a single *up-sampling* process $P_{up}(n)$. We extend this work by considering a *generic domain interface* that connects a clock domain with event rate $f_1$ to another clock domain with event rate $f_2$. The simplest form of the fraction $\frac{f_1}{f_2}$ is $\frac{m}{n}$, where $m$ and $n$ are coprime.

The generic interface from domain $f_1$ to domain $f_2$ is constructed by using two processes as $I_{f_1 \to f_2} = P_{dn}(m) \circ P_{up}(n)$, where $\circ$ is the composition operator. The processes, $P_{up}(n)$ and $P_{dn}(m)$, are formally defined as follows:

$$P_{up}(n)(\{x_1, x_2, \dots\}) = \{\underbrace{\perp, \dots, \perp}_{n-1}, x_1, \underbrace{\perp, \dots, \perp}_{n-1}, x_2, \dots\}$$

$$P_{dn}(m)(\{\underbrace{x_1, x_2, \dots, x_m}_{m}, \underbrace{x_{m+1}, \perp, \dots, \perp}_{m}, \dots\}) = \{x_m, x_{m+1}, \dots\}$$

The *up-sampling* process $P_{up}(n)$ samples out $n$ times of the input events, and does not result in event loss. The *down-sampling* process $P_{dn}(m)$ samples out $\frac{1}{m}$ times of the input events. At each down-sampling cycle, $m-1$ events are discarded and only the last token (non-absent value) is kept. The interface first does up-sampling and then down-sampling. If $f_1 \leq f_2$, no event drops, hence no token is lost. If $f_1 > f_2$, events are cyclically dropped. But this may or may not lead to the loss of tokens because the token rate may be less than the event rate. To guarantee that there is no data loss at the clock domain interface, the token rate of domain $f_1$ can not be faster than the event rate of domain $f_2$. This is to say, that a producer in domain $f_1$ can not use bandwidth (by generating tokens) more than the consumer domain's capacity. In our further analysis, we assume that this condition is satisfied and there is no data loss at the clock domain interfaces.

### B. The service channel model

As we mentioned previously in the analysis, we consider a generic service channel that provides inter-process communication using message passing. In our context, we are interested in that different processes are distributed in different cores. Thus, an inter-process communication corresponds to an inter-core communication. Such kind of communication involves the session/transport layer and the network layer. As having discussed previously, we focus on the network services and simplify the session/transport layer effects.

A service channel is logically a simplex point-to-point channel, offering in-order, lossless and bounded-in-time communication between two end-processes. A

service channel is mapped to a communication service in the underlying network. A basic distinction of network services is the guaranteed service and best-effort service. The guaranteed service requires the establishment of a virtual circuit before data transmission can start. Once a virtual circuit is set up, the message delivery is bounded in time. The best-effort service delivers messages as fast as possible. As long as the network does not drop packets and is free from deadlock and livelock, the delivery completion property is honored. Since no resources are pre-allocated, there is no guarantee on a delivery bound in general. This nondeterminism is due to that message delivery experiences dynamic contentions in the network. However, if such a bound does not exist, further analysis may be meaningless since the system performance becomes unpredictable. Therefore we assume that the best-effort service provides a delivery bound, but with an additional condition. The condition can be that the processes (the network clients) and the network interact on a contract basis. Processes inject traffic into the network in a controlled manner according to a traffic specification. Such a traffic specification may conform to, for example, the regulated $(\sigma, \rho)$ flow model [22, 23]. The network performs a disciplined arbitration on resource sharing. In this way, the network saturation is avoided and the delivery bound can be derived. But, in our current analysis, this regulated traffic admission as well as traffic discipline has not been modeled. Instead we have assumed that such a scheme exists and even the best-effort network service can guarantee the bounded-in-time delivery.

With this assumption, we concentrate on considering the net effect of message delivery, i.e., the delay and its variation (jitter) by resorting to a stochastic approach. Formally, we develop a unicast service channel as a point-to-point *stochastic* channel: given an input stream of messages $\{m_1, m_2, \cdots, m_n\}$ to the service channel, the output stream is $\{d_1, m_1, d_2, m_2, \cdots, d_n, m_n\}$, where $d_i$ denotes the delay of $m_i$ ($i \in [1, n]$), which may be expressed as the number of absent ($\sqcup$) values and is subject to a distribution with a minimum $d_{min}$ and maximum $d_{max}$ value. The actual distribution, which may differ from channel to channel, is irrelevant here. We do not make any further assumptions about this. If $d_i = k$ ($k$ is a positive integer), it means that there are $k$ absent values between $m_{i-1}$ and $m_i$. We can identify two important properties of the generic service channel: (1) $d_i$ may be varying; (2) $d_i$ is bounded. This behavior is purely viewed from the perspective of application processes and the implementation details are hidden.

Together with clock-domain interfaces, a service channel provides transparent communication for processes in different clock domains. Since the effect of clock-domain interfaces can be modeled by the delay distribution in a service channel, we do not explicitly consider them further.

### 4.2.4 Process Refinement

#### A. Interfacing with a service channel

Once an ideal channel is replaced by a service channel, the original process can not be directly connected to the service channel because a service channel uses a different data unit, *message*, and has limited bandwidth. A communication process must be introduced as an interface to connect the original process with the service channel. This communication process implements necessary data conversion and handshake-like control functionality, detailed as follows:

- *Data conversion*: The input/output data type of a service channel is a message that is of a bounded size. But a signal in the specification assumes an ideal data type, whose length is finite but arbitrary, e. g., a 32/64-bit integer, a 64-bit floating point or a user-defined 512-bit record type. Matching the data types requires data conversion, such as decomposition and composition.
- *Bandwidth-regulated control*: The service channel has limited bandwidth while a signal uses unlimited resources. The sending and receiving of messages using the service channel is subject to the available bandwidth. A control function is needed in the communication process to co-ordinate the message sending and receiving.

These adaptations are achieved by writer and reader processes. Specifically, to interface with the service channel, a producer needs to be wrapped with a *writer*, a consumer with a *reader*. As shown in Figure 4.1, $P_1$ is a producer and $P_2$ a consumer. $C_1$ implements the *writer* function and $C_2$ the *reader* function.

#### B. Synchronization consistency

Replacing the ideal channel (zero delay and unlimited bandwidth) with a stochastic channel (varying delay and limited bandwidth) leads to the violation of the synchrony hypothesis. Consequently, two synchronous events in the specification model may not have the same time tag any more because they may experience different delays in the service channels. Two synchronous signals in the specification model may no longer be synchronous. Furthermore, the synchronous system becomes globally asynchronous. This leads to possible behavior deviation from the specification. The entire system may not function properly. Correctness becomes the first issue to address in refinement.

We restrict our discussions to *continuous processes* [48]. Informally, we say a process *continuous* if, given partial input events, it generates partial output events.

In addition, adding more input events, more output events are generated but it will not affect the previously generated results. For a continuous process to work correctly, two conditions for delivering its input signals must be satisfied: (1) the event order of each signal must be maintained; (2) the synchronization requirement on the input events, called *process synchronization property*, must be satisfied before the process can fire. In the synchronous model, events are delivered in order and fully concurrent, the two conditions are satisfied cycle by cycle. However, with the NoC service channel model, the first condition is met but the second is not guaranteed. Our objective is to satisfy the process synchronization property, i.e., to maintain *synchronization consistency*. Our approach is to refine the system-level global synchronization into process-level local synchronization. We first classify the process synchronization properties and then use synchronizers to achieve synchronization consistency during refinement.

The synchronization in the system model requires all signals are synchronous and all processes are synchronous. This might over-specify the whole system, limiting implementation alternatives. We derive the synchronization property of a process according to its *evaluation conditions*. This is similar to firing rules that are used to discuss dataflow processes in [58]. By using evaluation conditions, we are able to decouple local computation synchrony from global computation synchrony. In effect, this refines the computation concurrency in the system model from being fully concurrent into being conditionally concurrent.

For a synchronous process with $n$ input signals, $PI$ is a set of $N$ input patterns, $PI = \{I_1, I_2, \cdots, I_N\}$. The input patterns of a synchronous process describe its *firing rules*, which give the conditions of evaluating input events at each event cycle. $I_i$ ($i \in [1, N]$) constitutes a set of event patterns, one for each of $n$ input signals, $I_i = \{I_{i,1}, I_{i,2}, \cdots, I_{i,n}\}$. A pattern $I_{i,j}$ contains only one element that can be either a token wildcard $*$ or an absent value $\sqcup$, where $*$ does not include $\sqcup$. Based on the definition of firing rules, we define four levels of process synchronization properties as follows:

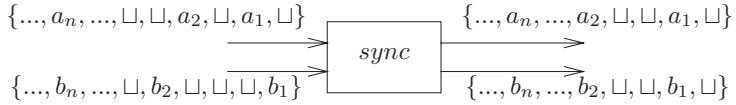
- *Strict synchronization*: All the input events of a process must be present before the process evaluates and consumes them. The only rule that the process can fire is $PI = \{I_1\}$ where $I_1 = \{[*], [*], \cdots, [*]\}$.

- *Nonstrict synchronization*: Not all the input events of a process are absent before the process can fire. The process can *not* fire with the pattern $I = \{[\sqcup], [\sqcup], \cdots, [\sqcup]\}$. This also includes cases where the process can not fire if one or more particular input events are $\sqcup$.

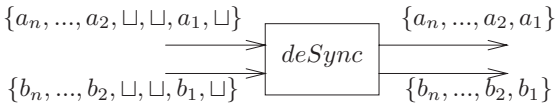
- *Strong synchronization*: All the input events of a process must be either present or absent in order to fire the process. The process has only two firing rules $PI = \{I_1, I_2\}$, where $I_1 = \{[*], [*], \cdots, [*]\}$ and $I_2 = \{[\sqcup], [\sqcup], \cdots, [\sqcup]\}$.

- *Weak synchronization*: The process can fire with any possible input patterns. For a 2-input process, its firing rules are $PI = \{I_1, I_2, I_3, I_4\}$ where $I_1 = \{[*], [*]\}$, $I_2 = \{[\sqcup], [\sqcup]\}$, $I_3 = \{[*], [\sqcup]\}$ and $I_4 = \{[\sqcup], [*]\}$.

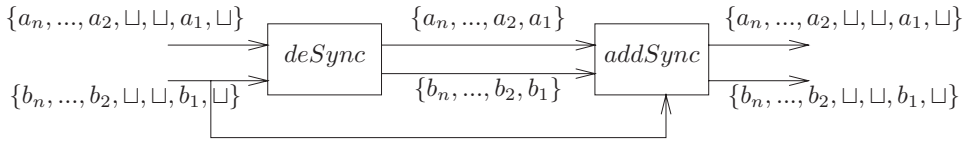
Apparently, for processes with a strict or strong synchronization, their synchronization properties can not be satisfied if any of their input signals passes through a service channel since the delays via the channel are stochastic. Although globally asynchronous, the processes can be locally synchronized by using synchronization processes, called *synchronizers*, to satisfy their synchronization properties.




a) An align-synchronization process




b) A de-synchronization process




c) An add-synchronization process

**Figure 4.3.** Processes for synchronization

We use a two-input process to illustrate these synchronizers in Figure 4.3. In the figure, we follow the direction of the signals and place the earlier events in the right side of a signal, i.e., $\{\cdots, x_n, \cdots, x_2, x_1\}$. An align-synchronization process *sync* aligns the tokens of its input events, as illustrated in Figure 4.3a. It does not change the time structure of the input signals. A desynchronizer *deSync* removes the absent values, as shown in Figure 4.3b. All its input signals must have the same token pattern, resembling the output signals of the *sync* process.

Removing absent values implies that the process is *stalled*. The desynchronizer changes the timing structure of the input signals, which must be recovered in order to prevent from causing unexpected behavior of other processes that use the timing information. An add-synchronizer *addSync* adds the absent values to recover the timing structure, as shown in Figure 4.3c. It must be used in relation to a *deSync* process. If the input events of the *deSync* is a token, the *addSync* reads one token from its internal buffers for each output signal; otherwise, it outputs a $\sqcup$ event. The two processes *deSync* and *addSync* are used as a pair to assist processes to fulfill strictness.

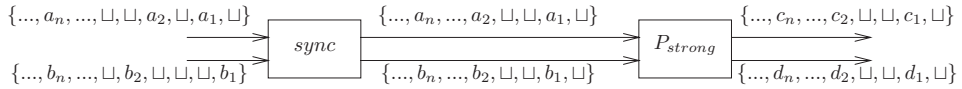



**Figure 4.4.** Wrap a strong process

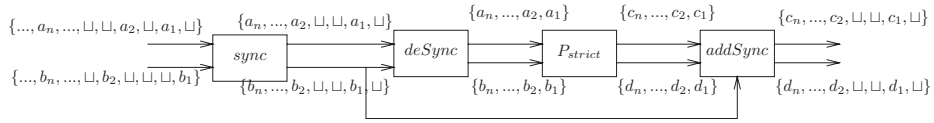



**Figure 4.5.** Wrap a strict process

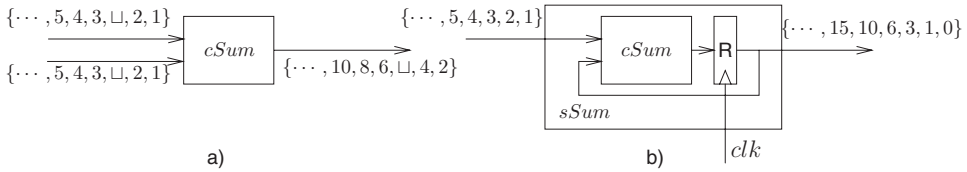



**Figure 4.6.** A strong and a strict process

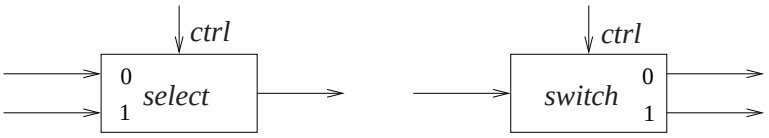



**Figure 4.7.** Two non-strict processes

To achieve strong synchronization, we use an align-synchronization process *sync* to wrap a strong process, as shown in Figure 4.4. To achieve strict synchronization, we use three processes, *sync*, *deSync* and *addSync*, to wrap a strict process,

as illustrated in Figure 4.5. A strong process is typically a *combinational* process, which is state-less. As long as both input tokens and delays are aligned, the delays on the input signals do not change the behavior of the process. For example, a combinational sum process $cSum$ in Figure 4.6a consumes two input events, one from each signal, adding them together. Delays on both input signals are tolerable as long as they are aligned. A strict process is typically a *sequential* process, which has states and thus is sensitive to the delay on its input signals. For instance, the sequential process $sSum$ in Figure 4.6b calculates the running sum of its input events by adding its state and the token value on the input signal. Its initial state is 0. Any delay on its input signal changes the output sequence. A non-strict process is often a control process, which can not fire if a control token is not available. For example, as shown in Figure 4.7, the *select* and *switch* processes can not fire if the control signal *ctrl* is neither 0 nor 1. Feeding control tokens is particularly important while refining non-strict processes. The refinement of processes with weak synchronization should be individually investigated. Practical examples of using synchronizers are given in Paper 8.

### C. Protocol refinement

Message passing between a producer process and a consumer process is essentially a process of moving data from the producer side buffer to the consumer side buffer. This requires a co-ordination between the producer process and the consumer process in order to guarantee some properties, such as reliability, completion and buffer-overflow freedom. As the communication via a service channel introduces variable delay, it is important to overlap computation with communication in order to hide the communication latency. Protocol refinement is to refine the communication protocol for various reasons, for example, coordinated and improved communication. We have shown in Paper 9 that our refinement approach can formally incorporate different communication protocols in the step of process refinement to satisfy reliability and to improve throughput. For reliability, we have introduced acknowledgment in the protocol. For throughput, we have shown that data pipelines may be elaborated to hide communication latency and thus increase concurrency in computation and communication.

### D. Feedback loop

Figure 4.8a illustrates a feedback loop in the specification. The loop passes through $n$ processes, $P_1, P_2, \cdots, P_n$. In the synchronous model, we insert a single *Delay*

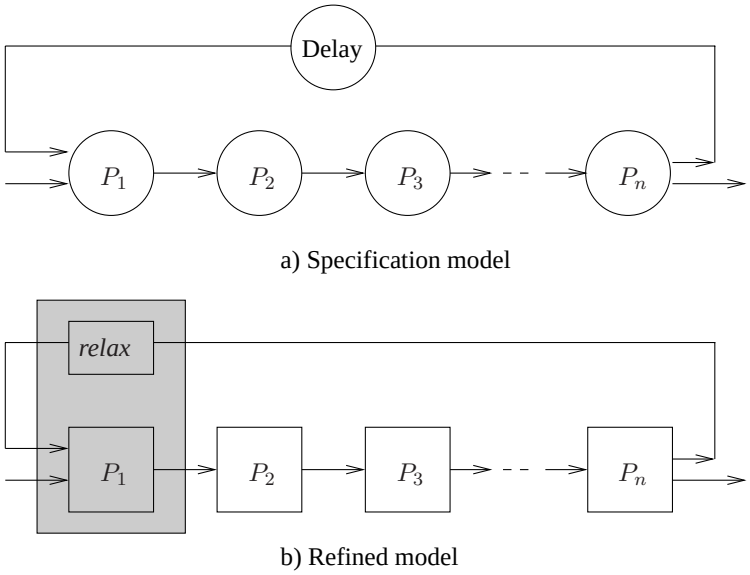



**Figure 4.8.** Feedback loop

process (can be viewed as a register) to break the zero-delay loop. In an implementation domain as shown in Figure 4.8b, even if one process has one cycle delay, it will take $n$ cycles for the feedback signal to loop back. If process $P_1$ sticks to this synchronization point, the system throughput can never be faster than $1/n$ samples/cycle. A similar and even worse situation occurs in the NoC refinement case. If a feedback signal passes through a best-effort service channel, the delays are nondeterministic. Depending on the length of such a loop, the varying delays may be very long. If strictly observing the dependency, the process has to wait for the availability of the feedback events and cannot fire. The entire system is slowed down and becomes unpredictable due to the feedback loop.

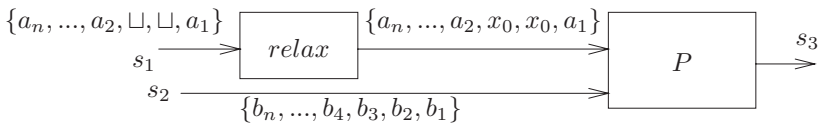



**Figure 4.9.** A relax-synchronization process

In our current proposal, we have used a *relaxed synchronization* method to force the synchronization satisfied. The synchronizer for this purpose is the relax-synchronization process *relax*, as illustrated in Figure 4.9. If the input event is a

token, it outputs the token; otherwise, a token $x_0$ is emitted. The exact value of $x_0$ is application dependent. Note that relaxing synchronization is a design decision leading to behavior discrepancy between the specification model and the refined model. Care must be taken to validate the resulting system.

An example of using *relax* is given in Paper 8 where an equalizer regulates the volume of an input audio stream to protect the speaker by preventing the audio bass from exceeding a threshold. It analyzes the power spectrum of the resulting output audio stream after the regulation. In case a certain level is reached, it feeds a control signal back to adjust the amplification level of the audio bass amplifier. In the specification, the equalizer has an immediate response whenever surpassing the threshold occurs. However, after mapping the signals to best-effort service channels, the feedback signal takes long and nondeterminate delays to reach the amplifier. If the amplifier sticks to this over-specified synchronization point by following exactly the specification model, the equalizer may not be able to process enough audio samples per second. The system performance might become unacceptable. In this case, a *relax* is inserted to generate an amplification level when a token is not available. The equalizer can thus fulfill the requirement on throughput. The side-effect of this refinement is that the response to the audio-volume control buttons is delayed by some cycles. We can validate that the small amount of delay is acceptable for users. Therefore this design-decision is justified.

### 4.2.5 Communication Mapping

#### A. Channel convergence and channel merge

In the mapping phase, a refined channel, i.e., a service channel, is to be mapped to an implementation channel. A simple way of doing this is to map one service channel to one implementation channel. This one-to-one channel mapping may lead to inefficient use of network resources. For example, considering a guaranteed service using TDM virtual-circuits, for low-delay low-bandwidth traffic, bandwidth has to be reserved to satisfy the low-delay requirement. This results in extra time slots reserved but under-utilized. Therefore, an implementation channel should allow multiplexing, i.e., shared by more than one service channel, if possible. Considering these effects, the refined model can be further optimized for efficiency. In Paper 9, we have introduced *channel convergence* and *channel merge*. Channel convergence is for two or multiple service channels to share one implementation channel provided the implementation channel can support their total bandwidth requirement. Channel merge means that two or multiple service channels may be

merged into one service channel so as to use one implementation channel by packing messages at the sender side and splitting them at the receiver side.

**B. Channel and communication process mapping**

The inputs to this task are the refined and optimized model as well as a process-to-core allocation scheme; the output is a communication implementation. We have used the NoC simulator *Semla* [125] as our implementation platform. The Application-Level Interface (ALI) is the set of message-passing primitives introduced in Chapter 3. The mapping stage involves *channel mapping* and *communication process mapping*.

With channel mapping, each pair of processes communicating via a service channel in the refined model results in its dedicated unicast implementation channel, which is mapped to the open channel primitive *open_channel()*.

In communication-process mapping, communication processes for interfacing service channels (*reader* and *writer*), maintaining synchronization consistency (synchronizers such as *sync*, *deSync*, *addSync* and *relax*), elaborating protocol and optimizing resource usage, are mapped onto cores. The original computation processes do not change, but their executions are controlled by their respective communication processes. Besides, with a single-thread implementation on a core, a static schedule has to be found to sequentialize the process executions and coordinate write and read operations [70]. With a multi-thread implementation on a core, processes may be dynamically fired when their synchronization requirements are met according to their evaluation conditions [121]. The reader and writer processes access the ALI by directly calling the communication primitives *read()* and *write()* defined in the NoC simulator. The resulting implementation is executable in the simulation framework.

## 4.3 Future Work

Our refinement proposal has sketched a way of refining synchronous communication into on-chip network-based communication. Through the refinement, correctness, constraint and efficiency have been taken into account. This approach has been validated in the ForSyDe framework and with our NoC simulator. The concept of synchronization consistency is independent of a particular communication implementation scheme. It can be applied to pure hardware, software and bus-based mixed hardware/software architectures. The proposed synchronizers have been implemented in hardware, software and mixed hardware/software [121].

To make our refinement approach fully-fledged, we realize that the research can be further carried on along the following three tracks:

- *Embed formal semantics in the refinement approach*: This is to give formal definitions for the synchronization issues and develop transformation rules for using synchronizers to represent and check the equivalence between the specification model and the refined model.
- *Conduct performance analysis and optimization*: The refined model allows us to derive performance figures. This requires that a stochastic process must be annotated with good-enough values. So far we just consider the effect of varying delay of the stochastic process. How to estimate the delay/jitter values and further system performance analysis are not addressed yet. Particularly, we will treat feedback using TDM virtual-circuits [34, 81] in order to obtain strong guarantees on delay and jitter bounds.
- *Automate the design flow*: With the well-defined synchronizers and well-controlled use of the synchronizers, automation is possible. Currently a process's synchronization property is annotated manually. But this can be done automatically. The reason is that, in the system model, a process can be defined using *pattern matching* evaluation [116], which nicely matches the process synchronization property. We are building synchronizer libraries in hardware, software and mixed hardware/software, and plan to develop programs that can automatically instantiate synchronizers for processes to maintain synchronization consistency. Optimization for performance and efficiency will be part of the automation.

# Chapter 5

# Summary

*This chapter summarizes the thesis and outlines future directions.*

## 5.1 Subject Summary

Moore's law has sustained in the semiconductor industry for 42 years. Following this law, the process technology has been ever-advancing. Meanwhile, the desire to exploit the technology capacity is ever-aggressive. However, the advancement of the chip capacity and the system integration capability is not evenly developed. The slower development of SoC integration is due to the extremely high level of complexity in system modeling, design, implementation and verification. As communication becomes a crucial issue, NoC is advocated as a systematic approach to address the challenges. NoC problems span the whole SoC spectrum in all domains at all levels. This thesis has focused on *on-chip network architectures*, *NoC network performance analysis*, and *NoC communication refinement*.

- Research on wormhole-switched networks has traditionally emphasized the flit delivery phase while simplifying flit admission and ejection. We have initiated investigation of these issues. It turns out that different flit-admission and flit-ejection models have quite different impact on cost, performance and power. In a classical wormhole switch architecture, we propose the coupled flit-admission and $p$-sink flit-ejection models. These optimizations are simple but effective. The coupled admission significantly reduces the crossbar complexity. Since the crossbar consumes a large portion of power in the switch, this adjustment is beneficial in both cost and power. The network performance, however, is not sensible to the adjustment before the network

reaches the saturation point. The $p$-sink model has a direct impact on decreasing buffering cost, and has negligible impact on performance before network saturation. As the support for one-to-many communication is necessary, we design a multicasting protocol and implement it in a wormhole-switched network. This multicast service is connection-oriented and QoS-aware. For the TDM virtual-circuit configuration, we utilize the generalized logical-network concept and develop theorems to guide the construction of contention-free virtual circuits. Moreover, we employ a back-tracking algorithm to explore the path diversity and systematically search for feasible configurations.

- On-chip networks expose a much larger design space to explore when compared with buses. The existence of a lot of design considerations at different layers leads to making design decisions difficult. As a consequence, it is desirable to explore these alternatives and to evaluate the resulting networks extensively. We have proposed traffic representation methods to configure various workload patterns. Together with the choices of the traffic configuration parameters, the exploration of the network design space can be conducted in our network simulation environment. We have suggested a contention-tree model which can be used to approximate network contentions. Using this model and its associated scheduling method, we develop a feasibility analysis test in which the satisfaction of timing constraints for real-time messages can be evaluated through an estimation program.

- As communication is taking the central role in a design flow, how to refine an abstract communication model onto on-chip network-based communication platform is an open problem. Starting from a synchronous specification, we have formulated the problem and proposed a refinement approach. This refinement is oriented for correctness, performance and resource usage. Correct-by-construction is achieved by maintaining synchronization consistency. We have also integrated the refinement of communication protocols in our approach, thus satisfying performance requirements. By composing and merging communication tasks of processes to share the underlying implementation channels, the network utilization can be improved.

## 5.2 Future Directions

With only a short history, Network-on-Chip (NoC) has become a very active research field. Looking into the future, we believe that NoC will continue to be vivid.

We list some key issues that have not been sufficiently addressed or emphasized in the community as follows:

- *Heterogeneous modeling*: In current SoC design flows, a number of modeling techniques have been used ranging from sequential to concurrent models, from untimed to timed models. Application complexity and heterogeneity have driven the need to model a system using heterogeneous models. The Ptolemy project [60] is such an example. This is particularly true for NoC since it also targets highly complex and heterogeneous applications. To which extent to model the underlying architecture characteristics is one issue. While a model itself does not necessarily reflect the detailed characteristics, refinement may be facilitated if the architecture characteristics such as concurrency, time and adaptivity are captured in the model properly. Since there does not exist a one-size-fits-all Model-of-Computation (MoC), multi-MoC modeling will be highly necessary. Based on our understanding on the various MoCs, one challenge is the cross-MoC-domain modeling, i.e., from untimed domain to timed domain, from discrete time to continuous time, from a sequential model to a concurrent model, and vice versa. The follow-up challenges include multi-MoC refinement, synthesis, and verification.

- *Programmability*: To reduce cost, making a NoC *soft* is essential. This requires the support of operating systems that offer various services such as I/O handling, memory management, system monitoring, process scheduling and migration, and inter-process communication, and provide programming models balancing ease-of-programming and efficiency. Efficient application-level interfaces and standardized core-level interfaces are the *hard* part. As NoC is a distributed (not centralized) system in nature, investigating parallel computing models beyond von Neumann models for NoC systems to achieve high performance will become *hot*. For example, the MultiFlex system [101] supports an object-oriented message passing model.

- *Composability*: To build complex systems, we are moving away from creating individual components from scratch towards methodologies that emphasize composition of re-usable components via the network paradigm. NoC systems should allow one to plug new validated components and upgrade old components with linear design efforts and without compromising performance, reliability and verifiability. This feature makes a NoC easy-to-integrate and easy-to-extend, leveraging the reuse to the system level and shrinking the time-to-market.

- *Autonomy*: There are several reasons to hope for an autonomous NoC. A nano-chip is an extremely condensed device where transient and permanent faults on wires and nodes are increasingly possible. Power consumption is workload-dependent and performance-sensible. System optimization involves the re-organization and orchestration of its computation and communication components to tradeoff power and performance and to balance the thermal distribution on the chip. These reliability, performance, power and thermal issues call for an intelligent way like human self-healing, self-vaccinating and self-adjusting systems to dynamically and autonomously adapt the NoC to suit its application demands and operating environments. Along this thread, a simulation tool may be aimed to be *intelligent* in, for example, pinpointing performance bottlenecks and suggesting hints on buffer dimensioning.

- *Design flow integration*: Present design flows for SoC/NoC are not seamlessly integrated. From application specification down to chip fabrication, there exist a number of concerns from physical issues (electrical and thermal), clocking, power, performance, verification, manufacturability and testability. A design flow usually targets one or a small subset of the design aspects. To enable a truly automated design flow, all relevant issues are preferably handled in an integrated design flow to leverage efficiency and overcome the inconsistency between different tools which may come from different vendors.

Technically, NoC has a huge potential to expand. It would come no surprise when yesterday's 1000-node supercomputers become tomorrow's 1000-node networks-on-chips. In addition, NoC will be driven not only for application-specific applications but also for general-purpose applications. Finally, SoC/NoC technology will be combined with other technologies, such as sensor-technology, nano-chemistry, biotechnology, micro-mechanics etc., into a multi-disciplinary technology. Innovative application domains will be further inspired by the needs of improving our life quality such as health care, entertainment, safety, information production and exchange, non-restricted communications and of improving our living, developing and ecological environment.

# References


[1] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. A. Zeferino. SPIN: A scalable, packet switched, on-chip micro-network. In *Design, Automation and Test in Europe Conference and Exhibition - Designers' Forum*, March 2003.

[2] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[3] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248 – 259, 2000.

[4] A. Allan, D. Edenfeld, J. W. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian. 2001 technology roadmap for semiconductors. *IEEE Computer*, 35(1):42–53, January 2002.

[5] D. Andreasson and S. Kumar. Slack-time aware routing in NoC systems. In *IEEE International Symposium on Circuits and Systems*, May 2005.

[6] ARM. AMBA advanced extensible interface (AXI) protocol specifcation, version 1.0. http://www.amba.com, 2004.

[7] S. Balakrishnan and F. Özgüner. A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):664–678, July 1998.

[8] N. Banerjee, P. Vellanki, and K. S. Chatha. A power and performance model for network-on-chip architectures. In *Proceedings of the Design Automation and Test in Europe Conference*, pages 1250–1255, 2004.

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

[10] L. Benini and G. D. Micheli, editors. *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.

[11] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[12] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages 12 years later. *Proceedings of The IEEE*, 91(1):64–83, January 2003.

[13] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Survey*, 38(1):1–54, 2006.

[14] T. Bjerregaard and J. Sparso. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 1226–1231, 2005.

[15] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. *The Journal of Systems Architecture*, December 2003.

[16] J. T. Brassil and R. L. Cruz. Bounds on maximum delay in networks with deflection routing. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):724–732, July 1995.

[17] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.

[18] S. Chalasani and R. V. Boppana. Fault-tolerant wormhole routing algorithms for mesh networks. *IEEE Transactions on Computers*, 44(7):848–864, 1995.

[19] T. Claasen. An industry perspective on current and future state-of-the-art in system-on-chip (SoC) technology. *Proceedings of the IEEE*, 94(6):1121–1137, June 2006.

[20] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. IPSIM: SystemC 3.0 enhancements for communication refinement. In *Proceedings of Design Automation and Test in Europe*, 2003.

[21] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. OCCN: A network-on-chip modeling and simulation framework. In *Proceedings of Design Automation and Test in Europe*, 2004.

[22] R. L. Cruz. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[23] R. L. Cruz. A calculus for network delay, part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.

[24] 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 21st International Conference on Computer Design*, September 2003.

[25] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–204, March 1992.

[26] W. J. Dally and C. L. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3):187–196, 1986.

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

[28] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufman Publishers, 2004.

[29] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli. A platform-based taxonomy for ESL design. *IEEE Design and Test of Computers*, 23(5):359– 374, September-October 2006.

[30] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Network - An Engineering Approach*. IEEE Computer Society Press, 1997.

[31] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded system: Formal models, validation and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[32] A. Gerstlauer, S. Dongwan, R. Domer, and D.D.Gajski. System-level communication modeling for network-on-chip synthesis. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 45–48, 2005.

[33] K. Goossens, J. Dielissen, J. Meerbergen, P. Poplavko, A. Rădulescu, E. Rijpkema, E. Waterlander, and P. Wielage. *Networks on Chip*, chapter Guaranteeing The Quality of Services. Kluwer Academic Publisher, 2003.

[34] K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):21–31, Sept-Oct 2005.

[35] A. G. Greenberg and J. Goodman. Sharp approximate models of deflection routing in mesh networks. *IEEE Transactions on Communications*, 41(1):210–223, January 1993.

[36] D. Gross and C. M. Harris. *Fundamentals of Queueing Theory*. Wiley, 1998.

[37] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 250–256, March 2000.

[38] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[39] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[40] S. L. Harry and F. Özgüner. Feasibility test for real-time communication using wormhole routing. *IEE Proceedings of Computers and Digital Techniques*, 144(5):273–278, September 1997.

[41] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Öberg, M. Millberg, and D. Lindqvist. Network on chip: An architecture for billion transistor era. In *Proceeding of the IEEE NorChip Conference*, November 2000.

[42] W. D. Hills. The Connection machine. *Scientific American*, 256(6), June 1987.

[43] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[44] J. Hu and R. Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. In *Proceedings of the Design Automation and Test in Europe Conference*, 2003.

[45] IBM. CoreConnect bus architecture - A 32-, 64-, 128-bit core on-chip bus structure. http://www-03.ibm.com/chips/products/coreconnect/.

[46] ITRS. International technology road map for semiconductors 2004 update: Design, 2004, wwww.itrs.net.

[47] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 158–168, 2002.

[48] A. Jantsch. *Modeling Embedded Systems and SoCs*. Morgan Kaufmann Publishers, 2004.

[49] A. Jantsch. Models of computation for networks on chip. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, June 2006.

[50] A. Jantsch and H. Tenhunen, editors. *Networks on Chip*. Kluwer Academic Publisher, 2003.

[51] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 1974.

[52] F. Karim, A. Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. *IEEE Micro*, 22(5):36–45, Sep/Oct 2002.

[53] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, January 1979.

[54] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits*, 19(12):1523–1543, December 2000.

[55] B. Kim, J. Kim, S. Hong, and S. Lee. A real-time communication method for wormhole switching networks. In *Proceedings of International Conference on Parallel Processing*, pages 527–534, Aug. 1998.

[56] 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*, 2002.

[57] S. K. Kunzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *Proceedings of Design, Automation and Test in Europe Conference*, pages 1–6, March 2006.

[58] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[59] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[60] E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, August 2004.

[61] A. Leroy, P. Marchal, A. Shickova, F. Catthoor, F. Robert, and D. Verkest. Spatial division multiplexing: a novel approach for guaranteed throughput on NoCs. In *Proceedings of the 3rd International Conference on Hardware/-Software Codesign and System Synthesis*, pages 81–86, 2005.

[62] J.-P. Li and M. W. Mutka. Real-time virtual channel flow control. *Journal of Parallel and Distributed Computing*, 32(1):49–65, 1996.

[63] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Depretter. System level design with SPADE: an M-JPEG case study. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2001.

[64] Z. Lu and R. Haukilahti. *Networks on Chip*, chapter NoC Application Programming Interfaces. Kluwer Academic Publisher, 2003.

[65] Z. Lu and A. Jantsch. TDM virtual-circuit configuration in network-on-chip using logical networks. *In submission to IEEE Transactions on Very Large Scale Integration Systems*.

[66] Z. Lu and A. Jantsch. Flit admission in on-chip wormhole-switched networks with virtual channels. In *Proceedings of International Symposium on System-on-Chip (ISSoC'04)*, pages 21–24, Tampere, Finland, November 2004.

[67] Z. Lu and A. Jantsch. Flit ejection in on-chip wormhole-switched networks with virtual channels. In *Proceedings of the IEEE Norchip Conference (Norchip'04)*, pages 273–276, Oslo, Norway, November 2004.

[68] Z. Lu and A. Jantsch. Traffic configuration for evaluating networks on chips. In *Proceedings of the 5th International Workshop on System on Chip for Real-time applications (IWSOC'05)*, pages 535–540, July 2005.

[69] Z. Lu, A. Jantsch, and I. Sander. Feasibility analysis of messages for on-chip networks using wormhole routing. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'05)*, pages 960–964, Shanghai, China, January 2005.

[70] Z. Lu, I. Sander, and A. Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS'02)*, pages 86–91, Kyoto, Japan, October 2002.

[71] Z. Lu, I. Sander, and A. Jantsch. *Applications of Specification and Design Languages for SoCs - Selected papers from FDL 2005*, chapter Refining synchronous communication onto network-on-chip best-effort services, pages 23–38. Springer, 2006.

[72] Z. Lu, I. Sander, and A. Jantsch. Towards performance-oriented pattern-based refinement of synchronous models onto NoC communication. In *Proceedings of the 9th Euromicro Conference on Digital System Design (DSD'06)*, pages 37–44, Dubrovnik, Croatia, August 2006.

[73] Z. Lu, R. Thid, M. Millberg, E. Nilsson, and A. Jantsch. NNSE: Nostrum network-on-chip simulation environment. In *The University Booth Tool-Demonstration Program of the Design, Automation and Test in Europe Conference*, March 2005.

[74] Z. Lu, L. Tong, B. Yin, and A. Jantsch. A power-efficient flit-admission scheme for wormhole-switched networks on chip. In *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, July 2005.

[75] Z. Lu, B. Yin, and A. Jantsch. Connection-oriented multicasting in wormhole-switched networks on chip. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 205–210, Karlsruhe, Germany, March 2006.

[76] Z. Lu, M. Zhong, and A. Jantsch. Evaluation of on-chip networks using deflection routing. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI'06)*, pages 296–301, Philadelphia, USA, May 2006.

[77] J. Madsen, S. Mahadevan, K. Virk, and M. Gonzalez. Network-on-chip modeling for system-level multiprocessor simulation. In *International Real-Time Systems Symposium*, 2003.

[78] S. Mahadevan, F. Angiolini, M. Storgaard, R. Olsen, J. Sparsø, and J. Madsen. A network traffic generator model for fast network-on-chip simulation. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 780–785, March 2005.

[79] G. Martin and H. Chang, editors. *Winning the SoC Revolution*. Kluwer Academic Publishers, 2003.

[80] J. W. McPherson. Reliability challenges for 45nm and beyond. In *Proceedings of the 43rd Design Automation Conference*, pages 176– 181, July 2006.

[81] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the Design Automation and Test in Europe Conference*, February 2004.

[82] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The Nostrum backbone - a communication protocol stack for networks on chip. In *Proceedings of the VLSI Design Conference*, Mumbai, India, January 2004.

[83] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(5), 1965.

[84] F. G. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69–93, 2004.

[85] T. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.

[86] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[87] M. D. Nava, P. Blouet, P. Teninge, M. Coppola, T. Ben-Ismail, S. Picchiottino, and R. Wilson. An open platform for developing multiprocessor SoCs. *IEEE Computer*, 38(7):60–67, July 2005.

[88] C. Neeb, M. Thul, and N. Wehn. Network-on-chip-centric approach to interleaving in high throughput channel decoders. In *IEEE International Symposium on Circuits and Systems*, pages 1766–1769, 2005.

[89] K.-H. Nielsen. Evaluation of real-time performance models in wormhole-routed on-chip networks. Master's thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Sweden, 2005.

[90] E. Nilsson, M. Millberg, J. Öberg, and A. Jantsch. Load distribution with the proximity congestion awareness in a network on chip. In *Proceedings of the Design Automation and Test in Europe Conference*, 2003.

[91] E. Nilsson and J. Öberg. Reducing peak power and latency in 2D mesh NoCs using globally pseudochronous locally synchronous clocking. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, September 2004.

[92] V. Nollet, M. Marescaux, and D. Verkest. Operating-system controlled network on chip. In *Proceedings of the 41st Design Automation Conference*, pages 256–259, Los Alamitos, CA, USA, 2004.

[93] J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, editors. *Interconnect-Centric Design for Advanced SoCs and NoCs*. Kluwer Academic Publisher, 2004.

[94] J. Öberg. *Networks on Chip*, chapter Clocking Strategies for Networks on Chip. Kluwer Accademic Publisher, 2003.

[95] OCP International Partnership. Open core protocol specification, version 2.0. http://www.ocpip.org, 2003.

[96] U. Y. Ogras, R. Marculescu, H. G. Lee, and N. Chang. Communication architecture optimization: making the shortest path shorter in regular networks-on-chip. In *Proceedings of Design, Automation and Test in Europe Conference*, March 2006.

[97] M. Palesi, R. Holsmark, and S. Kumar. A methodology for design of application specific deadlock-free routing algorithms for NoC systems. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 142–147, October 2006.

[98] D. Pamunuwa, J. Öberg, L.-R. Zheng, M. Millberg, A. Jantsch, and H. Tenhunen. A study on the implementation of 2D mesh based networks on chip in the nanoregime. *Integration - The VLSI Journal*, 38(2):3–17, October 2004.

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

[100] K. Park and W. Willinger, editors. *Self-Similar Network Traffic and performance Evaluation*. New York: Wiley, 2000.

[101] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, and G. Nicolescu. Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(7):667–680, July 2006.

[102] L. S. Peh and W. J. Dally. A delay model for router microarchitectures. *IEEE Micro*, 21(1):26–34, Jan.-Feb. 2001.

[103] S. G. Pestana, E. Rijpkema, A. Radulescu, K. Goossens, and O. P. Gangwal. Cost-performance trade-offs in networks on chip: A simulation-based approach. In *Proceedings of the Design, Automation and Test in Europe Conference*, 2004.

[104] Philips Semiconductors. Device transaction level (DTL) protocol specification, version 2.2, 2002.

[105] V. Raghunathan, M. B. Srivastava, and R. K. Gupta. A survey of techniques for energy efficient on-chip communication. In *Proceedings of Design Automation Conference*, June 2003.

[106] R. Ramaswami and K. N. Sivarajan. *Optical Networks: A Practical Perspective*. Morgan Kaufmann Publishers, 1998.

[107] P. Rashinkar, P. Paterson, and L. Singh. *System-On-A-Chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, 2001.

[108] T. Raudvere, I. Sander, A. K. Singh, and A. Jantsch. Verification of design decisions in ForSyDe. In *Proceedings of CODES+ISSS*, California, USA, October 2003.

[109] E. Rijpkema, K. Goossens, and P. Wielage. A router architecture for networks on silicon. In *Proceedings of Progress 2001, 2nd Workshop on Embedded Systems*, Veldhoven, The Netherlands, Oct. 2001.

[110] E. Rijpkema, K. G. W. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *Proceedings of Design Automation and Test in Europe Conference*, Mar. 2003.

[111] C. Rowen. *Engineering the Complex SoC*. Prentice Hall PTR, 2004.

[112] J. A. Rowson and A. Sangiovanni-Vincentelli. Interface based design. In *Proceedings of the 34th Design Automation Conference*, 1997.

[113] A. Rădulescu, J. Dielissen, P. S. González, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(1):4–17, 2005.

[114] E. Salminen, T. Kangas, V. Lahtinen, J. Riihimäki, K. Kuusilinna, and T. D. Hämäläinen. Benchmarking mesh and hierarchical bus networks in system-on-chip context (in press). *Journal of System Architectures*, 2007.

[115] I. Sander and A. Janstch. Transformation based communication and clock domain refinement for system design. In *Proceedings of the 39th Design Automation Conference*, pages 281 – 286, June 2002.

[116] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, February 2004.

[117] I. Sander, A. Jantsch, and Z. Lu. Development and application of design transformations in ForSyDe. In *Proceedings of Design, Automation and Test in Europe Conference*, pages 364–369, Munich, Germany, March 2003.

[118] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Design & Test of Computers*, pages 2–15, April-June 2000.

[119] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. Addressing the system-on-a-chip interconnect

woes through communication-based design. In *Proceedings of the 38th Design Automation Conference*, 2001.

[120] K. G. Shin and S. W. Daniel. Analysis and implementation of hybrid switching. In *Proc. of the 22nd International Symposium on Computer Architecture*, pages 211–219, 1995.

[121] J. Sicking. Implementation of asynchronous communication for ForSyDe in hardware and software. Master's thesis, Royal Institute of Technology, Sweden, IMIT/LECS/[2005-73].

[122] D. Siguenza-Tortosa, T. Ahonen, and J. Nurmi. Issues in the development of a practical NoC: the Proteo concept. *Integration, the VLSI Journal*, 38(1):95–105, 2004.

[123] K. Srinivasan, K. S. Chatha, and G. Konjevod. Linear programming based techniques for synthesis of network-on-chip architectures. *IEEE Transactions on VLSI Systems*, 14(4):407–420, 2006.

[124] R. Thid. A network on chip simulator. Master's thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Sweden, 2002.

[125] R. Thid, M. Millberg, and A. Jantsch. Evaluating NoC communication backbones with simulation. In *Proceedings of the IEEE NorChip Conference*, November 2003.

[126] R. Thid, I. Sander, and A. Jantsch. Flexible bus and NoC performance analysis with configurable synthetic workloads. In *Proceedings of the 9th Euromicro Conference on Digital System Design*, August 2006.

[127] S. Thompson. *Haskell - The Craft of Functional Programming*. Addison-Wesley, 2 edition, 1999.

[128] A. S. Vaidya, A. Sivasubramaniam, and C. R. Das. Impact of virtual channels and adaptive routing on application performance. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):223–237, 2001.

[129] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *International Conference on Hardware/Software Codesign and System Synthesis*, pages 206–217, 2004.

[130] VSI Alliance. Virtual component interface, standard version 2. http://www.vsi.org, 2000.

[131] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, November 2002.

[132] D. Wiklund and D. Liu. SoCBUS: Switched network on chip for hard real time embedded systems. In *International Parallel and Distributed Processing Symposium*, 2003.

[133] J. Xu, W. Wolf, J. Henkel, S. Chakradhar, and T. Lv. A case study in networks-on-chip design for embedded video. In *Proceedings of the Design Automation and Test in Europe Conference*, 2004.

[134] L.-R. Zheng. *Design, Analysis and Integration of Mixed-Signal Systems for Signal and Power Integrity*. PhD thesis, Royal Institute of Technology, 2001.

[135] H. Zimmermann. OSI Reference Model – the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.

# Paper 1

**Flit admission in on-chip wormhole-switched networks with virtual channels**

*Proceedings of the International Symposium on System-on-Chip*, pages 21-24, Tampere, Finland, November 2004.

# Flit Admission in On-chip Wormhole-switched Networks with Virtual Channels

Zhonghai Lu and Axel Jantsch
Laboratory of Electronics and Computer Systems
Royal Institute of Technology, Sweden
{zhonghai,axel}@imit.kth.se

## Abstract

*Flit-admission solutions for wormhole switches must minimize the complexity of the switches in order to achieve cheap implementations. We propose to couple flit-admission buffers with physical channels so that flits from a flit-admission buffer are dedicated to a physical channel. By the coupling strategy, for input-queuing wormhole lane switches, the complexity of the crossbars can be simplified from $2p \times p$ to $(p+1) \times p$, where $p$ is the number of physical channels; for output-queuing wormhole lane switches, the additional complexity is also minimal. We evaluate the flit-admission solutions derived from the coupling with uniformly distributed random traffic in a 2D mesh network. Experimental results show that these solutions exhibit good performance in terms of latency and throughput.*

## 1 Introduction

Wormhole switching is being proposed for Networks on Chips (NoCs) due to its better performance and smaller buffering requirement [1, 2]. To make efficient use of the Physical Channels (PCs), wormhole switching uses virtual channels (lanes) to gain higher throughput [3]. Several parallel lanes, each of which is a flit buffer queue, share a PC. For on chip wormhole switches, these lane buffers can be customized as dedicated hardware FIFOs instead of register-based or RAM-based FIFOs to reduce the area and thus achieve reasonable buffering cost [2]. To reduce the control complexity of the switches, deterministic routing is favored against adaptive routing. This may also be justified by exploiting the traffic predictability of specific applications [1]. Moreover, regular low-dimension topologies are considered for NoCs to further simplify the control [4, 5].

Figure 1 shows a 2D mesh NoC architecture [1, 4, 5]. Each resource is connected to a switch via a Network Interface (NI). The wormhole switch with bidirectional links has four PCs and several lanes per PC (not shown). Resources feed the network with packets, which are queued in the

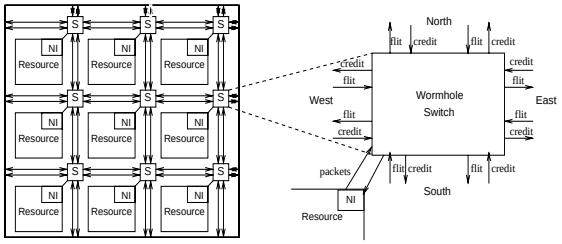



**Figure 1. A 2D mesh**

packet buffers of the switches. With wormhole switching, a packet is decomposed into a head flit, zero or more middle flit(s), and a tail flit. A single-flit packet is also possible. These flits are stored in flit-uploading buffers called *uploading/admission buffers* before being admitted to the network. There are various ways of organizing the packet buffer and the uploading buffers. In Figure 2.(a), flit-uploading buffers are organized as a FIFO. In Figure 2.(b) and 2.(c), they are arranged as $p$ parallel FIFO queues ($p$ is the number of PCs). Figure 2.(a) and 2.(b) allow at maximum one flit to be admitted to the network at a time while Figure 2.(c) allows up to $p$ flits to be admitted simultaneously.

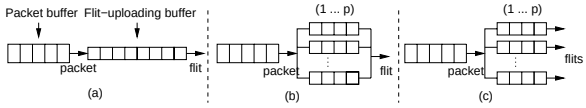



**Figure 2. Packet and flit-uploading buffers**

In the paper we investigate *flit admission* approaches in wormhole lane switches. We shall see that different solutions largely impact the complexity of the switches. Throughout the paper, we adopt the organization of flit-uploading buffers in Figure 2.(c) since it allows potentially higher performance. In the sequel, Section 2 discusses related work. In Section 3 we present our flit admission solutions for both input-queuing and output-queuing wormhole switches. Section 4 describes experimental results. Finally, we conclude the paper in Section 5.

## 2 Related Work

Wormhole switching with virtual channels was proposed in [3]. The performance model of a wormhole switch that considers implementation complexity was first noted by Chien [6]. Recently a more efficient canonical wormhole switch architecture for virtual-channel flow control and its performance model was presented in [7].

Admission control is commonly employed for real-time traffic to determine if admitting new real-time traffic can satisfy its timing bounds without jeopardizing the performance guarantees of real-time traffic already in the network. It has been a rich research area in packet-switched computer networks. In cluster computing, based on QoS-capable wormhole switches and network interfaces, an admission control algorithm in conjunction with a congestion control algorithm was designed for the admission of real-time traffic in the networks [8].

Our study on flit admission is different from the admission control for real-time traffic. By effectively sharing physical channels, our flit admission approaches are designed to minimize the complexity of wormhole switches without sacrificing performance.

## 3 Flit Admission Approaches

### 3.1 Flit admission in input-queuing switches

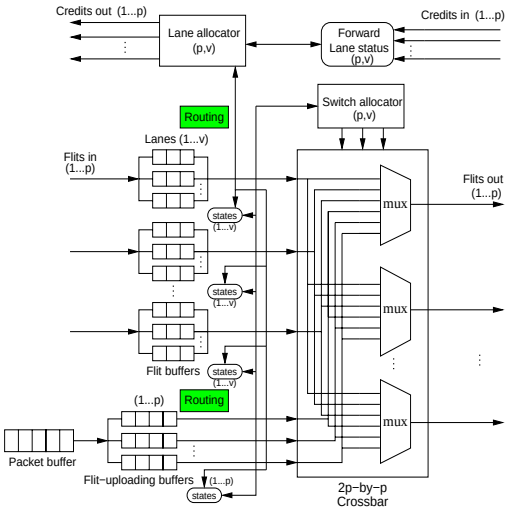



**Figure 3. A canonical wormhole lane switch**

Figure 3 shows a canonical wormhole switch architecture with virtual channels at inputs [2, 3, 7]. It has $p$ physical channels (PCs) and $v$ lanes per PC. A packet passes the switch through four states: *routing*, *lane allocation*, *flit scheduling*, and *switch arbitration*. In the routing state, the routing logic determines the routing path a packet advances. In the state of lane allocation, the lane allocator *associates* the lane the packet occupies with an available lane on its routing path in the next hop. If the lane allocation succeeds, the packet enters into the scheduling state. If there is a buffer available in the associated lane, the lane enters into the switch arbitration. The first level of arbitration is performed on the lanes sharing the same physical channel. The second level of arbitration is for the crossbar traversal. If the lane wins the two levels of arbitration, the flit situated at the head of the lane is switched out. Otherwise, the lane returns back to the scheduling state. The lane association is released after the tail flit is switched out. Credits are passed between adjacent switches in order to keep track of the status of lanes. Note that a lane is allocated at the packet level, i.e., packet-by-packet. Also, flits from different lanes can not be interleaved in a lane since flits other than head flits do not contain routing information. To guarantee this, a lane-to-lane association must be unique at a time.

In Figure 3, if an uploading buffer is available, a packet is split into flits which are then put into the uploading buffer. An uploading buffer takes the same four states as a lane in order to inject flits into the network. Flits from an uploading buffer can be switched to all the $p$ output PCs. Since the uploading buffers are decoupled from the PCs, the crossbar must be fully connected, resulting in a port size of $2p \times p$.

To alleviate the complexity of the switch, we propose to couple an uploading buffer with a PC in a *one-to-one* manner. In this way, flits from an uploading buffer are dedicated to a PC. Applying the coupling scheme to the switch in Figure 3, an uploading buffer only needs to be connected to one multiplexor instead of $p$ multiplexors. The size of the crossbar is sharply decreased from $2p \times p$ to $(p+1) \times p$, as shown in Figure 4. The number of control signals per multiplexor is reduced from $\lceil log(2p) \rceil$ to $\lceil log(p+1) \rceil$ for any $p > 1$ [1]. Alternatively, an uploading buffer can directly

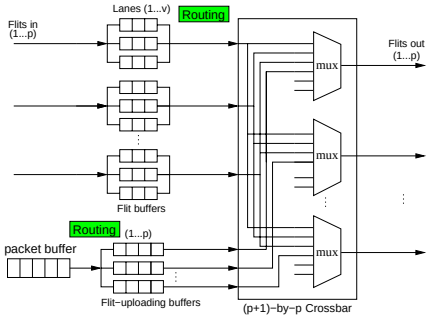



**Figure 4. Sharing a (p+1)-by-p crossbar**

share an output PC, as depicted in Figure 5.(a). This solu-

[1] $\lceil x \rceil$ is the ceiling function which returns the least integer that is not less than $x$.

tion can also be regarded as having a crossbar complexity of $(p+1) \times p$ since the combination of a $p \times 1$ multiplexor and a $2 \times 1$ multiplexor may be viewed as a $(p+1) \times 1$ multiplexor. The number of control signals per PC is reduced from $\lceil log(2p) \rceil$ to $\lceil logp \rceil + 1$ for any $p > 1$.

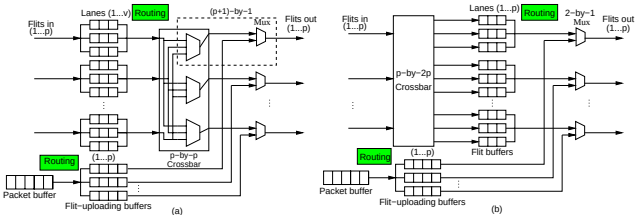



**Figure 5. Sharing output physical channels**

In order to support the coupling scheme, the routing must be performed before segmenting a packet and storing its flits in an uploading buffer instead. With a routing algorithm, the PC the packet requests can be determined. Hence, the corresponding uploading buffer is identified. One drawback due to the coupling is possible blocking propagation. Specifically, if the head packet in the packet buffer is blocked due to the bounded size of the uploading buffer it aims at, the packets behind the head packet are all unconditionally blocked during the head packet's blocking time.

### 3.2 Flit admission in output-queuing switches

In addition to sharing the crossbar or output PCs, flit admission may share input PCs of the switch. However, there is a *critical section* problem. Figure 6 illustrates the problem with a simplified graph of two connected wormhole switches, **A** and **B**. Suppose that lane $j$ in switch **B** is available at a certain clock cycle, the uploading buffer sees lane $j$ available and then associates itself to lane $j$ locally. At the same cycle, lane $i$ in switch **A** also detects lane $j$ available and remotely makes an association with lane $j$. This is possible since both switches maintain a consistent view of the lane status. As a result, two associations with a single lane are established in the same cycle. Consequently, lane $j$ will receive flits from lane $i$ and the uploading buffer, resulting in their flits possibly interleaved in lane $j$.

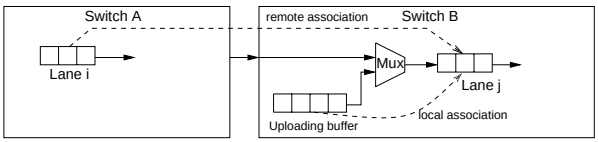



**Figure 6. Problem with sharing input PCs**

To avoid such a situation and thus achieve a mutual-excluded lane association needs both architectural support and a control protocol. This complicates the switch design and negatively impacts the network performance. Therefore, sharing input PCs is *not* favored as a flit admission solution. This observation illuminates flit admission in output-queuing wormhole switches. If sharing input PCs and crossbars in output-queuing wormhole switches, we encounter exactly the critical section problem, which is costly to resolve. This leads to only one low cost flit-admission path for output-queuing wormhole switches, i.e., sharing output PCs, as drawn in Figure 5.(b), where the multiplexors for admitting flits have a port size of $2 \times 1$. If the coupling strategy is not used, the multiplexors must be $(p+1) \times 1$.

## 4 Experiments

We developed a simulator in SystemC comprising the input-queuing wormhole switch model and other supporting objects. The switch is a single-cycle, flit-level model. We construct a 2D $K \times K$ ($K$=4) network without end-around connections (Figure 1). The network does dimension-order **X**-**Y** routing, which is deadlock-free and deterministic. The aim of our experiments is two-fold. First, we examine the performance of the two flit-admission solutions derived from the coupling scheme, i.e., sharing simplified crossbars (Figure 4) and sharing output PCs (Figure 5.(a)). The baseline architecture is the one admitting flits via full crossbars (Figure 3). Second, with admitting flits via output PCs, we investigate the impact of multiplexor arbitration.

The simulations were run with uniformly distributed traffic. Resources injected fixed-size packets to random destinations except for themselves at a constant rate. A flit is ejected from the network once it reaches a lane of its destination and the lane passes the routing state. Except otherwise noted, contentions for lanes and channel bandwidth were resolved randomly. Each simulation was run until the network reached steady state, i.e., increasing simulated network cycles did not change the results appreciably. We investigated the average latency of packets and the network throughput. Latency of a packet is calculated from the instant the packet's flits are created to that the last flit of the packet is accepted at the destination, including source queuing time. Throughput $\lambda$ is defined as the number of flits received per cycle per node.

| | |
|---|---|
| Number $v$ of lanes per physical channel | 3 |
| Size of a lane | 2 flits |
| Size of an uploading buffer | 4 flits |
| One packet | 4 flits |

**Table 1. Simulation parameters**

Simulation parameters are listed in Table 1. The size of a lane was chosen to be two, which is the minimal amount of buffer requirement for a lane in order to pipeline flits since sending and receiving credits take two cycles.

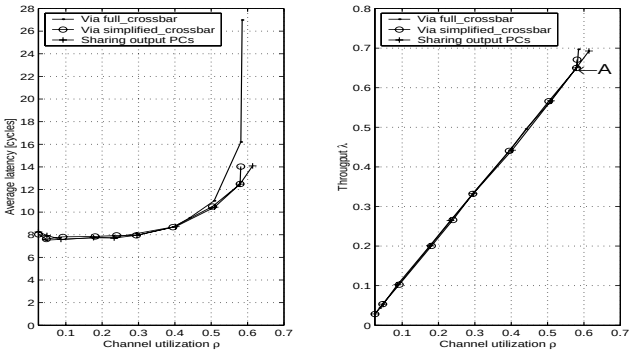



**Figure 7. Performance comparison**

Figure 7 compares the flit-admission approaches. Admitting flits via simplified crossbars and via output PCs achieve similar performance since they are equivalent for the input-queuing switch. Compared with admitting flits via full crossbars, the three solutions agree well when channel utilization $\rho$ is below 0.5. When $\rho$ is higher than 0.5, the average latency with admitting flits via full crossbars is worse. This suggests that faster uploading of flits results in higher congestion thus higher latency when the network is nearly saturated. It is interesting to note that the three approaches achieve the same channel utilization and throughput (point **A** in the Utilization-Throughput figure) given the same traffic patterns. Above this point, the packet buffers (refer to Figure 3) start to overflow, given a bounded packet buffer size. The Utilization-Throughput figure can serve as a validation of the network operations. The slope of the three lines is $\frac{9}{8}$, because $\lambda = C\rho/(MD_{avg})$, where $C$, $M$, $D_{avg}$ are the network capacity, the number of nodes, the average distance traveled by all received flits, respectively. For the 2D mesh, $C = 4K(K-1)$, $M = K^2$, $D_{avg} = \frac{2}{3}K$ for the random traffic. When $K = 4$, $\lambda = \frac{9}{8}\rho$.

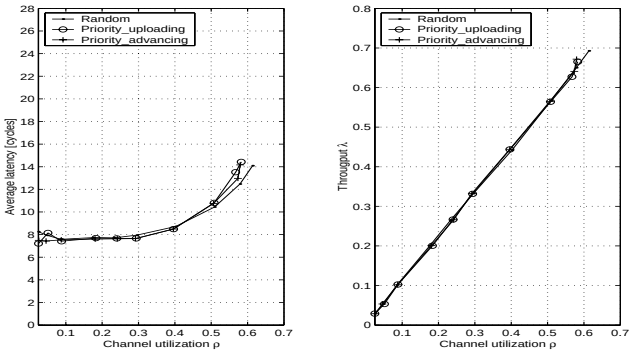



**Figure 8. Arbitration impact**

Figure 8 shows the performance of admitting flits via output PCs with three different arbitration criteria: random, priority_uploading meaning that flits to be admitted win contentions against network flits, priority_advancing meaning that network flits win contentions against flits to be uploaded. We can see that the three arbitration policies do not make significant difference. This is because, if the arbitration gives priority to uploading flits, uploading flits is faster, but the uploaded flits (network flits) lose arbitration along their routes; if the arbitration favors network flits, they win arbitration along their routes but are difficult to be admitted in the beginning. This sort of balance makes both cases close to the effect of the random arbitration.

## 5 Conclusions

We have discussed flit admission approaches for wormhole virtual-channel switches. By coupling flit-admission buffers with physical channels, the complexity of the crossbar can be reduced from $2p \times p$ to $(p+1) \times p$ for an input-queuing switch; the additional complexity for admitting flits is also minimal for an output-queuing switch. Simulation results show that these solutions derived from the coupling scheme do not compromise the performance. Although our discussions are equally applicable to macro wormhole-switched networks in parallel computing, the experiments were designed for a NoC that employs a low-dimension topology, deterministic routing, and smaller buffering cost.

Future work will consider flit admission together with packet admission. A higher-level admission control strategy can be devised to track network load so that packets are admitted with reasonable rates. Another direction is to combine flit admission with flit ejection. Practically cost-effective flit ejection models must be taken into account while evaluating the performance of the on-chip network.

## References


[1] J. Hu and R. Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. In *DATE*, 2003.

[2] E. Rijpkema, K. Goossens, et al. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *DATE*, 2003.

[3] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–204, March 1992.

[4] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *DAC*, 2001.

[5] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *DATE*, 2004.

[6] A. A. Chien. A cost and speed model for k-ary n-cube wormhole routers. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):150–162, Feb. 1998.

[7] L. S. Peh and W. J. Dally. A delay model for router microarchitectures. *IEEE Micro*, pages 26–34, Jan.-Feb. 2001.

[8] K. H. Yum et al. Integrated admission and congestion control for QoS support in clusters. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 325–332, Sept. 2002.

# Paper 2

**Flit ejection in on-chip wormhole-switched networks with virtual channels**

*Proceedings of the IEEE NorChip Conference*, pages 273-276, Oslo, Norway, November 2004.

# Flit Ejection in On-chip Wormhole-switched Networks with Virtual Channels

Zhonghai Lu and Axel Jantsch
Laboratory of Electronics and Computer Systems
Royal Institute of Technology, Sweden
{zhonghai,axel}@imit.kth.se

## Abstract

*An ideal flit-ejection model is typically assumed in the literature for wormhole switches with virtual channels. With such a model, flits are ejected from the network immediately upon reaching their destinations. This achieves optimal performance but is very costly. The required number of sink queues of a switch for absorbing flits is $p \cdot v$, where $p$ is the number of physical channels (PCs) of the switch; $v$ the number of lanes per PC. To achieve cheap silicon implementations, flit-ejection solutions must be cost-effective. We present a novel flit-ejection model and a variant of it where the required number of sink queues of a switch is $p$, i.e., independent of $v$. We evaluate the flit-ejection models with uniformly distributed random traffic in a 2D mesh network. Experimental results show that they exhibit good performance in latency and throughput.*

## 1 Introduction

Chip design is increasingly becoming communication-bound other than computation-bound with the steady technology scaling [1]. Network-on-Chip (NoC) [5, 6, 8, 11] addresses the design challenge by proposing networks to replace buses as a scalable global communication platform. In a NoC, heterogeneous resources such as processor cores, DSPs, FPGAs/ASICs, and memories are interconnected by switches, which route packets to enable communication between resources.

Network flow control governs how a packet is forwarded in a network, concerning shared resource allocation and contention handling. Wormhole switching [4] is a network flow control scheme that allocates buffers and physical channels (PCs) to flits instead of packets. A packet is decomposed into one or more flits. A flit, the smallest unit on which flow control is performed, can advance once buffering in the next hop is available to hold the flit. This results in that the flits of a packet are delivered in a pipeline fashion. For the same amount of storage, it achieves lower latency and greater throughput. However, wormhole switching uses channels inefficiently because a PC is held for the duration of a packet. If a packet is blocked, all PCs held by this packet are left idle. To mitigate this problem, wormhole switching adopts *virtual channels* (*lanes*) to make efficient use of the PCs [3]. Several parallel lanes, each of which is a flit buffer queue, share a PC. Therefore, if a packet is blocked, other packets can still traverse the PC via other lanes, leading to higher throughput. Because of these advantages, namely, better performance, smaller buffering requirement and greater throughput, wormhole switching with lanes is being advocated for on-chip networks [6, 11].

The ejection of flits in a wormhole-switched network concerns when and how the flits reaching destinations are ejected from the network and stored in flit sink queues (sinks) before being composed back into packets. An ideal ejection model has been assumed for wormhole lane switches. With such an ideal model, flits are ejected into sink queues instantly once they reach destinations (after routing). Also, ejecting flits does not interfere with advancing flits. Such a model is optimal for performance, but it is too costly to be suitable for silicon implementations. Given the number of PCs of a switch is $p$, the number of lanes per PC is $v$, the required number of sink queues of the switch is proportional to $p$ and $v$ in order to realize the ideal model.

In this paper, we present a novel *flit-ejection* or *sink* model and a variant of it for wormhole lane switches. Our models sharply reduce the required number of sink queues from $p \cdot v$ to $p$ without compromising much performance. In the sequel, Section 2 discusses related work. In Section 3 we describe a canonical wormhole lane switch and the ideal flit-ejection model. We present our flit-ejection models in Section 3, followed by experimental results in Section 4. Finally, we conclude the paper in Section 5.

## 2 Related Work

The performance model of a wormhole switch that considers implementation complexity was first noted by Chien [2]. A more efficient canonical wormhole lane switch archi-

tecture and its performance model was presented in [10]. In general, the design complexity of a wormhole lane switch is the function of $p$ and $v$. To gain further performance, flit-reservation flow control [9] was proposed which utilizes control flits to reserve bandwidth and buffers before transferring data flits. All of these works assume an ideal flit-ejection model while evaluating the network performance.

To our knowledge, no prior work discussing flit-ejection models other than an ideal model was reported. Our motivation is to reduce the switch complexity to achieve cost-effective designs on silicon. In line with this idea, Goossens et al. proposed to customize the lane buffers as dedicated hardware FIFOs instead of register-based or RAM-based FIFOs to reduce the area and thus achieve reasonable buffering cost [11]. Recently, cost-effective flit admission approaches for virtual-channel wormhole switches are discussed in [7]. To reduce the control complexity of the switches, deterministic routing is favored against adaptive routing. This may also be justified by exploiting the traffic predictability of specific applications [6], which NoCs target. Moreover, regular low-dimension topologies are considered for NoCs to further simplify the control [5, 8].

## 3 The Ideal Sink Model

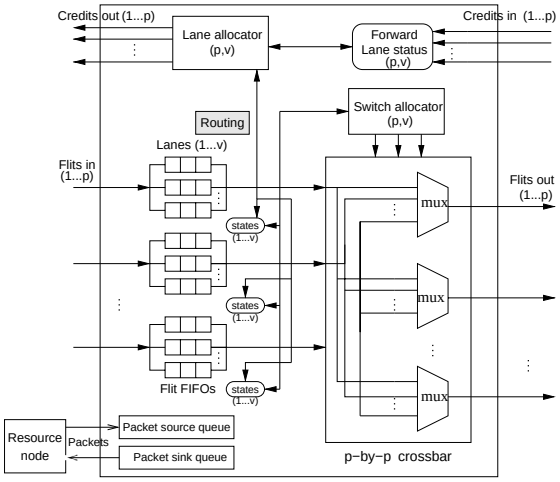



**Figure 1. A canonical wormhole lane switch**

Figure 1 shows a canonical wormhole switch architecture with virtual channels at inputs [3, 10, 11], connecting to a resource node. It has $p$ physical channels (PCs) and $v$ lanes per PC. A packet passes the switch through four states: *routing*, *lane allocation*, *flit scheduling*, and *switch arbitration*. In the routing state, the routing logic determines the routing path a packet advances. In the state of lane allocation, the lane allocator *associates* the lane the packet occupies with an available lane on its routing path in the next hop. If the lane allocation succeeds, the packet enters into the scheduling state. If there is a buffer available in the associated lane, the lane enters into the switch arbitration. The first level of arbitration is performed on the lanes sharing the same PC. The second level of arbitration is for the crossbar traversal. If the lane wins the two levels of arbitration, the flit situated at the head of the lane is switched out. Otherwise, the lane returns back to the scheduling state. The lane association is released after the tail flit is switched out. Credits are passed between adjacent switches in order to keep track of the status of lanes. Note that a lane is allocated at the packet level, i.e., packet-by-packet while the PC bandwidth is assigned at the flit level, i.e. flit-by-flit. In addition, flits from different lanes can not be interleaved in a lane since flits other than head flits do not contain routing and sequencing information. To guarantee this, a lane-to-lane association must be unique at a time.

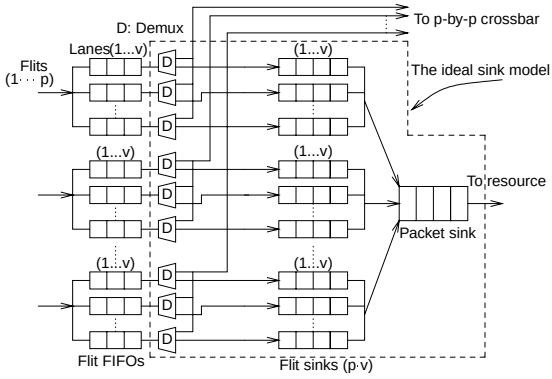



**Figure 2. The ideal sink model**

In a traditional wormhole switch design, an ideal sink model is assumed, as shown in Figure 2. The lane state is extended with a *reception* state. If the routing determines that the head flit of a packet reaches its destination, the lane enters the reception state immediately. Since flits from different packets can not interleave in a sink queue, there must be $p \cdot v$ sink queues, each of them corresponding to a lane, in order to realize an immediate transition to the reception state. The length of a sink is the maximum number of flits of a packet. After the lane transitting to the reception state, the head flit bypasses the crossbar and enters into its sink. The subsequent flits of the packet are ejected into the sink immediately upon arriving at the switch. When the tail flit is ejected, the lane is freed. This model is beneficial in both time and space. A non-head flit reaching its destination neither waits to be ejected nor occupies a flit buffer. Moreover, it does not interfere with flits buffered in other lanes from advancing to next hops. Upon receiving all the flits of a packet, the packet is composed and delivered into the packet sink. If the packet sink is not empty, the switch outputs one packet per cycle from the sink in a FIFO manner.

## 4 Proposed Sink Models

### 4.1 A $p$-sink model

Our objective is to simplify the ideal sink model with small performance penalty. We observe that the maximum number of flits entering a switch per cycle is $p$. This means that at maximum $p$ flits may need to be ejected from a switch per cycle. This number is independent of $v$. By this observation, we can use $p$ sink queues instead of $p \cdot v$ sink queues. The length of a sink is still the maximum number of flits of a packet. Besides, in order to have a more structured design, we could connect the $p$ sink queues to the crossbar, as illustrated in the dashed box of Figure 3.

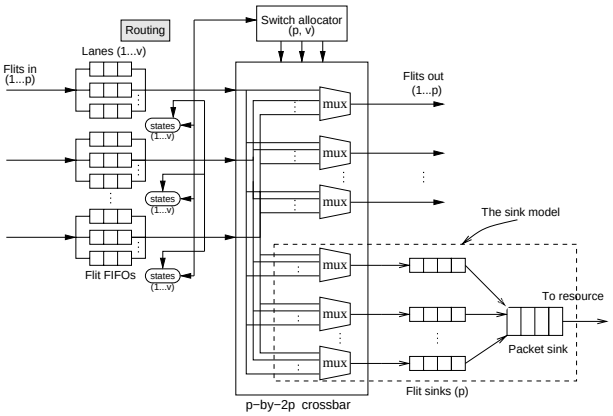



**Figure 3. A $p$-sink flit ejection model**

To enable ejecting flits by the $p$-sink model, we now extend the lane state to a *arriving* and a *reception* state. If a head flit reaches its destination, the lane the flit occupies transits from the routing to the arriving state. Then it will try to associate with an empty sink, i.e., to establish a *lane-to-sink* association. If the association is successful, the lane enters the *reception* state. Subsequently the other flits of the packet follow this association exactly like flits advancing in the network. Upon the tail flit entering into the sink, the association is torn down. If the association fails, the head flit is blocked in place holding the lane buffer. To speed up flit ejection, the contentions for the crossbar input channels and crossbar are arbitrated on priority. A lane in a reception state has a higher priority than a lane in a state for forwarding flits. The drawback due to this sink model is the increase of blocking time when flits reach their destinations. First, the lane-to-sink association may fail since all sink queues might be in use. Second, only one lane per PC can win arbitration to a crossbar input channel. In case of more than one lane of a PC are in an ejection state, only one can use the channel.

To implement this model, the crossbar must double its capacity from p-by-p ($p$ $p \times 1$ multiplexers) to p-by-2p ($2p$ $p \times 1$ multiplexers). The number of control ports of the crossbar is doubled proportionally.

### 4.2 A coupling scheme

To further simplify the switch, we could modify the $p$-sink model by using a coupling scheme in which the flits from a PC are dedicated to a sink. In other words, lane($i$, $j$), where $i$ ($i \in [1, p]$) is the PC identifier and $j$ ($j \in [1, v]$) the lane identifier, is dedicated to sink($i$). In this way, the $p$ $p \times 1$ multiplexers for sinking flits can be replaced with $p$ $1 \times 2$ demultiplexers, as shown in Figure 4.

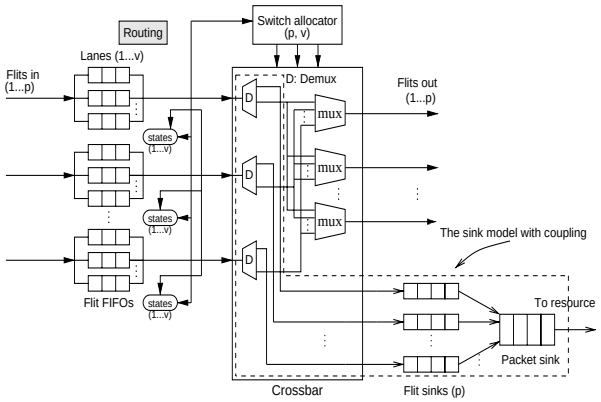



**Figure 4. The coupled $p$-sink model**

Due to this coupling, lane($i$, $j$) is not allowed to use a sink($k$), where $k \neq i$, even if it is empty. This potentially increases the blocking time of flits and the sink queues might be under-utilized.

| | $p \times 1$ Mux | $1 \times 2$ Demux | Sink queue |
|---|---|---|---|
| Ideal model | - | $p \cdot v$ | $p \cdot v$ |
| Decoupled model | $p$ | - | $p$ |
| Coupled model | - | $p$ | $p$ |

**Table 1. Cost of the sink models**

Table 1 summarizes the number of each component to implement the sink models.

## 5 Experiments

We developed a simulator in SystemC comprising the input-queuing wormhole switch model and other supporting objects. The switch is a single-cycle, flit-level model. The simulator is programmable as to network size, packet injection rate, sink model, etc. We construct a 2D $4 \times 4$ mesh network with bidirectional channels. The network does dimension-order **X-Y** routing, which is deadlock-free and deterministic. The purpose of our experiments is to examine the performance (latency and throughput) of the $p$

sink model in Figure 3 and its coupled counterpart in Figure 4. The baseline is the ideal sink model in Figure 2.

The simulations were run with uniformly distributed traffic. Resources injected 4-flit packets to random destinations except for themselves at a constant rate. Except otherwise noted, contentions for lanes and channel bandwidth were resolved randomly. Each simulation was run until the network reached steady state, i.e., increasing simulated network cycles did not change the results appreciably. We investigate the average latency of packets and the network throughput. Latency of a packet is calculated from the instant the packet's flits are created to that the packet is output to its destination resource, including source queuing time and packet queuing time at the destination. Throughput $\lambda$ is the number of packets received per cycle per node.

| | |
|---|---|
| Number $v$ of lanes per physical channel | 3 |
| Length of a lane | 2 flits |
| Length of a sink queue | 4 flits |

**Table 2. Simulation parameters**

Simulation parameters are listed in Table 2. To minimize buffering cost, the lane length was two, which is the minimal depth requirement of a lane in order to pipeline flits since sending and receiving credits take two cycles.

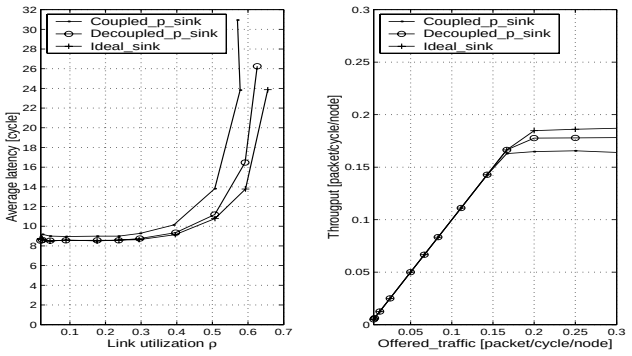



**Figure 5. Performance comparison**

Figure 5 compares the performance of the proposed sink models with that of the ideal model. The left figure gives the average latency in the function of link utilization (network load). Compared with the ideal model, the decoupled $p$-sink model achieves equivalent performance below link utilization $\rho = 0.5$. When $\rho$ is higher than 0.5, the latency with the ideal model is better. This is because slower ejection of flits results in higher congestion thus higher latency when the network is nearly saturated. As can be expected, the performance of the coupled $p$ sink model is worse than that of its decoupled counterpart. However, the penalty is not significant. Specifically, when the link utilization is below 0.4, the latency difference is about 0.5 cycle. The right figure reports the throughput versus the offered traffic, which is measured in terms of the number of packets injected per cycle per node. The saturation throughput for the coupled, decoupled, and ideal model is 0.165, 0.178, and 0.186, respectively. Normalized with the ideal model, the relative throughput is 0.96, 0.89, and 1, respectively.

## 6 Conclusions

Cost-effective flit-ejection models for wormhole switches are desired for chip implementations. We have presented a novel sink model that achieves approximate performance with the ideal model when the network is reasonably loaded (below 0.5 capacity). By coupling a physical channel with a sink queue, the switch complexity is further reduced with small performance penalty. Although our discussions are equally applicable to macro wormhole-switched networks in parallel computing, the experiments were designed for a NoC that employs a low-dimension topology, deterministic routing, and smaller buffering cost.

Future work will combine flit ejection together with flit admission in order to achieve practically cost-effective flit admission/ejection solutions. Such a combination is essential for evaluating the performance of an on-chip network.

## References


[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 248 – 259, 2000.

[2] A. A. Chien. A cost and speed model for k-ary n-cube wormhole routers. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):150–162, Feb. 1998.

[3] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–204, March 1992.

[4] W. J. Dally and C. L. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3):187–196, 1986.

[5] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *DAC*, 2001.

[6] J. Hu and R. Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. In *DATE*, 2003.

[7] Z. Lu and A. Jantsch. Flit admission in on-chip wormhole-switched networks with virtual channels. In *Proceedings of International Symposium on System-on-Chip*, 2004.

[8] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *DATE*, 2004.

[9] L. S. Peh and W. J. Dally. Flit-reservation flow control. In *Proceedings of High Performance Computer Architecture*, pages 73–84, Jan. 2000.

[10] L. S. Peh and W. J. Dally. A delay model for router microarchitectures. *IEEE Micro*, pages 26–34, Jan.-Feb. 2001.

[11] E. Rijpkema, K. Goossens, et al. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *DATE*, 2003.

# Paper 3

**Connection-oriented multicasting in wormhole-switched networks on chip**

*Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'06)*, pages 205-210, Karlsruhe Germany, March 2006.

# Connection-oriented Multicasting in Wormhole-switched Networks on Chip

Zhonghai Lu, Bei Yin and Axel Jantsch
Laboratory of Electronics and Computer Systems
Royal Institute of Technology, Sweden
{zhonghai,axel}@imit.kth.se, {beiy}@kth.se

## Abstract

*Network-on-Chip (NoC) proposes networks to replace buses as a scalable global communication interconnect for future SoC designs. However, a bus is very efficient in broadcasting. As the system size scales up to explore the chip capacity, broadcasting in NoCs must be efficiently supported. This paper presents a novel multicast scheme in wormhole-switched NoCs. By this scheme, a multicast procedure consists of establishment, communication and release phase. A multicast group can request to reserve virtual channels during establishment and has priority on arbitration of link bandwidth. This multicasting method has been effectively implemented in a mesh network with deadlock freedom. Our experiments show that the multicast technique improves throughput, and does not exhibit significant impact on unicast performance in a network with mixed unicast and multicast traffic if the network is not saturated.*

## 1 Introduction

As the technology steadily scales, chip design is increasingly becoming communication-bound. Network-on-Chip [1, 5, 10] addresses the design challenges by proposing networks to replace buses as a scalable global communication platform. In a NoC, heterogeneous resources such as processors, DSPs, FPGAs/ASICs, and memories are interconnected by switches. These resources communicate by routing packets instead of using dedicated wires.

Buses (a single bus, segmented or crossbar-type buses and a hierarchy of buses) do not scale well with the system size in bandwidth and clocking frequency. However, a bus is very efficient in broadcasting since all clients are directly connected to it. A network allows many more concurrent transactions, but it does not directly support multicast. As there exists a variety of SoC applications, many applications necessitates to support multicast in the case of passing global states, managing and configuring the network, and implementing cache coherency protocols etc. Particularly, real-time constrained, throughput-oriented embedded applications for multi-media processing will demand an efficient means to implement multicast. One crucial aspect for supporting multicast in SoCs is Quality-of-Service (QoS), which means that the performance of multicast traffic should be predictable. Implementing multicast by sending multiple unicast messages is neither efficient nor scalable. In addition, in a network with a mixture of unicast and multicast traffic, multicast traffic should not degrade the performance of unicast traffic since multicast traffic takes only a portion of the total network traffic.

In this paper we present a connection-oriented multicast scheme in wormhole-switched networks on chip. Wormhole switching [3] is a network flow control mechanism that allocates buffers and physical channels (PCs) to flits instead of packets. A packet is encapsulated into one or more flits. A flit, the smallest unit on which flow control is performed, can advance once buffering in the next hop is available to hold the flit. This results in that the flits of a packet are delivered in a pipeline fashion. In order to make an efficient use of link bandwidth, wormhole switching can employ *virtual channels* (VCs or lanes) to enhance throughput [2]. Because of these advantages, namely, *better performance*, *smaller buffering requirement* and *greater throughput*, wormhole switching with lanes is being advocated for on-chip networks [5, 8].

By our multicast scheme, multicasting consists of three phases: *group setup*, *communication*, and *group release*. A multicast is realized by sending a single copy of multicast packets to multicast group members along a pre-established path. This results in low packet overhead for multicasting. During the setup phase, multicasting can be aware of QoS in the sense that a multicast group may request to reserve VCs, and enjoy a higher priority against unicast packets for link bandwidth arbitration. Although the three-phase (setup, transmission, and release) communication has been used for establishing virtual-circuit communication to support QoS in store-and-forward packet-switched networks, applying the technique to implement multicast in a wormhole-switched network on chip is the novel aspect of this paper. Moreover, we shall look at how much impact multicast traffic will exert on unicast traffic, and the performance tradeoff between *multicast without VC reservation* and *multicast with VC reservation*.

Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI'06)
0-7695-2533-4/06 $20.00 © 2006 IEEE

## 2 Related Work

Multicasting in wormhole-switched networks has been extensively studied in parallel machines in order to support collective communications such as barrier synchronization, reduction and global combining [6, 9]. Multicast can be achieved via software or hardware approach. With software implementation of multicast, a multicast operation is implemented by sending a separate copy of the messages from the source node to every destination or to a subset of destinations, each of which in turn forwards the message to one or more other destinations in a multicast tree.

As the software approach is not efficient enough, hardware support of multicast communication is proposed. With the *tree-based multicast* [9], the destination set is partitioned at the source, and separate copies of the message are transmitted. A message may be replicated at intermediate nodes and forwarded to disjoint subsets of destinations in the tree. This scheme does not perform well to be deadlock free unless messages are very short because the entire tree is blocked if any of its branches are blocked. A solution is to forbid branching at intermediate nodes, leading to a multicast path pattern, called *path-based multicast* [6]. In order to reduce the length of the multicast path, the set of destination nodes may be divided into multiple disjoint subsets. A copy of the source message is sent across several multicast paths, each path for each subset of the destination nodes. In this scheme, multicasting is realized by sending multi-destination messages. The header of multi-destination messages must carry the addresses of all the destination nodes. As the header is an overhead, the message latency is increased and the effective network bandwidth is reduced. Besides, multicast traffic does not reserve network resources, equally competing with unicast traffic for buffers and link bandwidth, resulting in no QoS for multicasting.

By the traditional multicast schemes, group formation and multicast communication are not decoupled. The multicast-packet overhead is high and there is no QoS concern. In our multicast scheme, there is an explicit multicast group setup phase. After a group is set up, multi-destination messages carry only the group identity number not the addresses of all the destination nodes. In addition, a multicast group can be aware of QoS by reserving lanes for performance enhancement.

For a circuit-switched network on chip, a multicasting scheme using global traffic information is proposed in [7]. This scheme is difficult to scale to a large system size since it relies on the global network state. Connection-oriented communication has been proposed in the Mango [1] and Æthereal [5] NoCs to achieve QoS for unicasting. In Mango, connections are created using asynchronous/clockless circuitry. By reserving link bandwidth, Æthereal builds connections to provide a virtual contention-less path from sources to destinations. We use the connection-oriented technique to realize QoS-aware multicasting in a best-effort network. As stated, the contention-free route in the Æthereal NoC [5] and the looped containers [10] in the Nostrum NoC can be used to realize multicasting. But no concrete results are released so far.

## 3 The Multicast Scheme

### 3.1 Unicast in wormhole networks

Figure 1 sketches an input-buffering wormhole switch with lanes. It employs credit-based link-level flow control to coordinate packet delivery between switches.

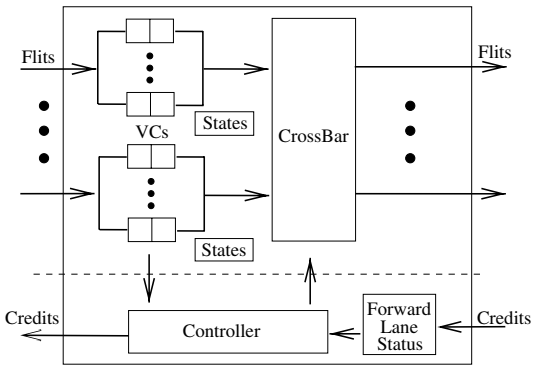



Figure 1: An input-buffering wormhole switch

A packet is segmented into flits, which are then delivered in the network. After the segmentation, a packet is typically composed of a head flit, a tail flit and body flit(s). A single-flit packet is also possible. A packet passes the switch through four states: *routing*, *lane allocation*, *flit scheduling*, and *switch arbitration*. In the routing state, the routing logic determines the routing path the packet advances. Routing is performed only when the head flit of a packet becomes the earliest-come flit in the lane. This means that if flits of a previous packet still stay in the lane, the routing will not be performed. Only when the earlier-coming flits are switched out, the head flit becomes the earliest-come flit. Then routing is performed, and the packet path and output physical channel are determined. In the state of lane allocation, the lane allocator *associates* the lane the packet occupies with an available lane in the next hop on its routing path, i.e., to make a *lane-to-lane* association. Note that it is not necessarily required that there is an empty buffer in the lane in order for the lane to be associated or allocated. A lane-to-lane association fails when all requested lanes in the next hop are already associated to other lanes in directly connected switches. If the lane-to-lane association succeeds, the packet enters into the scheduling state. If there is a buffer available in the associated lane, the lane

enters into the switch arbitration. The first level of arbitration is performed on the lanes sharing the same physical channel. The second level of arbitration is for the crossbar traversal to output physical channels. If the lane wins the two levels of arbitration, the earliest-come flit in the lane is switched out. Otherwise, the lane returns back to the scheduling state. Once the tail flit is switched out, the lane-to-lane association is released, thus the allocated lane is available to be used by other packets. Credits are passed between adjacent switches in order to keep track of the statues of downstream/forward lanes, such as if a lane is free, and a count of available buffers in the lane.

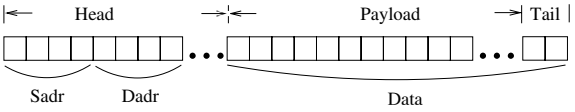



Figure 2: Unicast packet format

Figure 2 shows a typical unicast packet format, which consists of a head, a payload and a tail. The head and tail are the overhead for transmitting the payload. The head typically consists of routing and sequencing information. Basic routing information includes source address (sadr) and destination address (dadr). A switch uses the destination address to perform routing and switches the packet to the right output physical channel (PC). When the packet is split into flits, each flit contains a flit type field to identify if it is a head (H), body (B), tail (T) flit, or a single-flit packet.

## 3.2 The multicasting protocol

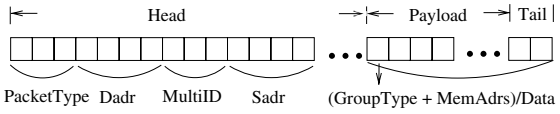



Figure 3: Multicast packet format

In order to support multicasting, we expand the packet format into that shown in Figure 3. We explain the packet fields as follows:

- **PacketType** indicates the purpose of a packet. It has six options, namely, *unicast, multicast setup, multicast setup response, multicast data, multicast group release* and *multicast group release acknowledgment*.
- **Dadr**: the destination address. In the case of multicast, it is the address of the next group member.
- **MultiID**: the multicast group identity number, which is unique for each multicast group to be established.
- **Sadr**: the source address. In the case of multicast, it is the address of the node that initiates the multicast setup. This node is called *group master*.
- **GroupType**: the type of the multicast group. It is used to inform the switches whether the multicast group will reserve a lane or not. It occupies only one bit. One group is allowed to reserve only one lane in a switch since the number of lanes is limited.
- **MemAdrs**: the multicast members' addresses. The order of the address list specifies the multicast path [1]. Specifically, the node with the first address in the list will be reached first and then second and so on. Upon reaching the node with address **Dadr**, the next member address in the list will replace the **Dadr** field.

Fields **GroupType** and **MemAdrs** are only needed for multicast setup packets, **MultiID** for multicast packets. A response packet for multicast group setup or release is handled as a unicast packet. By our scheme, a multicast group can be established and released dynamically. A multicasting procedure consists of three phases explained as follows:

1. *Group establishment*: First the group master sends a setup packet, which passes downstream to all the group member nodes along the predetermined path as indicated by **MemAdrs**. When the setup packet reaches a node, the switch records the multicast information and reserves resources according to the group type. This record will be used later to transmit multicast data. If the setup packet reaches the last group member, a setup response packet will be sent back to the group master to acknowledge the success. If the setup fails in a node, for example, due to lane unavailability, a response packet will be sent back to the master from the current node.

2. *Multicast communication*: After a successful setup, the master can send multicast data packets. The packets carrying **MultiID** will be transmitted along the same path and the same VCs which the setup packet used before. When a data packet reaches a destination node in the group, its payload is replicated and the packet is forwarded to the next member. In this way, all the members will receive the packet. The group members can also send multicast data packets, but only to the members in the downstream since a multicasting path is simplex and thus only simplex communication is allowed.

3. *Group release*: A group can only be released by its master by sending a release packet to its members. When the release packet reaches a node, the multicast record in the switch and the reserved lane will be freed after all on-going group transactions complete. Upon reaching the last member, a release acknowledgment is sent back to the master.

[1] In our approach, the multicast setup path can be diverse, and it shall follow the path-based schemes. But this is not the focus of the paper.

## 3.3 Multicast implementation

### 3.3.1 Extending the unicast switch

We have implemented the unicast switch in VHDL according to the model shown in Figure 1. The implementation consists of a data path and a control path. The data path is concerned with the flit movement through the crossbar and virtual channel. The control path realizes the functionality of the controller. For flit ejection, we implement a $p$-sink model to reduce cost [8]. By this model, a switch uses $p$ flit sinks to eject flits, where $p$ is typically equal to the number of PCs per switch. These $p$ sinks are shared by the $p \cdot v$ lanes, where $v$ is the number of VCs per PC.

Based on the unicast switch model, we have implemented the multicast scheme. The resultant switch supports both unicast and multicast. The data path is maintained the same as unicast while the control path is complicated. Specifically, the controller is extended to distinguish different packet types and perform actions according to the protocol. The switch must record the multicast information for each group passing it. The record of a multicast group includes {**MultiID**, **GroupType**, **Sadr**, **VCID**, **VCID downstream**, **output PC**, **next member adr.**}, where **VCID** is the identity number of the lane a multicast packet passes in the current switch; **VCID downstream** is the lane allocated downstream; **output PC** is the output physical channel the packet is to be switched out; **GroupType** indicates if the group reserves a lane or not. The current implementation arbitrates link bandwidth in favor of multicast traffic.

### 3.3.2 Deadlock avoidance

Deadlock is catastrophic to a network. It happens when a packet waits for an event that cannot happen. For example, a group of packets are unable to make progress because of waiting on one another to release buffers or channels. Forbidding such a cyclic resource dependency is a sufficient condition to design a deadlock-free network. Deadlock is related to many factors such as the network topology, flow control scheme, communication protocol and so on. Restricting the routing choice and adding buffer classes are the basic ways to deal with it.

Our multicast scheme has been implemented in a 2D mesh network employing dimension-order XY routing, which is proven to be deadlock free for unicast traffic on meshes. We constrain that a multicast path follows XY routing. This removes cyclic dependencies involving the two dimensions. However, care must be taken when planning a multicast path. Resulting from an improper path, a multicast packet may involve the turn from Y to X. As shown in Figure 4, a cycle is formed if the group is organized as $A \rightarrow B \rightarrow C \rightarrow D$. To avoid such a cycle, a group path must be organized so that no turn from Y to X

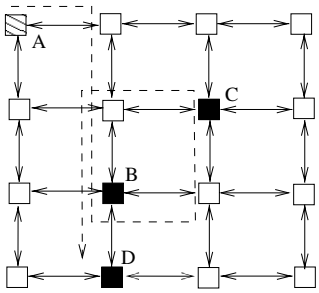



Figure 4: A cycle in a group

is resultant. For example, the group may be organized as $A \rightarrow C \rightarrow B \rightarrow D$. This simplification avoids deadlock at the expense of restrictions on planning groups and paths.

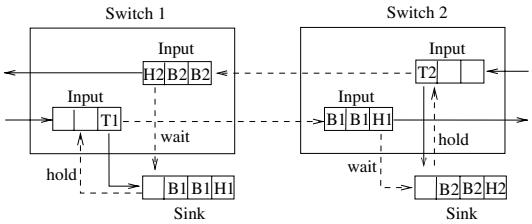



Figure 5: Deadlock while sinking and forwarding

Since a multicast packet has to sink locally at a member node and meanwhile to be forwarded downstream, the allocation of both a local sink and a lane in the next hop must be successful. In order not to introduce extra buffers and complicate the control, we decide to sink a flit when both the allocation conditions are true. This means that we perform sinking and forwarding on a multicast flit simultaneously. This may lead to deadlock with the $p$-sink ejection. As illustrated in Figure 5, two four-flit multicast packets pass two adjacent switches. While sinking and forwarding, both packets hold the sink the other waits for. The dashed lines in Figure 5 shows the wait-for graph [3], which forms a dependency cycle. Two solutions may be used to remove the cycle. One is to make the lane size long enough to hold an entire packet. The other is to ensure that the actual number $p$ of sinks is larger than the number of multicast groups passing a switch, and a multicast packet does not wait for the availability of a particular sink. This guarantees that there is at least one sink available to break a possible dependency cycle due to the exhaustion of sink resources.

When a multicast setup fails, a negative response (nack) packet will be sent from the failing switch back to the group master, which in turn will send a release packet to the network. This may create a dependent loop (A positive response does not cause a loop). By adopting the technique of the credit-based end-to-end flow control and separate buffer classes in [4], we are certain that this never causes deadlock.

# 4 Experiments

The purposes of our experiments are to (1) compare multicasting with unicasting multiple packets; (2) investigate the impact of multicast traffic on unicast traffic in a mixed unicast-multicast network; (3) evaluate the multicast scheme with/without lane reservation.

Using the multicast-supported wormhole model, we construct a 1×7 and 4×4 mesh. The networks operate synchronously. With the switch model, it takes 5 cycles for a head flit and 3 cycles for other flits to pass through a switch. Each switch has the same configuration parameters as follows: the number of VCs per PC is four for the 1×7 mesh and six for the 4×4 mesh; the depth of a VC is two, which is the minimal number in order to pipeline flits; the number of sinks is eight for the 1×7 mesh and 24 for the 4×4 mesh. Four-flit packets are injected into the network synchronously at a constant rate. Each node has a workload of 1000 packets. Simulations terminate when anyone of the nodes completes transmission. Latency of a packet is recorded from the instant that the packet is queued in the source FIFO to that the packet is ejected from the network. Network load is the average percentage of active links through the simulation cycles. Throughput is defined as the number of packets received per cycle per node.

## 4.1 Multicast vs. unicast

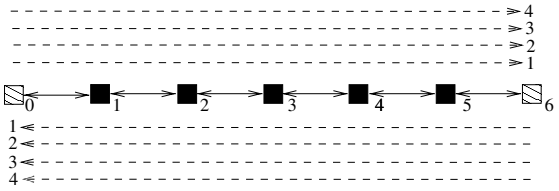



Figure 6: Traffic scenario in the 7-ary 1-mesh

As shown in Figure 6, we use the seven-node array in this set of experiments. Either unicast or multicast packets exist in the network. In the case of pure unicast, sending packets to multiple destinations is implemented by unicasting. In the experiments, node 0 and 6 send packets to the other six nodes randomly. In the multicast setting, four groups per direction are created, and the multicast groups do not reserve lanes. Node 0 and 6 are the group masters, which send multicast packets to the four groups alternatively.

Figure 7 depicts the results. With the same injection rate, the network is more loaded with the multicast, since multicast packets are delivered to all other nodes until reaching the other end while a unicast packet is sent to a particular node. The latency is worse with the multicast packets, this is due to the co-allocation of lane and sink for a multicast packet. However, the throughput of the multicast case is six

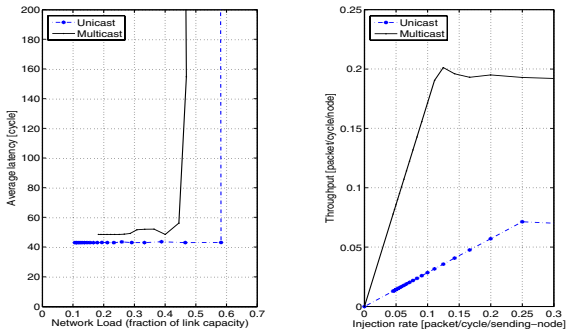



Figure 7: Multicast vs. unicast performance

times as much as that of the unicast case before the multicast network reaches saturation.

## 4.2 Multicast vs. mixed traffic

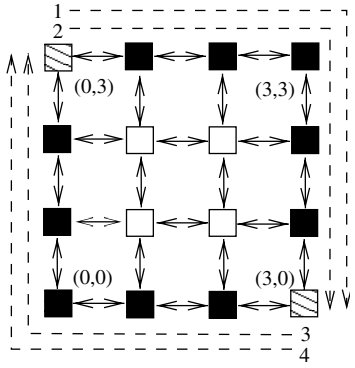



Figure 8: Traffic scenario in the mesh

The 4×4 mesh is used, and two traffic scenarios are created for the following experiments. One is *purely unicast traffic*. All network nodes send unicast packets to random destinations except themselves. The other is a *mixed unicast-multicast* scenario where four multicast groups start establishment upon simulation starts (meanwhile, unicast traffic is also injected.). As illustrated in Figure 8, four multicast groups are set up. From node (0, 3) to node (3, 0), group 1 and 2 are built following +X to -Y; from node (3, 0) to node (0, 3), group 3 and 4 are built following -X to +Y. Only the group masters send multicast packets to their members. They send one multicast packet every four packets. If a multicast packet is sent to $n$ members, the amount of traffic is counted as $n$ packets. The resultant multicast traffic takes 16.2% percent of the total network traffic.

We consider two cases. **Case 1**: the multicast groups do not reserve VCs (lanes); **Case 2**: the multicast groups reserve VCs. Figure 9 and Figure 10 draw the network per-

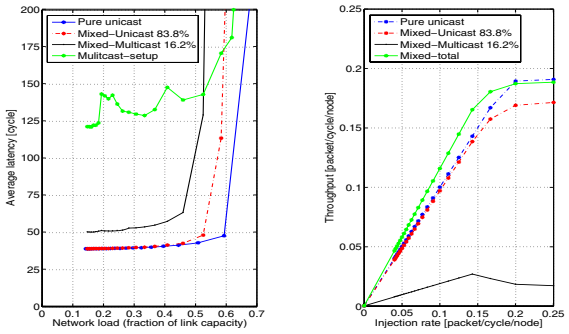



Figure 9: Performance without VC reservation

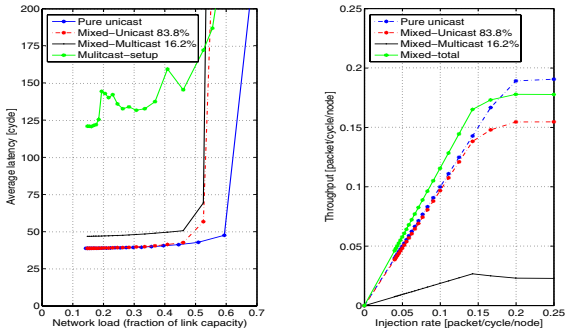



Figure 10: Performance with VC reservation

formance with the two scenarios for case 1 and case 2, respectively. In comparison with the purely unicast traffic, the unicast in the mixed traffic scenario performs equivalently to each other when the network load is below 0.45 for both cases. This means that the multicast traffic does not degrade the unicast performance if the network is not overloaded. The throughput is higher with the mixed traffic if the network is not saturated, since a multicast uses link bandwidth more efficiently. As shown in Figure 10, if a group reserves lanes, the average latency of the multicast traffic is improved 4.6 cycles on average if the network operates below load 0.45. However, due to lane reservation, the network saturation throughput is decreased by 4.2% from 0.185 to 0.177 packet/cycle/node.

As can be observed in Figure 9 and 10, the average group setup latency is 3-4 times as much as the average latency of unicast packets even when the network is not overloaded, since a group setup takes at least a round-trip time. This overhead suggests that our multicast scheme is beneficial to send block data where the amount of multicast traffic is high, if a multicast group is to be established dynamically. If a group is set up statically during the system warm-up phase, this overhead may be ameliorated.

# 5 Conclusions

We have presented our multicasting scheme in wormhole-switched networks on chip. With this scheme, multicasting starts after a multicast group is established. During establishment, a multicast group can reserve virtual channels. Our experimental results suggest that multicasting is beneficial in throughput. In addition, in a network with mixed unicast and multicast traffic, the multicast traffic does not show negative impact on the performance of unicast traffic if the network is not saturated. With the lane reservation, the latency of multicast traffic can be improved at the expense of slightly decreased throughput.

In future work, we aim at designing a synthesizable wormhole switch supporting the multicast scheme in order to obtain its cost overhead in area and speed penalty. The optimization of the controller will be essential for enhancing the hardware speed. Another direction is to use the QoS-aware multicast communication to emulate traditional buses. This could potentially address the problem of large amount of legacy code written for buses.

# References


[1] T. Bjerregaard and J. Sparso. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Proceedings of the Design, Automation and Test in Europe Conference*, volume 2, pages 1226–1231, 2005.

[2] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–204, March 1992.

[3] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufman Publishers, 2004.

[4] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Rădulescu. Deadlock prevention in the Æthereal protocol. In D. Borrione and W. Paul, editors, *Proc. Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science (LNCS)*, pages 345–348, Oct. 2005.

[5] K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):21–31, Sept-Oct 2005.

[6] X. Lin, P. K. McKinley, and L. M. Ni. Deadlock-free multicast wormhole routing in 2-d mesh multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):793–804, 1994.

[7] J. Liu, L.-R. Zheng, and H. Tenhunen. Interconnect intellectual property for network-on-chip. *Journal of System Architectures, Special issue on networks on chip*, 50(2):65–79, February 2004.

[8] Z. Lu and A. Jantsch. Flit ejection in on-chip wormhole-switched networks with virtual channels. In *Proceedings of the IEEE Norchip Conference*, November 2004.

[9] M. P. Malumbres, J. Duato, and J. Torrellas. An efficient implementation of tree-based multicast routing for distributed shared-memory multiprocessors. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, 1996.

[10] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the Design Automation and Test in Europe Conference*, 2004.

# Paper 4

**TDM virtual-circuit configuration in network-on-chip using logical networks**

In submission to *IEEE Transactions on Very Large Scale Integration Systems*.

# TDM Virtual-Circuit Configuration in Network-on-Chip Using Logical Networks

Zhonghai Lu, *Student Member, IEEE,* and Axel Jantsch, *Member, IEEE*

***Abstract*— Configuring Time-Division-Multiplexing (TDM) Virtual Circuits (VCs) on network-on-chip must guarantee conflict freedom for overlapping VCs besides allocating sufficient time slots to them. Using the generalized concept of a logical network, we develop and prove theorems that constitute sufficient and necessary conditions to establish conflict-free VCs. Moreover, we give a formulation of the multi-node VC configuration problem and suggest a back-tracking algorithm to find solutions by constructively searching the solution space.**

***Index Terms*— Time Division Multiplexing, Virtual Circuit, Logical Network, Quality of Service, Network-on-Chip**

## I. Introduction

TECHNOLOGY capacity and application complexity have driven bus-based system-on-chip (SoC) towards network-on-chip (NoC). As a global communication platform, signals are routed as packets via switches instead of being hard-wired. However, due to contention for shared links and buffers in the network, routing packets brings about unpredictable performance. To overcome the nondeterminism, researchers proposed various *resource reservation* and *priority* mechanisms to achieve Quality of Service (QoS), i.e., to provide guarantees in latency and bandwidth. The Æthereal [1] and Nostrum [2] NoCs establish *Time-Division-Multiplexing (TDM) virtual circuits (VCs)* to offer guaranteed services. The Æthereal VC, which is developed for a network using buffered flow control, is *open-ended*. The Nostrum VC, which is designed for a network employing bufferless flow control, is *closed-loop*. Both networks operate synchronously. Packets on a VC are consecutively transmitted in a pipeline by the nodes along the VC route. The Mango [3] NoC realizes guarantees in an asynchronous (clockless) network by preserving virtual channels for end-to-end connections and using priority-based scheduling in favor of connections in switches. Alternatively, QoS may be achieved through traffic classification using a differentiated service. For example, the QNoC [4] characterizes traffic into four priority classes, and switches make priority-based switching decisions.

In the paper, we address the multi-node TDM VC configuration problem. VC is a connection-oriented technique in which a deterministic path must be established and associated resources are pre-allocated before packet delivery can start. A TDM VC means that each node along the path configures a time-sliced routing table to reserve time slots for input packets to use output links. This reservation is accomplished in the connection setup phase. In this way, VCs multiplex link bandwidth in a time division fashion. As long as a VC is established, packets sent over it, called *VC packets*, encounter no contention and thus have guarantees in latency and bandwidth. In a network delivering both Best-Effort (BE) and guaranteed-service traffic, BE packets utilize resources that are not reserved by VCs. In case the requested resources are not available, the BE packets are buffered if the network uses drop-less buffered flow control. If the network realizes bufferless flow control without dropping packets, the packets are deflected to unfavored links. A VC is simplex, passing at least one source node and one destination node. But, in general, a VC may comprise multiple source and destination nodes (multi-node). In fact, it is important to construct multi-node VCs in order to use network bandwidth more efficiently. The VC configuration is an indispensable process in the application design flow because the establishment of VCs is the precondition for the guaranteed service. Besides, well-planned VCs (for instance, multi-node VCs and minimal routes) can make a better utilization of network resources and achieve better network performance. Configuring VCs involves (1) *path selection*: This has to explore the network path diversity. As a VC has a number of alternative paths, configuring a set of VCs involves an extremely large design space. The space is exponentially increased with the number of VCs; (2) *slot allocation*: Since VC packets can not contend with each other, VCs must be configured so that an output link of a switch is allocated to one VC per slot. Both steps together must ensure that VCs are contention free and equipped with sufficient slots. The network must be deadlock free and livelock free.

Current approaches to the TDM VC configuration problem can be found in [5] and [6]. These approaches have three major drawbacks: (1) the technique of contention avoidance is somewhat ad hoc. Contention is avoided mainly by locally scheduling available slots to a set of sorted VCs one by one, resulting in irregularly partitioned time slots. In case of lack of slots along a VC's shortest path, the VC uses a detour, i.e., non-minimal path. The traffic model assumes periodic messages and all message flows have the same period. To use flexible routing in a network, messages within a flow are scheduled individually and may use different routes [6]. Consequently, the message scheduling has also to ensure the correct message ordering; (2) the proposed heuristic algorithms are greedy, exploring only a very small subset of the solution space. In effect, this saves run time but wastes effectiveness. The heuristics limit the number of possible solutions, and for many problems, they can not find a solution or lead to over-dimensioning of the network; (3) a VC specification allows only one source node and one destination node. This makes the network utilization less efficient. Our work addresses these

The authors are with the Royal Institute of Technology, Sweden.

problems. For the first problem, we resort to a formal approach by utilizing the generalized concept of a *Logical Network (LN)* and developing theorems to guide the construction of conflict-free VCs. For the second problem, we use a standard technique to systematically search the solution space and constrain paths to minimal routes. For the third problem, we consider a VC specification with multiple source and destination nodes.

The rest of the paper is organized as follows. We outline the related work in Section II, detailing the weakness of the current approaches. In Section III, we describe the two types of TDM VCs proposed for networks on chips, namely, *open-ended* and *close-looped* VCs. Using logical networks to construct contention-free, bandwidth-satisfied VCs is exemplified in Section IV. Then we develop sufficient and necessary conditions and prove the theorems for overlapping VCs to be contention-free in Section V. Section VI formulates the multi-node VC configuration problem. In Section VII, we detail the multi-node VC configuration method including the backtracking algorithm. Experimental results are reported in Section VIII. Finally we conclude the paper in Section IX.

## II. Related Work

Researchers have formulated many sub-problems for NoC application design. Hu and Marculescu [7] formulated an IP-to-node mapping problem in presence of routing diversity. The mapping and path selection phases are decoupled. They used a branch-and-bound algorithm to optimize energy while exploiting minimal and deadlock-free routing functions. Murali et al. presented scheduling methods [8] to satisfy performance constraints of mapping multiple applications onto a single NoC. Srinivasan et al. [9] formulated the NoC topology synthesis problem as a linear programming problem, and proposed heuristics to improve the run time.

The TDM-based VC configuration is a general problem for reserving time slots to provide QoS in synchronous networks. As we mentioned previously, possible solutions can be found in [5] and [6]. The UMARS (Unified MApping, Routing and Slot allocation) [5] is a single-objective algorithm unifying the IP-to-node mapping, path selection and slot allocation. It is a greedy algorithm that iterates over a monotonically decreasing set of unmapped VCs until all VCs are allocated or until allocation fails. The outer loop of UMARS consists of three steps (1) select the VC with the highest bandwidth; (2) find a mapping and a path; (3) allocate slots on this path. The second step is guided by a cost function and finds *one locally optimal* path. Although it is fast, it leaves a wide solution space unexplored. Stuijt et al. analyzed the necessity of exploring wider solution space in [6]. Their experiments show that, if re-consideration of the path (path alternative) is allowed, the number of solvable problems in the synthetic benchmark set is improved 209%. However, the improved algorithm is still greedy allowing a limited number of path alternatives. They also demonstrated the significance of performing a proper contention avoidance strategy. Before scheduling message streams, they try to estimate the number of slots that are needed in each of the links. This knowledge of congestion on links is used to guide the path selection process. Together with a slot-sharing allocation strategy, this improves over UMARS by 334% with a small overhead on the run time. Both algorithms in [5] and [6] allow non-minimal routes to increase the solvable problem size. However, permitting detour consumes additional buffering and link bandwidth. It wastes power and may degrade the overall system performance because it affects other traffic in the network. In our opinion, detour can only be justified if the solution space is sufficiently searched. This is particularly true if VC configurations are derived off-line.

## III. TDM-based Virtual Circuits in NoC

### A. Open-ended VCs

The TDM VCs on NoCs assume that the network is synchronously clocked, thus all nodes share the same notion of time. VC packets synchronously advance one step per time slot. Since a VC packet encounters no contention, it never stalls. A node must configure a *routing table for VC packets* such that no simultaneous use of shared resources is possible. The routing table, by configuration, knows the *time slot* when a VC packet reaches which inport, and *addressing information* about which outport to use. In effect, the routing table partitions the link bandwidth and avoids contention. In implementation, the routing tables may be implemented in the switches. In this case, A VC packet does not need to carry routing information in its head field. Alternatively, the slot tables can be removed from the switches, and the information is embedded in a VC packet. This saves area at the expense of communication overhead in each VC packet.

Figure 1 shows two VCs, $v_1$ and $v_2$, and the respective routing tables for the switches. The output links of a switch are associated with a buffer or register. A routing table $(t, in, out)$ is equivalent to a routing function $\mathcal{R}(t, in) = out$, where $t$ is time slot, *in* an input link, and *out* an output link. $v_1$ passes switches $sw_1$ and $sw_2$ through $\{b_1 \rightarrow b_2\}$; $v_2$ passes switches $sw_3$ and $sw_2$ through $\{b_3 \rightarrow b_2\}$. The Æthereal NoC [1] proposes this type of VC for QoS. Since the path of such a VC is not a loop, we call it an open-ended VC.

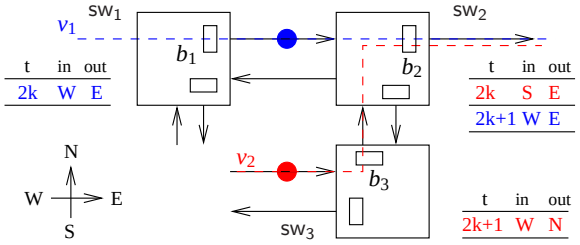



Fig. 1. Open-ended virtual circuits

In open-ended VCs, packets may be partitioned into *admission classes* [10] by the slots they are injected into the network. Formally, the admission class of a packet admitted at time $t_0$ with initial destination distance $i(t_0)$ is defined as $mod(t_0 + i(t_0), D)$, where $D \in \mathbb{N}$. As an admission class owns dedicated slots, packets of different classes do not collide in buffers. By globally orchestrating the packet admission, contention can be avoided for packets belonging to different

VCs. As illustrated in Figure 1, $v_1$ and $v_2$ only overlap in $b_2$, denoted $v_1 \cap v_2 = \{b_2\}$. $v_1$ packets are admitted on even slots of $b_1$. In $sw_1$, $(2k, W, E)$ means that $sw_1$ reserves its $E$ (East) output link at slots $2k$ ($k \in \mathbb{N}$) for its $W$ (West) inport ($\mathcal{R}(2k, W) = E$). As we can also see, $v_2$ packets are admitted on odd slots $2k+1$ of $b_3$, and $sw_3$ configures its odd slots for $v_2$. Since a $v_1$ packet reaches $sw_2$ one slot after reaching $sw_1$, $sw_2$ assigns its odd slots to $v_1$. Similarly, $sw_2$ allocates its even slots to $v_2$. As $v_1$ and $v_2$ interleave on the use of the shared buffer $b_2$ and its associated output link, $v_1$ and $v_2$ do not conflict.

### B. Closed-loop VCs with Containers

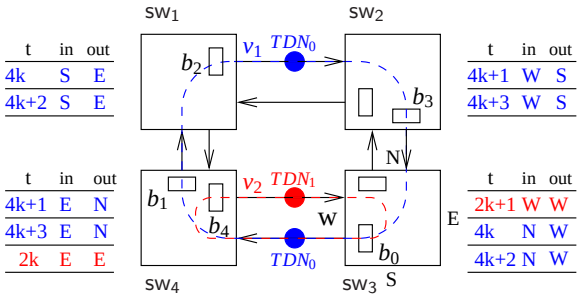



Fig. 2. Closed-loop virtual circuits

The Nostrum NoC [2] also suggests a TDM VC for QoS. However, a Nostrum VC has a cyclic path, i.e., a closed loop. On the loop, at least one *container* is rotated. A container is a *special packet* used to carry data packets, like a vehicle carrying passengers. The reason to have a loop is due to the fact that Nostrum uses deflection routing [11] whereas switches have no buffer queues. All packets are on the run cycle-by-cycle. Since all outgoing links of a switch might be occupied by all incoming packets, a looped container ensures that there is an output link available for locally admitting a VC packet into the container, thus the network. VC packets are loaded into the container from a source, and copied (for multicast) or unloaded at the destination, by-passing other switches. Similarly to open-ended VCs, containers as VC packet carriers have higher priority than BE packets and do not contend with each other.

The Nostrum VC [2] uses the concept of a *Temporally Disjoint Network* (*TDN*) to ensure conflict freedom. TDNs are independent of VC paths. They are globally set up in a network. The number of TDNs depends on the network topology and the buffer stages in the switches. For example, in a mesh network with one buffer per outport in the switches, exactly two TDNs exist, $TDN_0$ and $TDN_1$. As shown in Figure 2, two VCs, $v_1$ and $v_2$, are configured. $v_1$ loops on $sw_3$, $sw_4$, $sw_1$ and $sw_2$ through $\{b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_0\}$; $v_2$ loops on $sw_3$ and $sw_4$ through $\{b_0 \rightarrow b_4 \rightarrow b_0\}$; and $v_1 \cap v_2 = \{b_0\}$. $v_1$ and $v_2$ subscribe to $TDN_0$ and $TDN_1$, respectively. Besides, $v_1$ launches two containers and $v_2$ one container. The resulting routing tables for switches are also shown in Figure 2. Since TDNs are temporally disjoint, overlapping VCs allocated on different TDNs are free from conflict.

## IV. VC Configuration using Logical Networks

### A. An Overview of Multi-node VC configuration

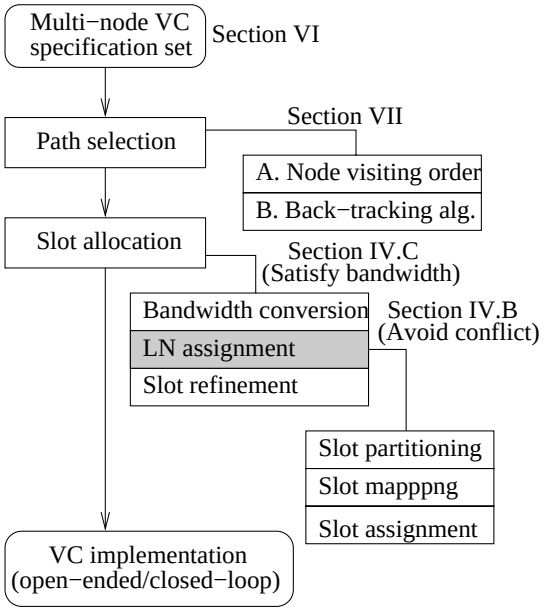



Fig. 3. An overview of the multi-node VC configuration method

Figure 3 sketches our multi-node VC configuration approach. The problem formulation is given in Section VI. We address both *path selection* and *slot allocation* in the paper. For path selection, we first discuss the node visiting order in a multi-node VC. This problem arises when a VC specification contains multiple source and destination nodes and only minimal routes are desired. We use a back-tracking algorithm to explore the diverse paths of VCs. The path selection is addressed in Section VII. But, our major contribution is to use the generalized concept of a Logical Network (LN) to perform slot allocation. This involves *avoiding conflict between overlapping VCs* and *satisfying VC bandwidth requirements*. Therefore, we first exemplify our LN-oriented slot allocation method in this section. Then, formal definitions and proofs for the conflict freedom are presented in Section V.

In the following of this section, assuming that the path selection is done, we give a loose definition of LN and exemplify the LN construction for conflict freedom in Section IV-B. Then we show how to satisfy bandwidth demand using LNs in Section IV-C. We shall see that our method is applicable to both *open-ended* and *closed-loop* VCs.

### B. Avoid Conflict using Logical Networks (LNs)

Both *admission classes* and *TDNs* are essentially *logical networks* (*LNs*). A logical network is a composition of associated *(time slot, buffer)* pairs. Contention is avoided since VCs inject packets on dedicated LNs. To be specific, the conflict avoidance is assured through three steps: *slot partitioning*, *slot mapping* and *slot assignment*. The first two steps partition slots into logical networks. The last step assigns sets of (time slot, buffer) pairs, thus logical networks, to virtual circuits. We

describe the three steps with a pair of *closed-loop VCs* $(v_1, v_2)$ in Figure 2.

1) *Slot partitioning*: As conflicts might occur in a shared buffer, we partition the slots of the shared buffer into slot sets with a regular interval. In Figure 2, $b_0$ is the only shared buffer of $v_1$ and $v_2$, $v_1 \cap v_2 = \{b_0\}$. We partition the slots of $b_0$ ($b_0$ is called the *reference buffer* for $v_1$ and $v_2$, $Ref(v_1, v_2) = b_0$.) into two sets, an even set $s_0^2(b_0)$ for $t = 2k$ and an odd set $s_1^2(b_0)$ for $t = 2k+1$. The notation $s_\tau^T(b_0)$ represents pairs $(\tau + kT, b_0)$, which is the $\tau$th slot set of the total $T$ slot sets, $\tau \in [0, T)$ and $T \in \mathbb{N}$. The pair $(t, b_0)$ refers to the slot of $b_0$ at time instant $t$.

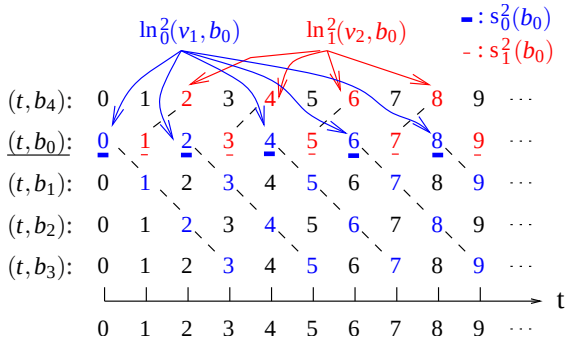



Fig. 4. Creating logical networks by mapping slots on VCs

2) *Slot mapping*: The partitioned slot sets can be mapped to slot sets of other buffers on a VC regularly and unambiguously because a VC packet or container advances one step each and every slot. For example, a $v_1$ packet holding slot $t$ at buffer $b_0$, i.e., pair $(t, b_0)$ will consecutively take slot $t+1$ at $b_1$ (pair $(t+1, b_1)$), slot $t+2$ at $b_2$ (pair $(t+2, b_2)$), and slot $t+3$ at $b_3$ (pair $(t+3, b_3)$). After mapping the slot set $s_0^2(b_0)$ on $v_1$ and $s_1^2(b_0)$ on $v_2$, we obtain two slot sets $\{s_0^2(b_0), s_1^2(b_1), s_0^2(b_2), s_1^2(b_3)\}$ and $\{s_1^2(b_0), s_0^2(b_4)\}$. We refer to the logically networked slot sets in a set of buffers of a VC as a *logical network (LN)*. We denote the two LNs as $ln_0^2(v_1, b_0)$ and $ln_1^2(v_2, b_0)$, respectively. The notation $ln_\tau^T(v, b)$ represents the $\tau$th LN of the total $T$ LNs on $v$ with respect to $b$. We illustrate the mapped slot sets for $s_0^2(b_0)$ and $s_1^2(b_0)$ and the resulting LNs in Figure 4.
3) *Slot assignment*: We assign $v_1$ and $v_2$ to different LNs, specifically, $ln_0^2(v_1, b_0)$ to $v_1$, and $ln_1^2(v_2, b_0)$ to $v_2$.

As $ln_0^2(v_1, b_0) \cap ln_1^2(v_2, b_0) = \emptyset$, $v_1$ and $v_2$ are conflict free, as we shall show formally in Section V.

### C. Satisfy Bandwidth using Logical Networks (LNs)

| $v$ | *Buf. set* | *bw* | $N$ | $W$ | LN | Slot set |
|---|---|---|---|---|---|---|
| $v_1$ | $b_1, b_2, b_3$ | 1/3 | 2 | 6 | $ln_0^2(v_1, b_1)$ | $s_{0,2}^6(b_1)$, $s_{1,3}^6(b_2)$, $s_{2,4}^6(b_3)$ |
| $v_2$ | $b_1, b_4$ | 1/4 | 1 | 4 | $ln_1^2(v_2, b_1)$ | $s_1^4(b_1)$, $s_0^4(b_4)$ |
| $v_3$ | $b_2, b_3$ | 3/8 | 3 | 8 | $ln_0^2(v_3, b_2)$ | $s_{0,2,4}^8(b_2)$, $s_{1,3,5}^8(b_3)$ |

TABLE I
VC PARAMETERS AND LN ASSIGNMENT RESULTS FOR FIG. 5

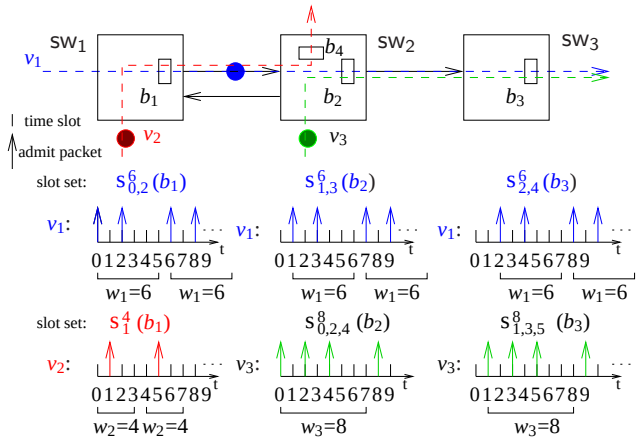



Fig. 5. Packets admitted on slot sets or logical networks

In addition to be contention free, VCs must satisfy their bandwidth requirements. This may be achieved through three steps: *bandwidth conversion*, *LN assignment* and *slot refinement*. We exemplify the three steps with Figure 5 that shows three *open-ended VCs*, $v_1$, $v_2$ and $v_3$. As can be seen, $v_1 \cap v_2 = \{b_1\}$, $v_1 \cap v_3 = \{b_2, b_3\}$ and $v_2 \cap v_3 = \emptyset$.

1) *Bandwidth conversion*: A VC is associated with a bandwidth requirement in bits/second, which can be directly translated into packets/cycle. As bandwidth is an average measurement, we can further scale it to the number of packets $N$ per window $W$ cycles. For example, we translate $bw_1 = 1/3$ into $2/6$ (2 packets every 6 cycles), i.e., $N_1 = 2$, $W_1 = 6$, as listed in Table I. Note that a window defines the pattern of a regular packet flow. But it is *not* a period since more than one packet (up to the window size) may be sent during a single window.
2) *LN assignment*: In this step, we assign VCs to LNs using the three steps for *conflict avoidance* in Section IV-B. Additionally we must check whether bandwidth demand can be satisfied. This check is conducted after the first step *slot partitioning*. Given a pair of overlapping VCs, the number $T$ of partitioned sets with respect to the reference buffer equals the number of LNs. To satisfy the bandwidth requirement of a VC $v$, a sufficient number $N_{ln}$ of LNs must be allocated to $v$. This number can be derived from $N_{ln} = \lceil NT/W \rceil$ [1], because we must satisfy $N_{ln}/T \geq N/W$, where $N_{ln}/T$ is the bandwidth supported by the allocated LNs and $N/W$ the requested bandwidth. The bandwidth requirements of the three VCs in Figure 5 are given in column *bw* of Table I.
We first perform the LN assignment with VC pair $(v_1, v_2)$. Since $b_1$ is the only shared buffer of $(v_1, v_2)$, $Ref(v_1, v_2) = b_1$. Let $T = 2$, we partition $b_1$'s slots into odd and even sets, implying two LNs. Either VC can be allocated to one LN, i.e., $N_{ln,1} = N_{ln,2} = 1$, offering bandwidth $N_{ln,1}/T = N_{ln,2}/T = 1/2$. Since the bandwidth demand of $v_1$ and $v_2$ is less then $1/2$, the resulting LN assignment will meet the bandwidth constraint. Then we can continue to map the even set on $v_1$ and the odd set on

[1] $\lceil x \rceil$ is the ceiling function that returns the least integer not less than $x$.

$v_2$, obtaining the even LN $ln_0^2(v_1,b_1)$ for $v_1$ and the odd LN $ln_1^2(v_2,b_1)$ for $v_2$. Since $ln_0^2(v_1,b_1) \cap ln_1^2(v_2,b_1) = \emptyset$, $v_1$ and $v_2$ are conflict free.

Next, we perform the LN assignment with VC pair $(v_1,v_3)$. Let the reference buffer of $v_1$ and $v_3$ be $b_2$, $Ref(v_1,v_3) = b_2$. Since $v_1$ already holds even slots in $b_1$, it takes odd slots in $b_2$, i.e., $s_1^2(b_2)$. We assign the remaining even slots in $b_2$, i.e., $s_0^2(b_2)$, to $v_3$. Therefore, $N_{ln,3}/T = 1/2 > 3/8$. We are certain that the supported bandwidth suffices the demand of $v_3$. We map the slot set $s_0^2(b_2)$ on $v_3$, obtaining $ln_0^2(v_3,b_2)$. As $ln_0^2(v_1,b_1) \cap ln_1^2(v_3,b_2) = \emptyset$, $v_1$ and $v_3$ are also conflict free. The LN assignments are shown in column LN of Table I.

3) *Slot refinement*: The success of LN assignment for all VCs means that all VCs are conflict free and enough bandwidth can be reserved. But, a VC may demand only a fraction of slot sets from its assigned LNs. For instance, "$v_2$ on $ln_1^2(v_2,b_1)$" means that $v_2$ can use one of every two slots. But $N_2 = 1$ and $W_2 = 4$, $v_2$ actually demands only one slot every four slots. This means that we need to further refine the supplied bandwidth. We first find the candidate slot sets of a reference buffer and then only assign $N$ of them within window size $W$ to $v$. For example, $v_3$ has four candidate slot sets over $b_2$, $s_{0,2,4,6}^8(b_2)$. We allocate any three of the four to $v_3$, for instance, $s_{0,2,4}^8(b_2)$. These slot sets mapped to $s_{1,3,5}^8(b_3)$, forming the LN $ln_0^2(v_3,b_2)$. The slot sets reserved by the three VCs are illustrated in Figure 5 and listed in the column Slot set of Table I.

After the three steps above, the VCs are constructed without conflict and with bandwidth requirements satisfied.

### D. Problems with LN-oriented VC Configuration

We have described so far three techniques: (1) establishing VCs by configuring slot-sliced routing tables; (2) partitioning and mapping slots into LNs; (3) assigning different LNs to VCs. These techniques must promise conflict freedom and provide enough bandwidth. However, there are several key questions that are not yet addressed:

- How many LNs exist when VCs overlap? LN is not global for all VCs. Instead it is local for a set of overlapping VCs. This number is crucial because it defines how to partition and then map slots.
- In the examples, assigning different LNs to overlapping VCs has secured conflict freedom. Is it a sufficient and necessary condition, in general?
- LN is partitioned with respect to a reference buffer, which is a shared buffer. As overlapping VCs may have many shared buffers, how is this reference buffer selected? Are LNs with respect to all shared buffers equivalent?

In the next section, we answer these questions formally.

## V. Formal Analysis

### A. Assumptions and Definitions

We consider static VCs, meaning that VCs do not change their paths and characteristics throughout system execution. We also assume that one LN is allocated to only one VC. But one VC may subscribe to multiple LNs.

*Definition 1:* A VC $v$ comprises an ordered set of buffers $< b_0, b_1, b_2, \cdots, b_{H-1} >$. The size of $v$, denoted $|v|$, is the number of buffers, $H$. $d_{b_i\vec{b}_j}$ is the distance (number of steps) from $b_i$ to $b_j$. On $v$, $d_{b_i\vec{b}_{i+1}} = 1$, meaning that the buffers are adjacent.

A VC packet or container $p$ on $v$, after initial delay $\tau'$ slots, starts from $b_i$ and visits each buffer in sequence one per time slot and never stalls. The *distance* $d'$ of $p$ to buffer $b_j$ equals $\tau' + d_{b_i\vec{b}_j}$.

*Definition 2:* The *packet-admission pattern* on a VC requires that $N$ packets are admitted in a sequence of $D$ ($D \geq N$) time slots. This gives a bandwidth requirement of $N/D$ packets/cycle, but the exact time slots for admitting the $N$ packets are not specified. A *packet flow* is defined by infinitely repeating the packet-admission pattern. We call $D$ the *admission cycle*. With respect to a buffer $b$, we define a *packet-admission class* as an infinite set of packets admitted on modulo $D$ slots, $d + kD$, $k \in \mathbb{N}$, where $d$ is the *initial distance* of the admission class to buffer $b$, i.e., the distance of the first packet on the admission class to buffer $b$.

For an open-ended VC, $D = W$, where $W$ is the window size of a VC packet flow. For a closed-loop VC, $D = H$, since $v$ is a loop and a container revisits the same buffer after $H$ slots. $N$ is the number of containers launched on the VC.

*Definition 3:* Two VCs $v_1$ and $v_2$ *overlap* if they share at least one buffer, i.e., $v_1 \cap v_2 \neq \emptyset$. The two VCs *conflict* in buffer $b$, denoted $b \in v_1 \wedge v_2$, if and only if it is possible that two packets, one from each VC, visit buffer $b$ at the same time. $v_1 \wedge v_2 = \emptyset$ means that $v_1$ and $v_2$ are conflict free.

*Definition 4:* Given a VC $v = < b_0, b_1, b_2, \cdots, b_{H-1} >$, $b_i \in v$, a natural $T \geq 1$ and a natural $\tau$, $0 \leq \tau < T$, we define a *logical network (LN)* $ln_\tau^T(v,b_i)$ as an infinite set of *(time slot, buffer)* pairs as follows:

$$ln_\tau^T(v,b_i) = \{(t,b_j) | t = \tau + d_{b_i\vec{b}_j} + kT, 0 \leq j < H, k \in \mathbb{N}\}$$

Hence, a LN is defined for a given VC and one of its buffers. The number of LNs for a VC is always equal to $T$. The motivation of the LN is to precisely define the flow of packets on the VC and each admission class is dedicated to exactly one LN. The time when packets visit buffers of the VC is given by the *(time slot, buffer)* pairs of the LN. On a LN, every $T$ cycles a packet visits a particular buffer. Consequently, the bandwidth possessed by a LN is $1/T$ packets/cycle.

The LNs of a VC has an inherent property: if $\tau_1, \tau_2 \in [0, T-1]$ and $\tau_1 \neq \tau_2$, then packets admitted on different LNs never collide, because

$$ln_{\tau_1}^T(v,b) \cap ln_{\tau_2}^T(v,b) = \emptyset$$

*Definition 5:* A LN-cover is a complete set of LNs defined for a VC $v$ with respect to a buffer $b_i$, $b_i \in v$,

$$LN\text{–}cover(v,b_i,T) = \{ln_\tau^T(v,b_i) \mid 0 \leq \tau < T\}$$

*Definition 6:* VC-to-LN assignment/subscription: a VC $v$ is assigned to or subscribes to $ln_\tau^T(v,b_i)$ if and only if, on $v$,

an admission class, which has an initial distance $d$ to buffer $b_i$ and the admission cycle $D$, satisfies $mod(d+kD,T)=\tau$, $k\in\mathbb{N}$.

If a VC $v$ does not overlap with any other VCs, the maximum number of LNs on $v$ is $D$, since $v$ allows for up to $D$ admission classes and one class uses exactly one LN.

### B. Overlapping VCs

*Lemma 1:* Let $v_1$ and $v_2$ be two overlapping VCs. $D_1, D_2$ be their admission cycles, respectively. Let $c_1$ and $c_2$ be any two admission classes on $v_1$ and $v_2$, respectively; $d_1$ and $d_2$ are the initial distances of $c_1$ and $c_2$ to a shared buffer $b$, respectively. We have $b\in v_1\wedge v_2$ iff $\exists\, k_1,k_2\in\mathbb{N}$ such that $d_1+k_1D_1=d_2+k_2D_2$.

*Proof:*

(1) Sufficient: We assume that $\exists k_1,k_2\in\mathbb{N}$ such that $d_1+k_1D_1=d_2+k_2D_2(=t)$. The left-hand side of the equation implies that $c_1$ enters buffer $b$ at time slot $t$, and the right-hand side implies that $c_2$ enters $b$ the same slot. Hence $b\in v_1\wedge v_2$.

(2) Necessary: Suppose, after $t$ slots, $v_1$ and $v_2$ collide in buffer $b$, $b\in v_1\wedge v_2$. For $c_1$, $t=d_1+k_1D_1$; for $c_2$, $t=d_2+k_2D_2$. Therefore $d_1+k_1D_1=d_2+k_2D_2$. ■

*Theorem 1:* The number $T$ of LNs, which two overlapping VCs, $v_1$ and $v_2$, can subscribe to without conflict, is a Common Factor (CF) of the admission cycles of both VCs, $D_1$ and $D_2$.

*Proof:* Suppose that $b$ is the reference buffer.

Let $ln^T_{\tau_1}(v_1,b)$ and $\ln^T_{\tau_2}(v_2,b)$ be the LN subscribed by $v_1$ and $v_2$, respectively. According to Definition 6, we have $\tau_1=mod(d_1+k_1D_1,T)$ and $\tau_2=mod(d_2+k_2D_2,T)$.

For $\tau_1=mod(d_1+k_1D_1,T)\ \forall k_1\in\mathbb{N}$. When $k_1=0$, $d_1=k_1'T+\tau_1$; when $k_1=1$, $d_1+D_1=k_1''T+\tau_1$ and $k_1''>k_1'$. From the last two equations, we get $D_1=(k_1''-k_1')T$. This means that $T$ is a factor of $D_1$.

Similarly, using $\tau_2=mod(d_2+k_2D_2,T)$, $\forall k_2\in\mathbb{N}$, we can derive that $T$ is a factor of $D_2$.

Therefore $T$ is a common factor of $D_1$ and $D_2$, i.e., $T\in CF(D_1,D_2)$. ■

By Theorem 1, the number $T$ of LNs for $v_1$ and $v_2$ can be any value in the common factor set $CF(D_1,D_2)$. The least number of LNs is 1. However, if the number of LNs for two VCs is 1, only one of the two VCs can subscribe to it. There is no room for the other VC. Therefore we need at least two LNs. In general, if $n$ VCs overlap in a shared buffer, there must be at least $n$ LNs, one for each VC, to avoid conflict. In order to maximize the number of options and have finer LN bandwidth granularity, we consider the number $T$ of LNs to be the Greatest Common Divisor (GCD) throughout the paper. Hence, for the two overlapping VCs, $v_1$ and $v_2$, the number $T$ of LNs equals the $GCD(D_1,D_2)$.

*Theorem 2:* Assigning $v_1$ and $v_2$ to different LNs with respect to any shared buffer is a sufficient and necessary condition to avoid conflict between $v_1$ and $v_2$.

*Proof:* By Theorem 1, the maximum number $T$ of LNs for $v_1$ and $v_2$ is $T=GCD(D_1,D_2)$. We can write $D_1=A_1T$ and $D_2=A_2T$, where $A_1$ and $A_2$ are co-prime.

By Definition 6, $v_1$ and $v_2$ subscribe to different LNs $\Leftrightarrow mod(d_1+k_1D_1,T)\neq mod(d_2+k_2D_2,T)$. Since $D_1=A_1T$ and $D_2=A_2T$, $mod(d_1+k_1D_1,T)\neq mod(d_2+k_2D_2,T)\Leftrightarrow mod(d_1,T)\neq mod(d_2,T)$.

(1) Sufficient: $mod(d_1,T)\neq mod(d_2,T)\Rightarrow d_1+k_1'T\neq d_2+k_2'T$, $\forall k_1',k_2'\in\mathbb{N}$. When $k_1'=k_1A_1$ and $k_2'=k_2A_2\ \forall k_1,k_2\in\mathbb{N}\Rightarrow d_1+k_1A_1T\neq d_2+k_2A_2T\Rightarrow d_1+k_1D_1\neq d_2+k_2D_2$. According to Lemma 1, $v_1$ and $v_2$ do not conflict, i.e., $v_1\wedge v_2=\emptyset$.

(2) Necessary: Suppose $v_1\wedge v_2=\emptyset\ \Rightarrow\ d_1+k_1D_1\neq d_2+k_2D_2$, $\forall k_1,k_2\in\mathbb{N}$. But let us assume $mod(d_1,T)=mod(d_2,T)$. Then we have $d_1-d_2\neq k_2D_2-k_1D_1$ but $d_1-d_2=kT$, $k\in\mathbb{Z}$. $\Rightarrow\ k+k_1A_1\neq k_2A_2\ \forall k_1,k_2\in\mathbb{N}$. However, this inequality is not always true, for example, when $k_1=A_2$; $k_2=A_1+1$; $k=A_2$. Thus, our assumption cannot be true, and $mod(d_1,T)\neq mod(d_2,T)$. This means that $v_1$ and $v_2$ subscribe to different LNs. ■

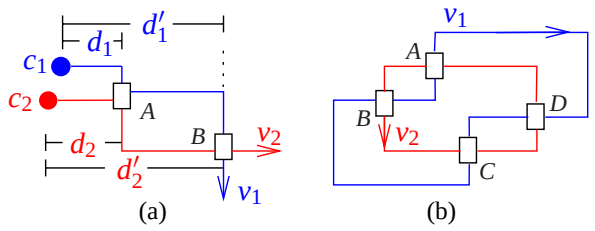



Fig. 6. Two or multiple shared buffers

By Theorem 2, VCs must stay in different LNs referring to *any shared buffer*. However, as overlapping VCs may have multiple shared buffers, LN partitioning might change with a different reference buffer. Figure 6a shows that two open-ended VCs, $v_1$ and $v_2$, overlap in buffers $A$ and $B$. Apparently, no conflict with respect to buffer $A$ does not imply no conflict with respect to another buffer $B$. We derive the following theorem to check the *reference consistency*.

*Theorem 3:* Suppose that two overlapping VCs, $v_1$ and $v_2$, have two shared buffers $A$ and $B$. Let the distances from buffer $A$ to $B$ along $v_1$ and $v_2$ be $d_{\vec{AB}}(v_1)$ and $d_{\vec{AB}}(v_2)$, respectively. Let the initial distance of $c_1$ to $A$ be $d_1$, to $B$ be $d_1'$; from $c_2$ to $A$ be $d_2$, to $B$ be $d_2'$. Assume that $c_1$ on $v_1$ and $c_2$ on $v_2$ do not conflict in $A$, then $d_{\vec{AB}}(v_1)-d_{\vec{AB}}(v_2)=kT$, where $T=GCD(D_1,D_2)$ and $k\in\mathbb{Z}$, is a sufficient and necessary condition for $c_1$ and $c_2$ to be conflict-free with respect to $B$. If so, we say the two shared buffers *consistent*.

*Proof:* As $d_{\vec{AB}}(v_1)=d_1'-d_1$ and $d_{\vec{AB}}(v_2)=d_2'-d_2$, $\Rightarrow d_{\vec{AB}}(v_1)-d_{\vec{AB}}(v_2)=(d_1'-d_2')-(d_1-d_2)$. Further, $d_{\vec{AB}}(v_1)-d_{\vec{AB}}(v_2)=kT\ \Leftrightarrow\ mod(d_1'-d_2',T)=mod(d_1-d_2,T)$. Condition $mod(d_1,T)\neq mod(d_2,T)\ \Leftrightarrow\ mod(d_1',T)\neq mod(d_2',T)$. Thus $c_1$ and $c_2$ are conflict free with respect to $B$. ■

By Theorem 3, we can further conclude that if two VCs have multiple shared buffers, all shared buffers must be consistent in order to be conflict-free. For instance, as shown in Figure 6b, if the two closed-loop VCs, $v_1$ and $v_2$, have no conflict, then all shared buffers $v_1\cap v_2=\{A,B,C,D\}$ must be consistent. If the consistency is checked *pair-wise*, the total number of checking times is $C_u^2=u(u-1)/2$, where $u$ is the number of shared buffers. However, the check can be done efficiently.

*Theorem 4:* Suppose that $v_1$ and $v_2$ have at least three shared buffers $A,B,C\in v_1\cap v_2$. If $A$ and $B$, and $B$ and $C$ are

consistent, then $A$ and $C$ are consistent.

*Proof:* As $A$ and $B$ are consistent, $d_{\vec{AB}}(v_1) - d_{\vec{AB}}(v_2) = k_1 T$. As $A$ and $C$ are consistent, $d_{\vec{AC}}(v_1) - d_{\vec{AC}}(v_2) = k_2 T$. By deducting the two equations, we have, $d_{\vec{AB}}(v_1) - d_{\vec{AB}}(v_2) - (d_{\vec{AC}}(v_1) - d_{\vec{AC}}(v_2)) = (k_1 - k_2)T$. Further, we have $d_{\vec{BC}}(v_1) - d_{\vec{BC}}(v_2) = k_3 T$ $k_3 \in \mathbb{Z}$. According to Theorem 3, $B$ and $C$ are consistent. ■

By Theorem 4, reference consistency may be linearly checked. As a result, the total number of checking times is reduced to $u-1$. If all shared buffers are consistent, any shared buffer can be used as a *reference buffer* to conduct LN partitioning and assignment. If they are not consistent, $v_1$ and $v_2$ conflict.

## VI. The Problem Formulation

We introduce additional definitions, and then define the multi-node VC configuration problem.

### A. Definitions

*Definition 7:* A network is a directed graph $\mathcal{G} = M \times E$, where each vertex $n_i \in M$ represents a node, and each edge (link) $e_i \in E$ represents a link. On $\mathcal{G}$, if there is an edge directing from one node to another, the edge is unique.

*Definition 8:* A *VC specification set* $\bar{V}$ comprises a set of VCs to be configured on the network $\mathcal{G}$. Each VC $\bar{v_i} \in \bar{V}$ is associated with:

- $m_i \subseteq M$: a subset of nodes in $M$ to be visited by $\bar{v_i}$. The node set is not necessarily ordered and two consecutive nodes in $m_i$ do not have to be adjacent in the network.
- $\bar{bw_i}$: minimum bandwidth requirement (bits/second) of $\bar{v_i}$.

*Definition 9:* A *VC implementation set* $V$ captures implementation options of $\bar{V}$ on the network $\mathcal{G}$. Each VC implementation $v_i \in V$, which implements $\bar{v_i}$, is associated with:

- $P_i$: a set of candidate shortest-distance paths which $v_i$ may travel. A shortest path $p_i \in P_i$ is expressed by a set of ordered and *adjacent* nodes on $\mathcal{G}$, since a pair of two adjacent nodes on $\mathcal{G}$ implies exactly the directed link connecting them. $\forall p_i \in P_i$, $m_i \subseteq p_i$.
- $BW_i$: a set of supported bandwidth (bits/second) of $v_i$. $bw_i \in BW_i$ is the bandwidth of implementation $v_i$ taking path option $p_i$.
- $R_{i,j}$: a partial routing table created for a visiting node $n_j$ by $v_i$. $\forall r_{i,j} \in R_{i,j}$, $r_{i,j}$ is an entry $(t, e_{in,i}, e_{out,i})$, specifying that node $n_j$ reserves slot $t$ for a $v_i$ packet from input link $e_{in,i}$ to use output link $e_{out,i}$. $R_j$ is the routing table of $n_j$, and $R_j = \sum_i R_{i,j}$.

*Definition 10:* On a network $\mathcal{G}$, a *path* function: $\mathcal{P} : M \rightarrow E$ maps $m_i \subseteq M$ to *one* path $p_i \in P_i$.

*Definition 11:* At node $n_j$, a *routing* function $\mathcal{R}_j : (\mathcal{T}, E_{in,j}) \rightarrow E_{out,j}$ maps an $e_{in,j} \in E_{in,j}$ to an $e_{out,j} \in E_{out,j}$ for slot $t \in \mathcal{T}$.

### B. The Problem

Using the definitions above, we formulate the problem as follows:

*Given* a network $\mathcal{G}$ and a VC specification set $\bar{V}$, *find* a VC implementation set $V$ and *determine* from $V$ (1) a *path function* $\mathcal{P}()$ and (2) a *routing function* $\mathcal{R}_j()$ for each node $n_j$, such that

$$\forall e_{in,i} \neq e_{in,k}, \mathcal{R}_j(t, e_{in,i}) \neq \mathcal{R}_j(t, e_{in,k}) \quad (1)$$

$$\bar{bw_i} \leq bw_i \quad (2)$$

$$\forall \text{ edge } e_k,\ Bw(e_k) \leq \kappa_{bw}(e_k) \quad (3)$$
$$\text{where } Bw(e_k) = \textstyle\sum_i bw_i \text{ if } e_k \in Edge(p_i)$$

Condition (1) says that VC packets from two different input links of a switch can not be switched to the same output link simultaneously, i.e., VCs must be set up without conflict. Condition (2) expresses that each VC's bandwidth constraint must be satisfied. Condition (3) means that the total normalized (with the link capacity) bandwidth reserved by all VCs on a link cannot exceed the link bandwidth threshold $\kappa_{bw}$, which is defined in terms of the link capacity and $0 \leq \kappa_{bw} \leq 1$. $\kappa_{bw}$ can be set for each link. If $0 \leq \kappa_{bw} < 1$ for a link, this means the link has room for routing BE traffic.

## VII. The Multi-Node VC Configuration Method

### A. The Node Visiting Order of a VC

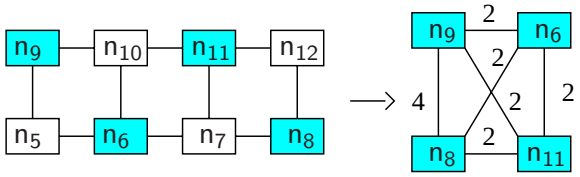



Fig. 7. The node set of a VC specification and its subgraph

A VC specification consists of a set of nodes, which may be un-ordered. To visit the nodes in shortest distance, we must first order them into a sequence, called a *tour*. We constrain that a node appears exactly once in the sequence, implying that we do not consider *forked* shortest paths. For each VC, its node set can be used to construct a subgraph, where an edge is weighted by the shortest distance between two nodes. Figure 7 exemplifies the problem. On the partial mesh, a VC specification has a node set $\{n_{11}, n_6, n_8, n_9\}$. The subgraph is drawn on the right side by labeling the edge with the shortest distance between a pair of nodes. If the VC implementation is open-ended, the visiting order following $\{n_{11} \rightarrow n_6 \rightarrow n_8 \rightarrow n_9\}$ requires 8 hops. A re-ordering of the nodes into $\{n_{11} \rightarrow n_8 \rightarrow n_6 \rightarrow n_9\}$ leads to 6 hops. This is a shortest path. If the VC implementation is a closed loop, however, this is not shortest. If the loop visits nodes in order $\{n_9 \rightarrow n_6 \rightarrow n_{11} \rightarrow n_8 \rightarrow n_9\}$, the traveling distance is 10. If the loop takes the order $\{n_9 \rightarrow n_{11} \rightarrow n_8 \rightarrow n_6 \rightarrow n_9\}$, the traveling distance is 8. Therefore, for each VC, we need to order the node set so that the traveling distance is shortest. It turns out that, for open-ended VCs, finding a tour visiting each node exactly once is the Hamiltonian Path Problem (HPP) [12]. We use the randomized algorithm to solve it. For closed-loop VCs, finding a tour visiting all nodes only once and back to its starting node

is exactly the Traveling Sales Person (TSP) problem [12]. We use a branch-and-bound algorithm to find a shortest tour.

### *B. The Back-Tracking Algorithm*

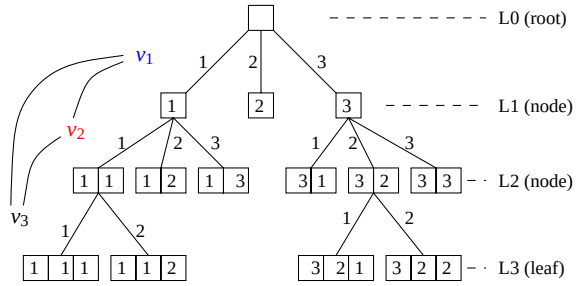



Fig. 8. Solution space

After ordering the node set for each VC specification into a shortest tour, we can then find the path options of the tour. We utilize a standard back-tracking algorithm to explore the path diversity. The algorithm is a recursive function performing a depth-first search, shown as Algorithm 1. The solution space is generated while the search is conducted. Figure 8 shows an example of solution space for three VCs, $v_1$, $v_2$ and $v_3$. $v_1$ and $v_2$ have three path options each, and $v_3$ two options. The tree has three mapped levels, each corresponding to determining a path for one VC. Specifically, at level $i$ ($L_i$), $v_i$ is mapped. Each level's node consists of a path of its own level VC plus a prefix that comprises the paths of its upper level VCs. For example, node 12 at $L_2$ means that $v_1$ selects path option 1 and $v_2$ option 2. The last level nodes, called *leaves*, are solutions found. For instance, leaf 321 at $L_3$ is a solution in which $v_1$ takes path option 3, $v_2$ option 2 and $v_3$ option 1.

The backtracking algorithm trades runtime for memory consumption. At any time during the search, only the route from the start node to the current expansion node is saved. As a result, the memory requirement of the algorithm is $O(m)$. This is important since the solution space organization needs excessive memory if stored in its entirety. Suppose that the size of a VC specification set is $m$, each VC has $p$ alternative paths, such a tree has $p^m$ leaf nodes and $(p^{m+1}-1)/(p-1)$ total nodes. As a result, an algorithm that moves through all nodes in the tree must spend $\Omega(p^m)$ time. This is time-consuming and not scalable. Therefore, bounding functions are required to cut infeasible branches. In our case, the *pair-wise* feasibility test of VC construction serves as a bounding function. While reaching each node in the tree, the paths of corresponding VCs are determined. This test then checks if *VC-to-LN assignment* can be done without conflict and bandwidth constraints can be satisfied. For example, at $L_2$ nodes, the feasibility of ($v_2$, $v_1$) is checked; at $L_3$ nodes (leaves), the feasibility of ($v_3$, $v_1$) and then ($v_3$, $v_2$) is checked. Note that the checking order must be observed. If this test fails, the algorithm prunes the current expansion node's subtrees, thus making the search efficient.

### *C. VC-to-LN Assignment*

The VC-to-LN assignment is the key step while constructing VCs. It is conducted pair-wise and incrementally. We detail this function in Algorithm 2. The input to the algorithm is a pair of VCs, $(v_i, v_j)$ [2], and their paths are known. The function returns true if VC-to-LN is done successfully for both VCs, and returns false otherwise. Besides, the configuration updates for $v_i$ and $v_j$ are stored and may be used in further tests. A VC $v$ has two configuration states, either 0 or 1. 0 means that VC-to-LN is not performed for $v$ yet; 1 means that the VC-to-LN is done successfully.

[2]VC pairs $(v_i, v_j)$ and $(v_j, v_i)$ are equivalent in the paper.

**Algorithm 1** The pseudo code of back-tracking algorithm

```
Input: Q: a queue of all VCs' implementation options.
Output: S: a queue of feasible solutions (s: a solution).
Initially, cur_level=0; Q.size=m; S and s are empty;
Sort Q by a priority criterion;
void back_tracking(cur_level, Q, S, s){
if (cur_level==Q.size()){
    // at a leaf, create routing tables
    s.configure_routing_table();
    S.push(s);
} else {
    // not a leaf, expand subtrees
    V=Q.pop(); // get options of current level's VC
    // try all branches of this level's VC
    for each v in V {
     if (feasibility_check(v, s)) //if false, prune the subtree
       s.push(v);
       back_tracking(cur_level+1, Q, S, s)}}
}
int feasibility_check(v, s){
    for each v' in s
      if (VC_to_LN(v,v')==false)
        return false;
    return true; }
```

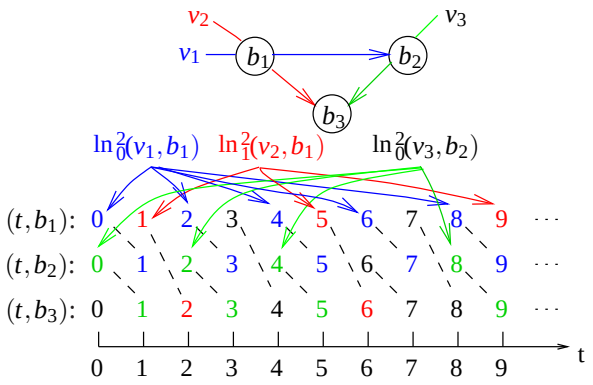



Fig. 9. An example of VC-to-LN assignment

| $v$ | *Buf. set* | $bw$ | $N$ | $W(D)$ | LN | Slot set |
|---|---|---|---|---|---|---|
| $v_1$ | $b_1, b_2$ | 1/2 | 1 | 2 | $ln_0^2(v_1,b_1)$ | $s_0^2(b_1)$, $s_1^2(b_2)$ |
| $v_2$ | $b_1, b_3$ | 1/4 | 1 | 4 | $ln_1^2(v_2,b_1)$ | $s_1^4(b_1)$, $s_2^4(b_3)$ |
| $v_3$ | $b_2, b_3$ | 3/8 | 3 | 8 | $ln_0^2(v_3,b_2)$ | $s_{0,2,4}^8(b_2)$, $s_{1,3,5}^8(b_3)$ |

TABLE II
VC PARAMETERS AND LN ASSIGNMENT RESULTS FOR FIG.9

**Algorithm 2** The VC-to-LN assignment procedure

```
int VC_to_LN(v_i, v_j){
if (v_i ∩ v_j == ∅) return true;
if (reference_consistency(v_i, v_j)==false) return false;
// v_i and v_j overlap but satisfy reference consistency
take any shared buffer b as the reference buffer Ref(v_i, v_j) = b;
compute the shared number T of LNs, T = GCD(D_i, D_j);
if (state(v_i)==0 && state(v_j)==0) {
   // Both states are 0
   for v in {v_i, v_j} {
    compute the available LN set AS_ln(v) for v;
    compute the required number of LNs N_ln(v) = ⌈NT/D⌉;
    if |AS_ln(v)| < N_ln(v) return false;
     assign LNs from AS_ln to v;
     allocate slot sets in the assigned LNs within D to v;
     state(v_i)=1; state(v_j)=1; }
    return true; }
if (state(v_i) != state(v_j) {
    // One state is 0 and the other 1
    // suppose (state(v_i)=0 and state(v_j)=1)
     map v_j's allocated slot sets to the new LN set as the
consumed LN set by v_j, CS_ln(v_j);
   compute the available LN set AS_ln for v_i;
    compute the required number of LNs N_ln(v_i) = ⌈N_i T_i/D_i⌉;
    if |AS_ln(v_i)| < N_ln(v_i) return false;
    assign LNs from AS_ln(v_i) to v_i;
    allocate slot sets in the assigned LNs within D_i to v_i;
    state(v_i)=1;
    return true; }
if (state(v_i)==1 && state(v_j)==1) {
    // Both states are 1
      map v_i's allocated slot sets to the new LN set as the
consumed LN set by v_i, CS_ln(v_i);
     map v_j's allocated slot sets to the new LN set as the
consumed LN set by v_j, CS_ln(v_j);
    if (CS_ln(v_i) ∩ CS_ln(v_j) == ∅)
      return true;
    else return false;
}
```

We detail an example on how this VC-to-LN assignment is conducted. Figure 9 shows three VCs, $v_1$, $v_2$ and $v_3$. In the figure, a bubble represents a buffer. Their parameters are listed in Table II. Their paths are represented by the respective ordered buffer sets. The VC-to-LN assignments are performed in order ($v_1$, $v_2$), ($v_1$, $v_3$) and ($v_2$, $v_3$).

1) $VC{-}to{-}LN(v_1,v_2)$: $Ref(v_1,v_2) = b_1$. Since $D_1 = 2$ and $D_2 = 4$, $T = GCD(D_1,D_2) = 2$. We can partition $b_1$'s slots into two logical sets. Initially, state($v_1$)=0 and state($v_2$)=0. The branch of "Both states are 0" is executed. We take $v_1$ first. The available LN set for $v_1$ $AS_{ln}(v_1) = \{0,1\}$, thus $|AS_{ln}(v_1)| = 2$. The required number of LNs $N_{ln}(v_1) = \lceil N_1T/W_1 \rceil = 1$. As $|AS_{ln}(v_1)| > N_{ln}(v_1)$, there are enough LNs to support $v_1$ bandwidth. We assign $ln_0^2(v_1,b_1)$ to $v_1$. The consumed LN set of $v_1$ $CS_{ln}(v_1) = \{0\}$. We then allocate slot sets $s_0^2(b_1)$ and $s_1^2(b_2)$ to $v_1$. The two sets constitute LN $ln_0^2(v_1,b_1)$. Next, we take $v_2$ up. $AS_{ln}(v_2) = \{0,1\} - CS_{ln}(v_1) = \{1\}$. The required number of LNs of $v_2$ $N_{ln}(v_2) = \lceil N_2T/W_2 \rceil = 1$. We assign $ln_1^2(v_2,b_1)$ to $v_2$. Then we allocate slot sets $s_1^4(b_1)$ and $s_2^4(b_3)$ to $v_2$. After this assignment, state($v_1$)=1 and state($v_2$)=1.
2) $VC{-}to{-}LN(v_1,v_3)$: $Ref(v_1,v_3) = b_2$. As $D_1 = 2$ and $D_3 = 8$, $T = GCD(D_1,D_3) = 2$. Since state($v_1$)=1 and state($v_3$)=0, the branch of "One state is 0 and the other 1" is executed. We map $ln_0^2(v_1,b_1)$ with respect to the reference buffer $b_2$, resulting in an equivalent LN $ln_1^2(v_1,b_2)$. Thus the consumed LN set of $v_1$ $CS_{ln}(v_1) = \{1\}$. The available LN set of $v_3$ is $AS_{ln}(v_3) = \{0,1\} - CS_{ln}(v_1) = \{0\}$. The required number of LNs of $v_3$ $N_{ln}(v_3) = \lceil N_3T/W_3 \rceil = 1$. We assign $ln_0^2(v_3,b_2)$ to $v_3$. Then we allocate slot sets $s_{0,2,4}^8(b_2)$ and $s_{1,3,5}^8(b_3)$ to $v_3$. After this assignment, state($v_3$)=1.
3) $VC{-}to{-}LN(v_2,v_3)$: $Ref(v_2,v_3) = b_3$. As $D_2 = 4$ and $D_3 = 8$, $T = GCD(D_2,D_3) = 4$. Since state($v_2$)=1 and state($v_3$)=1, the branch of "Both states are 1" is executed. In this step, we check whether the allocated slot sets for $v_2$ and $v_3$ can stay in different LNs after mapping them to the four LNs with respect to the reference buffer $b_3$. We map $s_1^4(b_1)$ of $v_2$ on $b_3$, obtaining an equivalent LN $ln_2^4(v_2,b_3)$. Then we map $s_{0,2,4}^8(b_2)$ of $v_3$ on $b_3$, obtaining LN $ln_{1,3}^4(v_3,b_3)$. Because $ln_2^4(v_2,b_3) \cap ln_{1,3}^4(v_3,b_3) = \emptyset$, $v_2$ and $v_3$ are conflict free with their slot assignment.

After the above three steps, the VC-to-LN assignments for the three VCs are successful. The slot sets are allocated accordingly, as shown in Table II. These can be used to create routing tables in switches.

### D. Routing Table Creation

While reaching a leaf (cur_level=Q.size()), a feasible solution is found. With each VC, a switch's partial routing table is created according to the VC's path and the allocated LNs, more accurately, the allocated slot sets within the admission cycle. The slot sets determine when the VC passes a particular buffer in a switch. For instance, if a VC $v$ with an admission cycle $D$ subscribes to $s_{\tau_1}^D(b)$ and $s_{\tau_2}^D(b)$, then slots $\tau_1 + kD$ and $\tau_2 + kD$ ($k \in \mathbb{N}$) of $b$ are reserved for $v$. The VC path determines the input link $e_{in}$ and the output link $e_{out}$ of the switch used by $v$ packets at the reserved slots. Thus, two routing table entries, $(\tau_1 + kD, e_{in}, e_{out})$ and $(\tau_2 + kD, e_{in}, e_{out})$, can be created in the switch. By composing the partial routing tables of all visiting VCs in a switch, we obtain a complete routing table for the switch. Optimization is also used to shrink the size of the routing tables. For example, entries $(4k, e_{in}, e_{out})$ and $(4k+2, e_{in}, e_{out})$ can be reduced to one entry $(2k, e_{in}, e_{out})$.

## VIII. Experimental Results

### A. Synthetic Communication Patterns

To investigate the tradeoff between capability and runtime with our approach, we conduct experiments on mesh networks using *open-ended VCs*. Although the experiments are performed on meshes, our method is topology independent. Since

the path diversity is the factor that complicates the exploration of the solution space, we realize and compare three different ways of considering alternative paths:

- One Path Option per VC (OPO): A VC considers one and only one locally optimal path. This path is selected from a set of candidates by minimizing the number of overlapped links with the previous VC in the VC set. This is a greedy scheme.
- Half Path Options per VC (HPO): A VC randomly chooses half of its path alternatives.
- Full Path Options per VC (FPO): A VC considers all alternative paths.

The back-tracking algorithm can be used to explore all alternative paths of VCs, allowing us to find all possible solutions. However, finding all solutions consumes long execution time and may not be necessary in some cases, because all solutions are equally good in the sense that VCs are conflict free, VC bandwidth demand and link bandwidth constraints are satisfied, and VC paths are shortest. Therefore we are interested in finding the first solution in this experiment.

| $n_v$ | OPO | | | HPO | | | FPO | | |
|---|---|---|---|---|---|---|---|---|---|
| | min. | avg. | max. | min. | avg. | max. | min. | avg. | max. |
| 12 | 0.31 | 0.38 | 0.46 | 0.36 | 0.61 | 1.33 | 0.44 | 4.33 | 22 |
| 16 | 0.46 | 0.53 | 0.6 | 0.49 | 1.5 | 6.35 | 0.57 | 25.2 | 138 |
| 20 | 0.8 | 0.83 | 1.2 | 0.99 | 2.25 | 30 | 2.08 | 48.5 | 480 |

TABLE III
RUNTIME WITH OPO, HPO AND FPO

Because there is no standard NoC benchmark available, we build a synthetic traffic generator to create random VC sets. The traffic generator takes as arguments the number of VCs, the bandwidth requirement of VCs and the maximum number of nodes in a VC. Note that a randomly generated VC set may not have a solution, for example, in case that VCs are demanding bandwidth more than the capacity of a particular link. To create only valid problems, we first run our program using FPO as a filter to get the valid problem set. In a 4×4 network, we set the VC bandwidth up to 1/2 link capacity, and the maximum number of nodes to 7. We vary the number $n_v$ of VCs from 11 up to 20. We obtain 200 problems after executing FPO, 20 for each $n_v$ in the range [11, 20]. Then we run HPO and OPO on the problem set. The percentage of the problems solved by HPO is 80%. The OPO solves only 22% of the problems. We compare their runtime in seconds for $n_v = 12, 16, 20$ in Table III.

As can be observed, OPO runs fastest, HPO the second, FPO the slowest. Besides, both HPO and FPO show great runtime variation from minimum to maximum. This is due to that a different problem may vary very differently on which leaf a solution exists. Since the solution is found while systematically constructing the solution tree, the runtime variation is high. The OPO does not exhibit much runtime variation because it explores only a linear path in the solution tree. The variation can be attributed to how fast the feasibility check is completed. In summary, back-tracking with FPO is the most powerful but takes longest execution time. The OPO method does no back-tracking. It is the least effective but fastest. Back-tracking with HPO sits in between. We also observed in our experiments, if limiting the program execution time, HPO/OPO can find a solution for some problems for which FPO/HPO cannot find a solution before the time threshold is reached.

### *B. An Industrial Case Study*

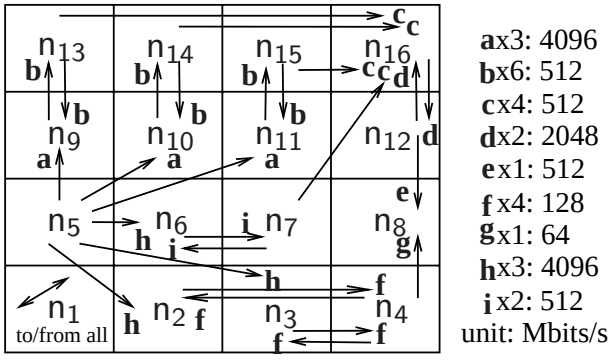



Fig. 10. Traffic flows for a radio system

We applied our program to a real application provided by Ericsson Radio Systems. As mapped onto a 4×4 mesh in Figure 10, this application consists of 16 IPs. Specifically, $n_2$, $n_3$, $n_6$, $n_8$, $n_{10}$ and $n_{11}$ are ASICs; $n_4$, $n_7$, $n_{12}$, $n_{13}$, $n_{14}$ and $n_{15}$ are DSPs; $n_5$, $n_8$ and $n_{16}$ are FPGAs; $n_1$ is a device processor which loads all nodes with program and parameters at start-up, sets up and controls resources in normal operation. Traffic to/from $n_1$ is for system initial configuration and no longer used afterwards. There are 26 node-to-node traffic flows that are categorized into nine types of traffic flows {**a**, **b**, **c**, **d**, **e**, **f**, **g**, **h**, **i**}, as marked in the figure. The traffic flows are associated with a bandwidth requirement. We use *closed-loop VCs* in this case study.

The case study comprises two phases: *VC specification* and *VC configuration*. The VC specification phase consists of *determining link capacity*, *normalizing VC bandwidth demand* and *merging traffic flows*. The VC implementation phase runs the configuration program, exploring the path diversity using FPO. In the following, we detail the case-study steps.

We first determine the minimum required link capacity by considering a heaviest loaded link. For each link $j$, suppose that $L_j$ is a set of traffic flows passing it, we have

$$\sum_{i=1}^{|L_j|} bw_i \leq \kappa_{bw} \cdot bw_{link}$$

where $bw_i$ is the data rate (bandwidth) of a traffic flow $i$ and $bw_{link} = l_w \cdot f_{clk}$ where $l_w$ is the payload bit width of a flit and $f_{clk}$ is the network clocking frequency. In the example, **a** and **h** are multi-cast traffic, and others are unicast traffic. The most heavily loaded link may be $e(n_5, n_9)$. The **a**-type traffic passes it and $bw_a = 4096$ Mbits/s. As all traffic flows are implemented using VCs, $\kappa_{bw} = 1$ in this case. To support $bw_a$, $bw_{link}$ must not less than 4096 Mbits/s. We choose the minimum 4096 Mbits/s for $bw_{link}$.

After obtaining the link capacity, we normalize the bandwidth demand into a fraction of link capacity. For example, 512 Mbits/s is equivalent to 1/8 link capacity.

Then we *merge traffic flows* by taking advantage of multi-node VCs. This can be done for multicast and low-bandwidth traffic. In the example, for the two multi-cast traffic **a** and **h**, if a VC specification contains only one source node and one destination node, we have to define 6 VCs for **a** and **h**, specifically, $\overline{v_1}(n_5,n_9)$, $\overline{v_2}(n_5,n_{10})$, and $\overline{v_3}(n_5,n_{11})$ for three **a** flows, and $\overline{v_4}(n_5,n_6)$, $\overline{v_5}(n_5,n_3)$, and $\overline{v_6}(n_5,n_2)$ for three **h** flows. Each of them must provide a normalized bandwidth 1. However, there are only 3 outgoing links for node $n_5$ on the mesh. Hence it is impossible to build such 6 VCs with the chosen link capacity. As our VC specification considers multiple nodes, we need to build only two multi-node VCs for traffic **a** and **h** as $\overline{v_a}(n_5,n_9,n_{10},n_{11})$ and $\overline{v_h}(n_5,n_6,n_2,n_3)$. In the example, traffic **b**, **c** and **f** require low bandwidth. We specify a VC to include as many nodes as a type of traffic flow spreads. For traffic **b**, we define a six-node VC, $\overline{v_b}(n_9,n_{10},n_{11},n_{13},n_{14},n_{15})$; for **c**, a five-node VC $\overline{v_c}(n_{13},n_{14},n_{15},n_{16},n_7)$; for **f**, a three-node VC $\overline{v_f}(n_2,n_3,n_4)$. In addition, on $v_b$, $v_c$ and $v_f$, only one container on each needs to be launched. The number $n_c$ of containers on a VC implementation $v$ is derived from $n_c \geq \overline{bw} \cdot |v|$, where $\overline{bw}$ is the normalized bandwidth demand of $\overline{v}$ and $|v|$ is the length of the loop path of the VC implementation $v$. If $n$ node-to-node flows are specified and implemented with closed-loop VCs, $n$ VCs must be set up and at least $n$ containers are required, one for each VC. Furthermore, as we use a closed-loop VC, two-simplex traffic flows can be merged into one duplex flow. For instance, for two **i** flows, we specify only one VC $\overline{v_i}(n_6,n_7)$. Performing this traffic-merge step results in 9 multi-node VCs, requiring 19 containers (6 for **a**, 6 for **h** and 1 for each of the rest). If the 26 node-to-node flows were *specified* with VCs containing only two nodes (one source node and one destination node), 26 VCs would have been defined, demanding at least 26 containers. This tells us that, with the multi-node VC specification, the network can be much more efficiently utilized.

With the three steps above, we complete defining the VC specification set. While executing the program to configure the VCs, we investigate the impact of VC sorting. Since VC sorting determines the VC levels in the solution tree and the VC-to-LN assignment order, it affects the runtime and the number of solutions. We tried three sorting schemes: *random*, *higher bandwidth first*, *less number of path options first*. In order to compare the potential of the schemes, our algorithm terminates after all solutions are found using FPO. *We did not do any tweaking or tuning* but used the original IP-to-node mapping and IP communication patterns without change. Corresponding to the three sorting schemes, the numbers of solutions found are 33, 30 and 76; the run time is 6, 6 and 12 seconds. Sorting by the number of path options is best in this example. This means that VCs with fewer alternative paths should be layouted first because they are more constrained. As a result, pruning their subtrees is more effective when they are considered in the upper levels in the tree.

## IX. Conclusion

Configuring VCs is a general problem for NoC application design with TDM-type guaranteed services. Its complexity arises from the network path diversity and various path overlapping scenarios. In the paper, based on the generalized concept of a logical network, we develop theorems for the configuration of conflict-free VCs. They are applicable to both open-ended and closed-loop VCs in current NoC proposals. Furthermore, we give a formulation on the multi-node VC configuration problem, and propose a back-tracking algorithm to constructively search for feasible solutions. Our experimental results with synthetic traffic and an industrial case study justify our approach in effectiveness and efficiency.

While our algorithm can cut branches which do not possibly lead to a solution, it allows us to find all possible solutions. These solutions can be further evaluated using an objective function, for example, for load balancing, to find optimal ones. This also helps to make the search more efficient by cutting more branches. This potential has not yet been explored, and it remains to be done in the future.

## Acknowledgment

The authors appreciate helpful discussions on the algorithms with Dr. Christian Schulte and Mikael Lagerkvist, and would like to thank Dr. Ingo Sander, Mikael Millberg and Tarvo Raudvere for their comments in improving the paper.

The research is partially supported by the EU FP6 project Sprint under contract 027580.

## References

[1] K. Goossens, J. Dielissen, and A. Rădulescu, "The Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 21–31, Sept-Oct 2005.

[2] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip," in *Proceedings of the Design Automation and Test in Europe Conference*, February 2004.

[3] T. Bjerregaard and J. Sparso, "A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2005, pp. 1226–1231.

[4] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *The Journal of Systems Architecture*, December 2003.

[5] A. Hansson, K. Goossens, and A. Rădulescu, "A unified approach to constrained mapping and routing on network-on-chip architectures," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 2005.

[6] S. Stuijk, T. Basten, M. Geilen, A. H. Ghamarian, and B. Theelen, "Resource-efficient routing and scheduling of time-constrained network-on-chip communication," in *Proceedings of the 9th Euromicro Conference on Digital System Design*, Aug. 2006.

[7] J. Hu and R. Marculescu, "Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures," in *Proceedings of the Design Automation and Test in Europe Conference*, 2003.

[8] S. Murali, M. Coenen, A. Rădulescu, K. Goossens, and G. De Micheli, "A methodology for mapping multiple use-cases onto networks on chip," in *Proc. of Design, Automation and Test in Europe Conference*, Mar. 2006, pp. 118–123.

[9] K. Srinivasan, K. S. Chatha, and G. Konjevod, "Linear programming based techniques for synthesis of network-on-chip architectures," *IEEE Transactions on VLSI Systems*, vol. 14, no. 4, pp. 407–420, 2006.

[10] J. T. Brassil and R. L. Cruz, "Bounds on maximum delay in networks with deflection routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 724–732, July 1995.

[11] A. Borodin, Y. Rabani, and B. Schieber, "Deterministic many-to-many hot potato routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 6, pp. 587–596, 1997.

[12] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, 2000.

# Paper 5

**Traffic configuration for evaluating networks on chip**

*Proceedings of the 5th International Workshop on System-on-Chip for Real-time Applications*, pages 535-540, Alberta, Canada, July 2005.

# Traffic Configuration for Evaluating Networks on Chips

Zhonghai Lu and Axel Jantsch
Laboratory of Electronics and Computer Systems
Royal Institute of Technology, Sweden
{zhonghai,axel}@imit.kth.se

## Abstract

*Network-on-Chip (NoC) provides a network as a global communication platform for future SoC designs. Evaluating network architectures requires both synthetic workloads and application-oriented traffic. We present our traffic configuration methods that can be used to configure uniform and locality traffic as synthetic workloads, and to configure channel-based traffic for specific application(s). We also illustrate the significance of applying these methods to configure traffic for network evaluation and system simulation. These traffic configuration methods have been integrated into our Nostrum NoC simulation environment.*

## 1 Introduction

One crucial aspect of Network-on-Chip design is to determine its network architecture [1]. It is challenging mainly due to two facts. One is that there exists a huge network design space with respect to topology, routing and flow control schemes etc. The other is that the network design should be customized for applications or a class of applications under current consideration and even for future upgrades or extensions. It is therefore very important to evaluate the network extensively in order to make right decisions on the network architecture.

Network evaluation commonly employs two kinds of traffic [2]. One is application-driven traffic, and the other synthetic traffic. Application-driven traffic models the network and its clients simultaneously. This is based on full-system simulation and communication traces. Full-system simulation requires building the client models. Besides, the feedback from the network influences the workload. Alternatively, execution traces may be recorded in advance and then replay this sequence for the network simulation. Application-driven traffic can be too cumbersome to develop and control. Synthetic traffic captures the salient aspects of the application-driven workload but can also be more easily designed and manipulated. Because of this, synthetic traffic is widely used for network evaluation.

In this paper, we present our traffic configuration schemes for the evaluation of networks on chips. Our contributions are (1) a unified representation for describing synthetic traffic. This representation is used to construct both uniform and locality traffic. The latter is essential to capture the traffic characteristics that explore communication locality for performance enhancement and energy saving; (2) a method to configure *application-oriented* traffic. Application-oriented traffic can be viewed as a traffic type between application-driven traffic and synthetic traffic. The spatial pattern of the application-oriented traffic reflects the communication distribution of the application(s). The temporal and message size specification may be synthetic or characterized from execution traces.

The rest of the paper is structured as follows. Section 2 briefs related work. In Section 3, we first describe the traffic configuration tree by which we introduce traffic characteristic parameters, then we detail the unified synthetic traffic representation and application-oriented traffic configuration. The experiments of applying the traffic configuration methods are reported in Section 4. Finally we conclude the paper in Section 5.

## 2 Related Work

In the communication community, traffic traces from physical networks are usually collected and analyzed to detect, identify, and quantify pertinent characteristics. For example, scale-invariant burstiness or self-similarity is an ubiquitous phenomenon found in diverse context, from LANs and WANs to IP and ATM protocol stacks [4].

For domain-specific applications, researchers use analysis or characterization methods to model the traffic of applications. In [3], a method is proposed to create abstract instruction-level workload models from source code for simulating application domain-specific multi-processor systems. It is shown in [8] that the traffic of multi-media applications has self-similarity characteristics.

There exist a variety of sources for application-driven

workloads. Parallel computing benchmarks such as SPLASH [9] or database benchmarks are possible tests for processor interconnection networks. The construction of synthetic workloads for network simulation can be found in [2].

# 3 Traffic Configuration

## 3.1 The traffic configuration tree

Network messages (traffic) can typically be characterized and constructed by considering their distributions along the three dimensions: spatial distribution, temporal characteristics, and message size specification. The spatial distribution gives the communication partnership between sources and destinations. The temporal characteristics describe the message generation probability over time. The size specification defines the length of communicated messages. We use a traffic configuration tree to express the elements and their attributes of traffic in Figure 1.

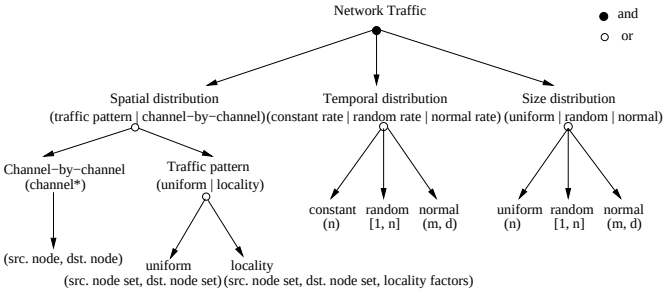



**Figure 1. Traffic configuration tree**

By the spatial distribution, traffic is classified into two categories: *traffic pattern* and *channel-by-channel* traffic. Traffic patterns consist of uniform and locality traffic. Each simulation cycle, the destinations of a traffic pattern may vary. That is to say, the same source node may send messages to a different channel. [1] With a traffic pattern, all the channels share the same temporal and size parameters. In contrast, channel-by-channel traffic consists of a set of channels with each channel taking its own temporal and size parameters. In addition, channel-by-channel traffic statically defines communication channels before simulation starts, implying that the source and destination nodes are fixed during the whole network simulation.

The temporal distribution has a list of items such as constant rate (periodic), random rate, and normal rate etc. The size distribution has a list of items such as uniform, random, and normal. As can be observed, these lists are just examples of possible distributions. Other interested distributions can be integrated into the tree with their associated parameters. By the tree, each traffic configuration can be set with a set of parameters.

Please note that, although we have divided the traffic configuration into three independent axes, it also allows one to configure traffic by jointly considering two axes. For example, the configuration of burstiness traffic may involve both the time and size axis.

## 3.2 Traffic patterns

### 3.2.1 Communication distribution probability $DP$

In the tree, two classes of traffic patterns are considered, namely, uniform and locality traffic. In order to build a unified expression for both traffic, we define *communication distribution probability* in relation to *communication probability* as follows:

- Communication probability of node $i$ $P_i$: the probability of sending messages to the network from node $i$.
- Communication probability from node $i$ to node $j$ $P_{i>j}$: the probability of sending messages to node $j$ from node $i$. For any source node $i$ with $N$ destination nodes numbering from 1 to $N$, $\sum_{j=1}^{N} P_{i>j} = P_i$.
- Communication distribution probability from node $i$ to node $j$ $DP_{i>j}$: the probability of distributing messages to node $j$ from node $i$ while node $i$ sends messages to the network. For any source node $i$ with its $N$ destination nodes, we have Equation 1, meaning that all the messages from node $i$ are aimed to all the $N$ destination nodes in the network.

$$\sum_{j=1}^{N} DP_{i>j} = 1 \tag{1}$$

By the definitions, we also have $P_{i>j} = P_i \cdot DP_{i>j}$.

We consider on-chip networks with regular topologies such as 2D meshes/tori, rings, trees etc. The benefit of the topological regularity is that the network nodes can be identified more structurally and with less bits. In the rest of the section, we use two-dimension topology to illustrate our representation for synthetic traffic patterns.

With two dimensions, the network topology directly maps to the Cartesian coordinate. Each network node can be identified and denoted as ($x$, $y$). For a source node ($x_s$, $y_s$), we define its *communication distribution matrix* $M_{(x_s,y_s)}$, which represents the spatial communication distribution of the node. Each item $v$ in position ($x_d$, $y_d$) of the matrix expresses the communication distribution probability $DP_{(x_s,y_s)>(x_d,y_d)}$, i.e., from the given source node ($x_s$,

[1] A *channel* in this paper refers to a logical path from a source node to a destination node.

$y_s$) to the destination node ($x_d$, $y_d$). For example, the following gives the communication distribution matrix $M_{(0,1)}$ of node $(0,1)$ in a 4x3 network.

$$M_{(0,1)} = \begin{bmatrix} 0 & 0 & 0 & 0.3 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0.5 & 0 & 0 \end{bmatrix}$$

From the matrix, we can see that $DP_{(0,1)>(1,0)}$=0.5, $DP_{(0,1)>(2,1)}$=0.2, $DP_{(0,1)>(3,2)}$=0.3. If an item in the matrix is zero, it means that no traffic is distributed/sent to the node from the source node.

### 3.2.2 Distribution coefficient *coef*

Suppose the distance between a source node ($x_s$, $y_s$) and a destination node ($x_d$, $y_d$) is $d$, we define communication distribution probability $DP_{(x_s,y_s)>(x_d,y_d)}$ as a relative probability to a common probability factor $P_c$ ($0 \leq P_c \leq 1$) in Equation 2 and 3:

$$DP_{(x_s,y_s)>(x_d,y_d)} = coef \cdot P_c \quad (2)$$

$$coef = 1 + \frac{\alpha}{d+1} \quad (3)$$

where *coef* is the *distribution coefficient*; $\alpha$ is called *locality factor*. Since $DP_{(x_s,y_s)>(x_d,y_d)} \geq 0$, $\alpha \geq -(d+1)$. This suggests that the valid region of $\alpha$ depends on distance $d$. Particularly when $\alpha = -(d+1)$, $DP_{(x_s,y_s)>(x_d,y_d)} = 0$; when $\alpha = 0$, $DP_{(x_s,y_s)>(x_d,y_d)} = P_c$. Besides, when $-(d+1) < \alpha < 0$, $DP_{(x_s,y_s)>(x_d,y_d)}$ is proportional to the distance $d$; When $\alpha > 0$, $DP_{(x_s,y_s)>(x_d,y_d)}$ is inversely proportional to the distance $d$. Intuitively, we would have defined the distribution coefficient as $coef = \alpha/d$. We use $d+1$ instead of $d$ in order to allow $d = 0$; We use $coef = 1+\alpha/(d+1)$ in order to incorporate the case when $\alpha = 0$. In this case, the distribution coefficient *coef* becomes independent of $d$, thus the traffic is uniformly distributed to nodes with a different distance.

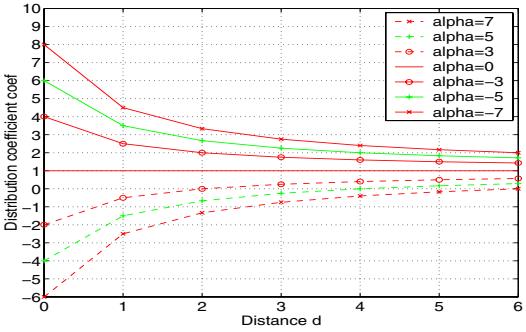



**Figure 2. The distribution coefficient**

By Equation 3, we depict a set of curves between *coef* and $d$ when $\alpha = 7, 5, 3, 0, -3, -5, -7$ in Figure 2. Note that the region when $coef < 0$ is invalid region. As can be seen, the value range of *coef* differs when $\alpha > 0$ from that when $\alpha < 0$. In order for coefficient *coef* to have a symmetric range when $\alpha > 0$ and $\alpha < 0$, we constrain the distribution coefficient $coef(d)$ to be not greater than 2, then the locality factor $\alpha$ falls into the region $[-(d+1), (d+1)]$ and $0 \leq P_c \leq 0.5$. The distribution coefficient *coef* in relation to locality factor $\alpha$ is shown in Figure 3.

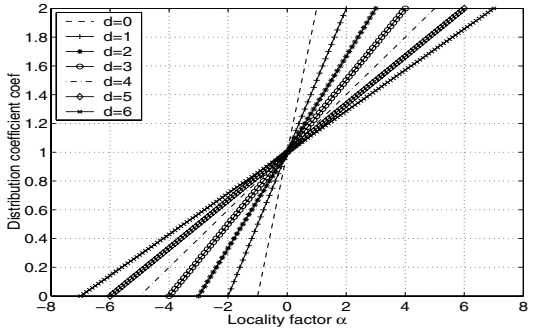



**Figure 3. The locality factor and distribution coefficient**

Applying the formula $\sum_{j=1}^{N} DP_{i>j} = 1$ on Equation 2, we have

$$\sum_{k=0}^{D} N_k coef_k \cdot P_c = 1 \quad (4)$$

where $N_k$ is the number of nodes with distance $k$ and $\sum_{k=0}^{D} N_k = N$; $D$ is the maximum distance between the source node and the destination nodes; $coef_k$ is the distribution coefficient for a destination node with distance $k$.

### 3.2.3 Common probability factor $P_c$

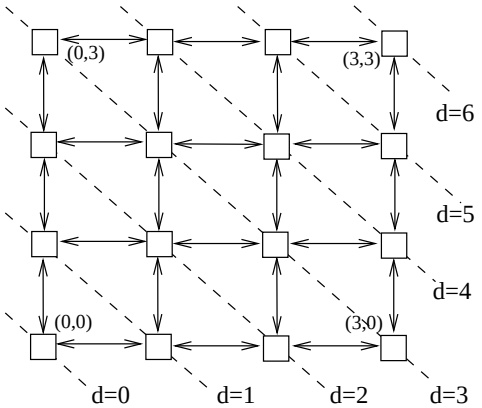



**Figure 4. A distance graph**

The common probability factor $P_c$ is calculated after $\alpha(d)$ is given. This allows us to give the value(s) of $\alpha$ freely, ignoring the details about the exact number of destination nodes with a different distance. As this detail is dependent on the source node position in the network topology, it may vary from node to node. We use an example to illustrate how $P_c$ is determined. Figure 4 shows a $4 \times 4$ mesh topology. We define on this mesh that $\alpha = 1$, which means $\alpha$ is a constant, i.e., irrespective of distance $d$. To determine $P_c$ for node $(0, 0)$, we draw the dashed lines to indicate all the destination nodes with a different distance. For its distance array $[0, 1, 2, 3, 4, 5, 6]$, it has the array of the number of destination nodes $[1, 2, 3, 4, 3, 2, 1]$. By Equation 3, we obtain the coefficient array $coef(d)$ of node $(0, 0)$ as $[2, 1.5, 1.3333, 1.25, 1.2, 1.1667, 1.1429]$.

Using Equation 4, we have

$$(1{\cdot}2+2{\cdot}1.5+3{\cdot}1.3333+4{\cdot}1.25+3{\cdot}1.2+2{\cdot}1.1667+1{\cdot}1.1429){\cdot}P_c = 1$$

Solving the equation, we receive $P_c = 0.0474$. If $\alpha = 0$, $P_c = 1/16 = 0.0625$. After obtaining $P_c$, we can calculate the distribution probability $DP(d)$ from node $(0, 0)$ to destination node(s) with distance $d$ by Equation 2 as an array $[0.0948, 0.0711, 0.0632, 0.0592, 0.0569, 0.0553, 0.0542]$. Clearly, for a different source node, $P_c$ may be different even if $\alpha$ is the same, since its distance array and the number of destination nodes with a certain distance may be different.

Since locality factor $\alpha$ may be set individually with $d$, we can control the amount of traffic distributed to a certain distance. Again with the example in Figure 4, if we define the node $(0, 0)$'s locality factor array $\alpha(d)$ as $[-1, 0, -1.2, -2.4, -4.0, -5.4, -6.3]$, its distribution coefficient array $coef(d)$ is $[0, 1, 0.6, 0.4, 0.2, 0.1, 0.1]$. Then its $P_c = 0.1587$. As a result, the communication distribution array $DP(d)$ of node $(0, 0)$ is $[0, 0.1587, 0.0952, 0.0635, 0.0317, 0.0159, 0.0159]$.

If all the source nodes' locality factors $\alpha$ are zero, their distribution coefficients $coef(d)$ to all destinations are one, i.e., independent of distance $d$. In this case, the traffic is uniformly distributed.

### 3.3 Application-oriented Workloads

Channel-by-channel traffic differs from the traffic patterns in that the traffic's spatial pattern is built on per-channel basis and static. This type of traffic is used to construct application-oriented workloads for specific applications. The temporal characteristics and message size specification may be approximated using analysis or communication traces. The set of traffic parameters of a channel is $\{s\_node, d\_node, \mathcal{T}, \mathcal{S}\}$, where $s\_node$ represents the source node, $d\_node$ the destination node, $\mathcal{T}$ its temporal characteristics, and $\mathcal{S}$ is its message size specification. In the following, we use a Motion JPEG (M-JPEG) encoder to illustrate how to approximately construct its application workload, supposing that its functional blocks are mapped to network nodes in a one-to-one manner. For simplicity, we assume a single synchronous clock.

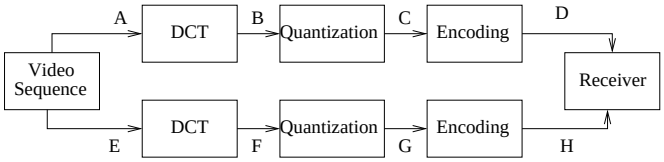



**Figure 5. An M-JPEG encoder**

Figure 5 shows the functional blocks of an M-JPEG encoder where each frame is separately compressed into a JPEG image [5]. The characters A, B, C, D, E, F, G, H indicate eight communication channels between modules. The M-JPEG codec is decomposed into three independent pipelined stages, namely, DCT (Discrete Cosine Transform), quantization and an encoding module. The figure consists of two parallel pipelines. Computations are performed on an 8x8 pixel block. Suppose one pixel contains 8 bits, one block has 64 bytes. The encoding module processes four blocks of data before outputting results. Due to data compression, the output from this module has a variable size. From recorded communication traces, we found it falls in the region $[16, 56]$ bytes.

| Channels | Period (Cycles) | Size (bytes) |
|---|---|---|
| A, E; B, F; C, G; | 160 | 64 |
| D, H | 640 | [16, 56] |

**Table 1. Channel parameters**

The DCT module is the performance bottleneck. To process one block of data, it consumes 135 cycles [6]. Assuming the maximum latency for processing one block of data over the communication channels is 25 cycles, we could consider that the critical path passes from the video sequence sender through the channel A/E and DCT. Following the worst-case style for synchronous design, we can design the period of the pipelined stages to be 160 cycles. Consequently, we may configure the traffic to model the application as follows: The channels A, B, C, and E, F, G are periodic channels with a constant period of 160 cycles and a uniform size of 64 bytes. The encoding modules may not generate output periodically. But, with the support of traffic shaping, the channel D and H can be assumed to transmit messages with a period of 640 cycles and a random size in region $[16, 56]$ bytes. We summarize the traffic parameters in Table 1.

# 4 Experiments

## 4.1 Use of the synthetic traffic patterns

We have integrated the traffic configuration methods into our Nostrum Network-on-chip Simulation Environment (NNSE) [7]. Based on a SystemC NoC simulation kernel, this tool allows one to construct a network and traffic by using the network and traffic characteristic parameters, and then evaluate the network with the traffic.

| TRAFFIC | $d$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| Locality | $\alpha$ | -1 | 0 | -1.2 | -2.4 | -4.0 | -5.4 | -6.3 |
| | *coef* | 0 | 1 | 0.6 | 0.4 | 0.2 | 0.1 | 0.1 |
| Uniform | $\alpha$ | -1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | *coef* | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| Non-locality | $\alpha$ | -1 | -1.8 | -2.7 | -3.2 | -3 | -2.4 | 0 |
| | *coef* | 0 | 0.1 | 0.1 | 0.2 | 0.4 | 0.6 | 1 |

**Table 2. Traffic specifications**

In order to understand the network behavior for locality traffic, we can create synthetic traffic using the method described in Section 3.2. We first construct a 4×4 mesh network operating synchronously. The network employs wormhole-based virtual-channel (VC) flow control with dimension-ordered XY routing, which is deterministic and deadlock free on meshes. The switch model is a single-cycle model. The number of VCs per physical channel of a switch is 4 and the depth of a VC is 2. The network diameter is 6. Then we inject one of the three classes of traffic into the network: *locality traffic*, *uniform traffic*, and *non-locality traffic*. With the uniform traffic, a network node sends packets to other nodes with the same probability. With the locality traffic, a network node sends packets to its *nearer* nodes with higher probability. With the non-locality traffic, a network node sends packets to its *further* nodes with higher probability. We list the traffic's locality factors $\alpha(d)$ and calculated distribution coefficients $coef(d)$ in Table 2. All the network nodes are both packet sources and sinks. The source nodes inject packets into the network via bounded FIFOs with a constant rate. The flits of packets are ejected from the network immediately once they reach destinations. One message contains only one packet and each packet is 4-flit long. By our configuration tree, all the traffic classes belong to periodic traffic with a uniform message size.

The left one of Figure 6 illustrates the average latency of packets. This figure demonstrates that, the more locality the traffic has, the lower average latency the network achieves. Note that, with the same injection rate, the offered load is different for the three types of traffic since the average packet distance is different. The non-locality traffic results in the largest offered load since it has the largest average packet distance. The uniform traffic is the second,

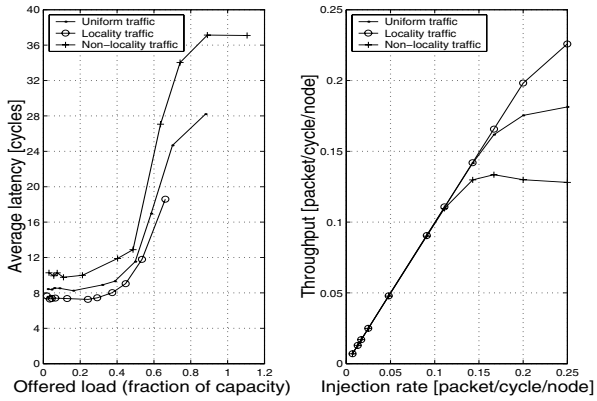



**Figure 6. Performance with traffic classes**

followed by the locality traffic. This also implies that the locality traffic allows higher packet injection rate before saturation. Besides, due to buffer overflow, the offered load does not increase proportionally when the injection rate is high. The right one of Figure 6 shows the throughput in relation to the packet injection rate. As can be observed, under the same amount of workload, the network can reach a higher throughput if the locality of traffic is higher.

Clearly the network shows significant performance improvement if the traffic is more locally distributed. We can conclude that, with the dimension-order routing, the wormhole network can efficiently benefit from traffic locality. Our traffic configuration method can flexibly enable to explore this by changing the traffic's locality factors.

## 4.2 Use of the application-oriented traffic

In NNSE, we manually map the functional modules of the M-JPEG model in Figure 5 onto the 4×4 mesh network described above. The mapping is done in a one-to-one fashion. Messages from these modules are encapsulated into packets. One packet has a payload length of 96 bits. This means a message with size $s$ bytes will be decomposed into $\lceil s/12 \rceil$ packets. The traffic is created and injected into the network according to Table 1. Simulation results show that the average latency of the packets is 9.68 cycles, the link utilization is $14.7\%$. Designers can use the simulated results to optimize the mapping or adjust the pipeline period to achieve design goals. Since the network is under-utilized, the designers may decide to shrink the network size to 3×3, or to implement more concurrent pipelines in the network.

# 5 Conclusions

We have proposed and demonstrated our traffic configuration schemes for evaluating networks on chips. The unified expression for configuring regular traffic patterns al-

lows designers to adjust the locality of traffic so as to analyze the network behavior under locality traffic. The configuration of application-oriented traffic enables to take into account the spatial communication patterns of the application while generating traffic, and it maintains the flexibility of synthetic traffic. The configured synthetic and application-oriented traffic can help designers to evaluate and thus make right decisions on the network architecture. In addition, the configuration of application-oriented traffic is beneficial to obtain performance data at an early design phase, thus useful for fast design space exploration.

Future work will fledge the traffic configuration tree to include more traffic distributions. Another direction is to integrate Quality-of-Service (QoS) traffic into the framework in order to evaluate NoC architectures supporting QoS with different guarantees on throughput and delay bounds.

## References


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

[2] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufman Publishers, 2004.

[3] J. Kreku, T. Kauppi, and J.-P. Soininen. Evaluation of platform architecture performance using abstract instruction-level workload models. In *Proceedings of International Symposium on System-on-Chip*, November 2004.

[4] K. Park, G. Kim, and M. Crovella. The effect of traffic self-similarity on network performance. In *Proceedings of SPIE International Conference on Performance and Control of Network Systems*, November 1997.

[5] S.-H. Sun and S.-J. Lee. A JPEG chip for image compression and decompression. *The Journal of VLSI Signal Processing*, 35(1):43–60, August 2003.

[6] The CS6100 M-JPEG core. www.amphion.com.

[7] The University Booth Program at the Design Automation and Test in Europe 2005. http://www.date-conference.com/conference/universitybooth.htm.

[8] G. Varatkar and R. Marculescu. Traffic analysis for on-chip networks design of multimedia applications. In *Proceedings of Design Automation Conference*, June 2002.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.

# Paper 6

**Evaluation of on-chip networks using deflection routing**

*Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI'06)*, pages 296-301, Philadelphia USA, May 2006.

# Evaluation of On-chip Networks Using Deflection Routing

Zhonghai Lu
zhonghai@imit.kth.se

Mingchen Zhong
mingchen@kth.se

Axel Jantsch
axel@imit.kth.se

Department of Electronic, Computer and Software Systems
Royal Institute of Technology in Sweden

## ABSTRACT

Deflection routing is being proposed for networks on chips since it is simple and adaptive. A deflection switch can be much smaller and faster than a wormhole or virtual cut-through switch. A deflection-routed network has three orthogonal characteristics: *topology*, *routing algorithm* and *deflection policy*. In this paper we evaluate deflection networks with different topologies such as *mesh*, *torus* and *Manhattan Street Network*, different routing algorithms such as *random*, *dimension XY*, *delta XY* and *minimum deflection*, as well as different deflection policies such as *non-priority*, *weighted priority* and *straight-through* policies. Our results suggest that the performance of a deflection network is more sensitive to its topology than the other two parameters. It is less sensitive to its routing algorithm, but a routing algorithm should be minimal. A priority-based deflection policy that uses global and history-related criterion can achieve both better average-case and worst-case performance than a non-priority or priority policy that uses local and stateless criterion. These findings are important since they can guide designers to make right decisions on the deflection network architecture, for instance, selecting a routing algorithm or deflection policy which has potentially low cost and high speed for hardware implementation.

**Categories and Subject Descriptors:** B.7.2 [Integrated Circuits]: Design Aids — Simulation; C.4.1 [Performance of Systems]: Design studies — Deflection routing

**General Terms:** Design, Performance

**Keywords:** Network-on-Chip, System-on-Chip communication network, Performance evaluation

## 1. INTRODUCTION

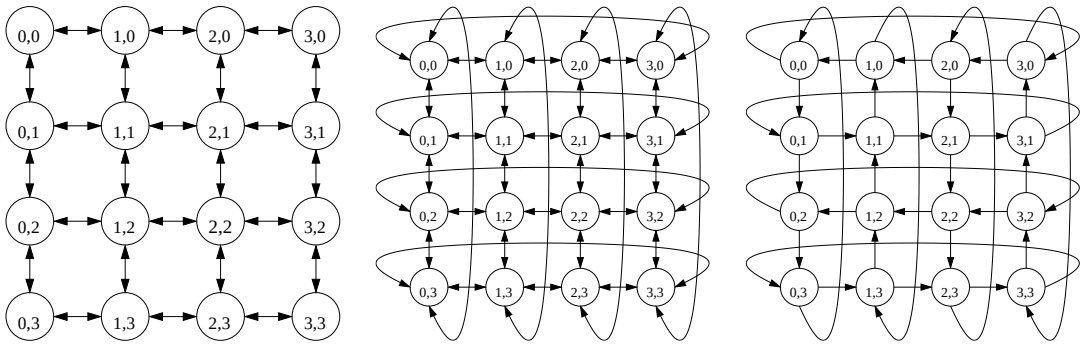
During the past five years, Network-on-Chip (NoC) [4, 10, 11, 13] has been suggested as a systematic approach to cope with the future System-on-Chip (SoC) design challenges such as interconnect difficulty, design productivity and stringent power constraints. Instead of using dedicated wires like bus interconnects, on-chip networks route packets to communicate data. First, by allowing concurrent transactions, NoC can potentially overcome the bandwidth limitation of buses to deal with the alarming design complexity enabled by the steady technology scaling [4]. Second, a network with a well-defined interface could serve as a communication platform to provide various services with diverse guarantees to upper-layer IP blocks. The possibility of the architectural reuse may reduce the Non-Recurring Engineering (NRE) cost and shrink time-to-market since an efficient domain-specific platform may be shared across many applications [8]. Third, the concurrent computation-communication structure and localized clock synchronization can efficiently reduce power consumption and thus satisfy power constraints, which are increasingly becoming design bottleneck and have to trade off with performance [12].

On-chip network is the core of network-on-chip. In general, a network has a much larger design space than a bus. A packet-switched network may be characterized by its topology, flow control and routing algorithm. For on-chip networks, a regular topology is favored against an irregular topology since it simplifies routing and layouting, and enables to modularize switches. Deflection routing [1, 2, 3, 6, 7, 10, 11] is a decentralized and adaptive routing mechanism. The distinguishing feature of a deflection switch lies in that it has no buffer queues. Packets are always on the run cycle by cycle, routing towards their destinations. Upon contending for links, packets with a lower priority will be *misrouted* to unfavored links according to a *deflection policy*. Since it has no buffer and flow management, a deflection switch can be designed with higher speed and lower cost than a wormhole or virtual cut-through switch. Thanks to its fully adaptive nature, it is also possible to avoid hot spots and provide fault-tolerance in the network.

The performance of a deflection network is the function of three parameters, namely, the topology, routing algorithm and deflection policy. In this paper, we explore the design space by means of cycle-true simulation. It is crucial to explore these design alternatives since they are implemented in hardware and may not be dynamically configurable or too costly to permit dynamic configuration. Therefore identifying the significance of each factor and evaluating their alternatives play a vital role in helping designers to make right decisions on the network architecture.

In the sequel, we brief the related work in Section 2. Then we describe and exemplify the characteristics of a deflection network in Section 3. Experimental results are reported in Section 4. Finally we draw conclusions in Section 5.


*GLSVLSI'06,* April 30–May 2, 2006, Philadelphia, PA, USA.




(a) A mesh (b) A torus (c) A MSN

**Figure 1: Three $4 \times 4$ network topologies**

## 2. RELATED WORK

Deflection routing, which is also called *hot potato* routing, has its root in [1]. It has been widely used in optical networks where buffering optical signals is too expensive. Because of its simplicity and adaptivity, it is also adopted and implemented in communication networks embedded in massively parallel computers such as the Connection machine [7]. It was initially proposed for on-chip networks in the Nostrum NoC [10, 11]. As projected in [11], a deflection switch can run 2.38 GHz with a gate count of 19370 in 65 nm technology.

Both average-case and worst-case performance in deflection networks have been analytically studied in [2, 3, 6]. Greenberg and Goodman [6] presented two approximate performance models to estimate the steady state throughput and average packet latency in the Manhattan Street Network (MSN). A deflection network is deadlock free but it has to avoid livelock, i.e., a packet continues routing in the network but never reaches its destination. A lot of work was focusing on deriving a performance bound based on assumptions of network traffic. Brassil and Cruz [3] derived upper bounds on the evacuation time of batch admissions [1] on an arbitrary topology and bounds on worst-case transit delay for hypercube networks admitting packets continuously. Borodin *et al.* presented bounds of deterministic algorithms for many-to-many traffic patterns in hypercube, mesh and torus networks [2]. A performance comparison between wormhole networks and deflection networks on chip can be found in [13].

Our work evaluates different kinds of deflection networks by simulation. The evaluation results are aimed to help make architectural decisions for on-chip deflection networks.

## 3. DEFLECTION ROUTING

A packet-switched deflection network is characterized by its topology, routing algorithm and deflection policy. We describe them with examples in this section.

### 3.1 Topology

The topology of a network defines how the network nodes are physically connected. It generally influences *network diameter*, *switch degree*, *link capacity* and *layout & wiring* for any kinds of networks. The network diameter is the length of the maximum shortest path between any two nodes. The switch degree is the number of input/output ports of a switch. The link capacity represents the number of links of the network. The layout & wiring refer to how the switches may be laid out and how links between switches may be wired. For deflection networks, the network topology has implications on two additional network properties, *deflection index* [3] and *don't-care density* [6].

- *Deflection index* is the largest number of hops that a single deflection adds to a packet's shortest path.
- *Don't-care density* is the percentage of destination nodes to which a source node has more than one non-overlapped shortest path to send packets. As a packet has multiple preferred links to take out of a node, whichever preferred link to take is regarded as a favorable choice. Hence the choice of link at this node for the packet is don't-care.

We consider two dimensional regular topologies since lower dimension and regular structure have advantages in simplifying the routing controller of switches, wiring, as well as potentially refraining Deep SubMicron (DSM) effects over wires [4, 10]. Examples are the 2D torus network proposed in [4] and the 2D mesh network suggested in [10]. Specifically we consider three 2D topologies with the same number of nodes, namely, 2D mesh, 2D torus and the Manhattan Street Network (MSN) [6]. Along each of the two dimensions, there are $K$ nodes. The mesh network has bi-directional links between nodes. But it has no toroidal connections. The torus network can be viewed as a mesh network with wrap-around connections. The MSN has a uni-directional 2D torus structure. But on a MSN, horizontal and vertical paths alternate in direction. Similarly to [6], for a MSN, we constrain $K$ is a multiple of 4. The three topologies are depicted in Figure 1. We qualitatively compare them in Table 1.

Note that the switches in the torus and MSN networks can be connected with equal-length of wires [4]. The don't-care density in Table 1 is calculated as follows: The three topologies are all node-symmetric. We can pick up any node

[1]Batch admission refers to that packets are admitted at a time slot in a batch without subsequent admissions.

| | Mesh | Torus | MSN |
|---|---|---|---|
| Diameter | $2(K\text{-}1)$ | $K$ | $K+1$ |
| Switch degree | 2,3,4 | 4 | 2 |
| Deflection index | 2 | 2 | 4 |
| Link capacity | $4K(K\text{-}1)$ | $4K^2$ | $2K^2$ |
| Don't-care density ($K$=4) | 60% | 73% | 60% |

**Table 1: The topological factors of three networks**

to calculate the don't-care density. For example, on the $K \times K$ mesh, any node has $K^2-1$ destination nodes. Among them, $2(K-1)$ nodes lie either on the same row or column as the source node. The source node sending packets to the $2(K-1)$ nodes has only one shortest path, but sending packets to the other $(K-1)^2$ nodes has two non-overlapped shortest paths. Therefore the don't-care density of the mesh is $(K-1)^2/(K^2-1)$. When $k=4$, it equals 60% (9/15).

## 3.2 Routing Algorithm

A routing algorithm determines the path and thus link a packet is to be delivered. Obviously a packet should be delivered along its shortest path whenever possible to reduce latency and increase throughput. For deflection routing, a routing algorithm determines a packet's favorable path and link. A deflection occurs only when a packet has to deviate from its shortest path, no matter whether a routing algorithm is minimal or not.

We consider four routing algorithms: Random, Dimension XY, Delta XY and Minimum deflection. Using random algorithms for on-chip network communication is beneficial for fault-tolerance, as discussed in [5].

- Random: A switch randomly chooses an available path to send packets.
- Dimension XY: A packet tries to first route along the X axis and then the Y axis.
- Delta XY: it routes packets by the minimal number of hops along the X ($\Delta_x$) and Y ($\Delta_y$) axis a packet has to travel. $\Delta_x/\Delta_y$ is the difference along the X/Y axis between a packet's source and destination address. $\Delta$ can be negative. If both desired links are available, it randomly chooses one. It differs from Dimension XY in that it does not prefer the X against the Y axis.
- Minimum deflection: it minimizes the occurrence of deflection at each hop. When a switch prepares to send packets, the switch calculates all possible permutations of packets' emission. Then it chooses the arrangement that the minimum number of packets will be deflected.

Among the above routing algorithms, the Dimension XY and Delta XY route packets with best-effort along their shortest paths, and thus are minimal. The others are not minimal. The routing by the Random algorithm is probability-based. While making routing decisions, the Minimum deflection algorithm tries to minimize the number of deflections by the local and oblivious [2] criterion. Note that it is not appropriate to separate a deflection policy from this routing algorithm. Since it makes routing and deflection decisions all at once, we can view Minimum deflection as both a routing algorithm and a deflection policy.

[2]Oblivious means stateless, i.e., do not consider the history of packet delivery, such as age and deflection times.

## 3.3 Deflection Policy

A deflection policy resolves packet contentions for links. Together with a routing algorithm, it determines packet-to-link assignment rules according to a pre-determined criterion. In addition, it can be used to design a livelock-free network. Since a higher priority packet wins link arbitration, the packet tends to reach its destination step-by-step deterministically. Nevertheless, it is difficult to derive an upper bound for arbitrary traffic patterns analytically [2].

In addition to the Minimum deflection, we consider both non-priority and priority-based deflection policies. For priority-based policies, we consider the straight-through [6] and a weight-based priority policy. With a priority policy, packet-to-link resolution is performed in favor of packets with a higher priority. A tie is resolved randomly.

- Non-priority: Misrouting decisions are made randomly. Packets have equal probability to be misrouted.
- Straight-through: A straight-through direction has a higher priority than a turn. The packet that arrives on the incoming row/column link is emitted on the outgoing row/column link.
- Weighted priority: The priority of a packet is based on multiple properties of the packet. It is explained in detail as follows.

The weighted priority takes into account a packet's multiple properties, such as *age*, *distance*, *deflection times*, and *default value*. Age indicates how long the packet has been alive in the network. A packet has a hop count field that records the number of hops (age) the packet has been routed. A higher hop count implies an elder age. The distance refers to the minimal number of links the packet has to travel from current node to its destination. Each packet also has an overhead field of deflection count, which bookkeeps the times of deflection during its delivery. In addition, packets may be of a different purpose. A default priority may be assigned to a packet from application by the source node. As all these properties can be meaningful for the packet priority, we use a weighted expression to calculate the priority. Each property $i$ is associated with a weight $w_i$, and $\sum |w_i| = 1$.

$$P = w_a \times A + w_h \times H + w_d \times D + w_f \times F \qquad (1)$$

where $P$ indicates the value of priority; $A$ is the *age* of packet; $H$ is the distance; $D$ represents the times of deflection; $F$ is the initial priority level; $w_a$, $w_h$, $w_d$ and $w_f$ refer to the weight of age, distance, deflection times and initial level, respectively. A weight embodies the impact that a certain packet property exerts on the packet priority. For example, if $w_a$=*0*, it implies the packet priority has nothing to do with its age; if $w_a > 0$, it means that an elder packet has higher priority than a younger one; if $w_a < 0$, a younger packet will result in a higher priority than an elder one. The absolute value of a weight $|w_i|$ reveals the significance of the property $i$ on the packet priority.

A deflection policy can take advantage of don't-care packets. For example, on the mesh or torus, if packet $G_1$ goes from node (1,1) to (2,2), and packet $G_2$ routes via node (1,1) to (2,1), contention for the eastern link occurs by Dimension XY routing. But $G_1$ is a don't-care packet at node (1,1), it can be routed to the southern link without deflection and $G_2$ takes the eastern link. If a deflection policy does not

consider $G_1$'s don't-care preference, $G_2$ may be misrouted if $P_{G_1} \geq P_{G_2}$. In implementation, we can use one extra bit in a packet to denote if it is don't-care. This attribute has to be checked and set at each hop. If it is asserted, the packet priority is temporarily negated for the local priority comparison, causing the don't-care packet to lose arbitration.

# 4. SIMULATION

In this section, we report results of simulations experimenting topology, routing algorithm and deflection policy.

## 4.1 Experimental setting

In order to evaluate the alternatives on network topology, routing algorithm and deflection policy, we construct $4 \times 4$ networks in our network-on-chip simulation environment [9]. This tool has a cycle-true NoC simulation kernel developed in SystemC. Besides it features a Graphical User Interface (GUI) which allows one to conveniently configure network parameters, traffic parameters, and then invoke kernel simulation. The deflection switch is a single-cycle packet-level model. Delivering packets through a switch takes exactly one cycle. The traffic pattern is uniformly-distributed random traffic. Each node sends traffic to other nodes with equal probability at a constant rate. The highest injection rate is one packet per cycle per node. Each node is equipped with a packet source queue. Packets are injected into the network through the source queue. The queue has conceptually infinite depth since it does not drop packets. The packet ejection model is ideal. Whenever a packet reaches its destination, it is ejected from the network immediately. In the case of multiple packets reaching destinations, all of them will be immediately sunk. Each simulation runs to the steady state, meaning that increasing simulation cycles does not change the results appreciably. Performance statistics are then collected at the steady state.

We measure both time-related and volume-related performance. For the time-related metrics, we consider *latency* $T$ and *network delivery time* $T_{net}$. Latency $T$ is counted from the instant a packet is injected into the source queue until that it reaches destination. The delivery time $T_{net}$ is the time a packet routes in the network after leaving its source queue until reaching destination. Therefore latency $T$ comprises network delivery time $T_{net}$ and source queuing time $T_{src}$ which is the time a packet waits in the source queue. The hop count of a packet gives its network delivery time. It is incremented by one for each hop. For the volume-related measurement, we consider *throughput*. It is defined as the average number of packets received per cycle in normalization with the number of nodes or links. We also measure *link utilization*, which is the average percentage of active/utilized links. It is important because the links are valuable network resources besides switches, and therefore should be efficiently used. The link capacity of a network gives an absolute constraint on network performance. An over-dimensioned network may use more links than necessary to improve performance. In addition, the number of active links directly relates to power consumption.

## 4.2 Topology

For this set of experiments, we set the routing algorithm to Dimension XY; the deflection policy is the Weighted priority policy with weight for age $w_a = 0.3$, for distance $w_h = 0.2$, for deflection count $w_d = 0.5$ and for initial priority $w_f = 0$. The deflection does not take advantage of don't-care packets. The performance results are shown in Figure 2.

As the packet injection rate $r$ increases, the network delivery time increases (Figure 2(b)). The increase is not linear but rather exponentially. This trend sustains until the network is saturated. For the MSN and mesh, they saturate at $r = 0.35$ and $r = 0.6$, respectively. The torus does not saturate at even the highest rate $r = 1$, since it has twice ideal throughput as much as the mesh and MSN (The bisection bandwidth of the torus, mesh and MSN is 16, 8, and 8 packets/cycle, thus the ideal throughput under the uniform traffic is 2, 1 and 1 packet/cycle/node, respectively).

With minimal routing, the network delivery time $T_{net}$ is related to the deflection count $D$ and deflection index $I$ by

$$T_{net} = D \cdot I + H_{min} \tag{2}$$

where $H_{min}$ is the average shortest distance of traffic. For example, as can be seen in Figure 2(a), when $r = 1$, $D(mesh) = 1.15$, $D(MSN) = 0.66$ and $D(torus) = 0.9$. As $H_{min}(mesh) = 2.67$, $H_{min}(MSN) = 2.93$ and $H_{min}(torus) = 2$, we have $T_{net}(mesh) = 4.97$, $T_{net}(MSN) = 5.6$, and $T_{net}(torus) = 3.8$. These figures match those on Figure 2(b). At the very low injection rate ($r = 0.025$), the deflection count is not zero. This is because the nodes inject traffic into the network synchronously, leading to contentions even under low load. When $r > 0.58$, the mesh incurrs the highest number of deflections (Figure 2(a)). This is because the traffic tends to conjest in the center, not well-balanced like the torus and MSN, since it has no toroidal connections. Figure 2(c) implies that the source queuing time $T_{src}$ ($T_{src} = T - T_{net}$) becomes extremely high when the network is saturated. From Figure 2(e), we can see that the link utilization in a deflection network can reach 100% upon network saturation.

The torus performs best because it has the highest link bandwidth and wrap-around connections to balance traffic. If we normalize the throughput in Figure 2(d) with the link capacity, it turns out that the MSN has the highest throughput per link before the network saturation (Figure 2(f)).

## 4.3 Routing Algorithm

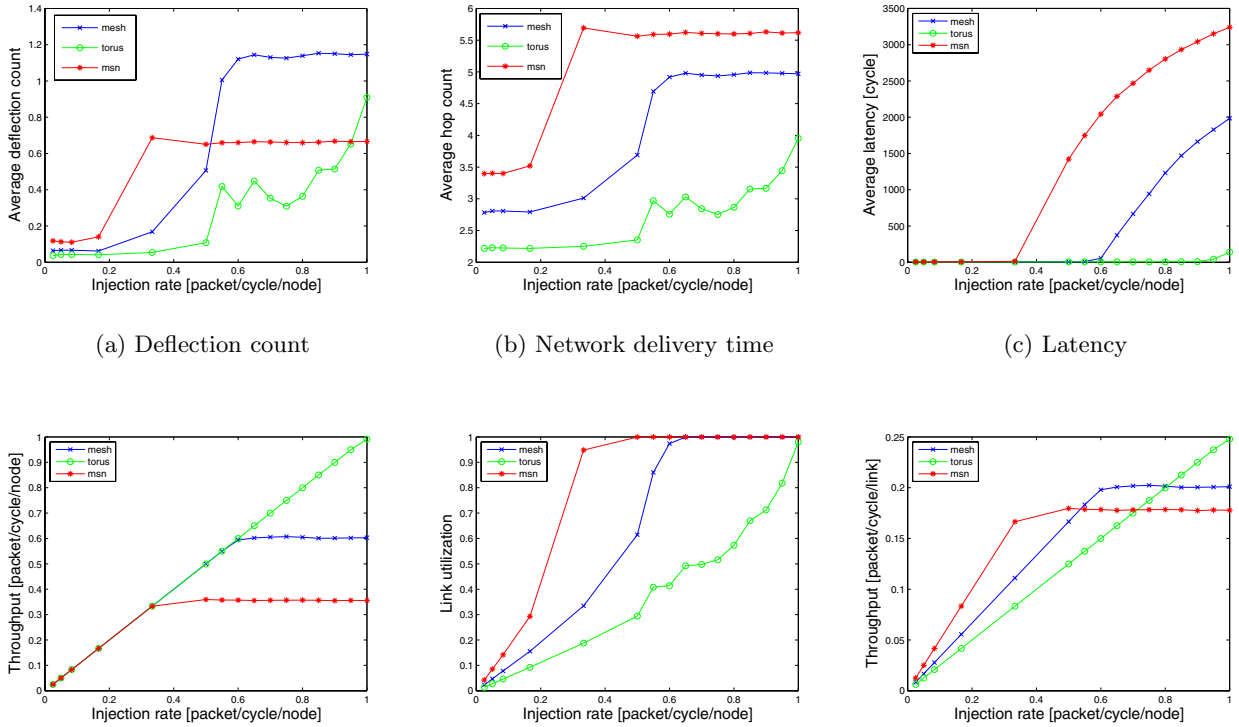
In this set of experiments, the topology is the mesh; the deflection policy is the Weighted priority policy used in section 4.2, but utilizes packets' don't-care preference. The results are depicted in Figure 3.

As can be seen, the Random algorithm performs worst in terms of latency and throughput since this algorithm does not guarantee packets to progress towards their destinations at each hop even there is no contentions for links. Even at the lower injection rate $r \approx 0.2$, the network is saturated and link is utilized 100%. The Dimension XY and Delta XY perform equivalently in latency, throughput and link utilization. The Minimum deflection algorithm performs a bit worse than Dimension XY and Delta XY, because its routing is not minimal and its deflection policy does not take packet delivery history into account. Although it minimizes deflection at each switch, the resulting overall performance is inferior when the network contention is high ($r > 0.7$).

## 4.4 Deflection Policy

In this group of experiments, we use the mesh network. The routing algorithm is Dimension XY.

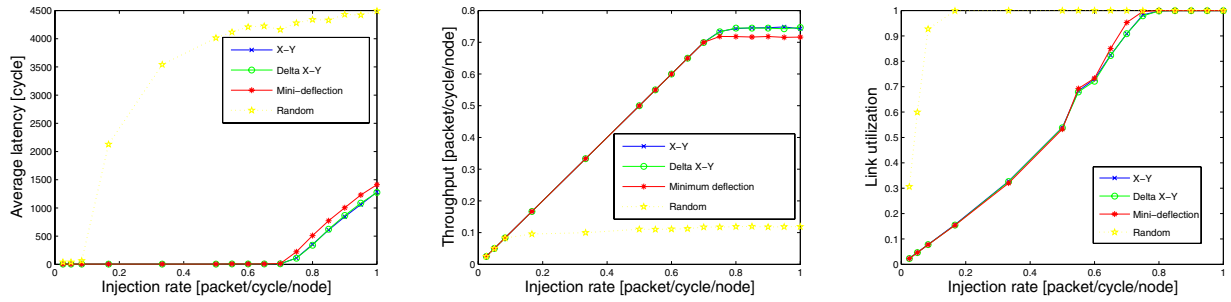
We compare the performance with the three deflection policies (Non-priority, Weighted priority and Straight-through)




(a) Deflection count (b) Network delivery time (c) Latency

(d) Throughput (e) Link utilization (f) Link-normalized throughput

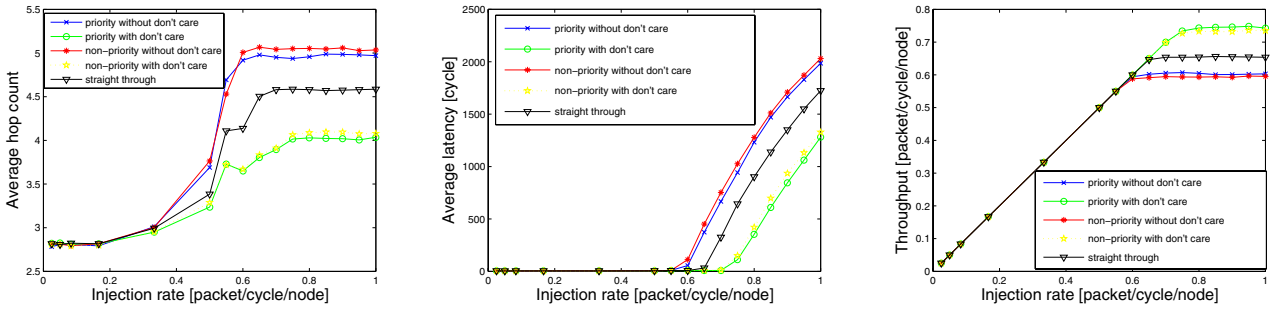
**Figure 2: Performance of different topologies**




(a) Latency (b) Throughput (c) Link utilization

**Figure 3: Performance of different routing algorithms**

in Figure 4. The priority weights use the same values as before. For the non-priority and priority policies, we consider both cases. One is with don't-care, the other without don't-care. As can be observed, the Weighted priority policy achieves better performance than the Non-priority policy. This is because distinguishing priority ensures higher-priority packets not misrouted, hence progressing faster to destinations. The Straight-through policy performs the midway between those with priority and those without priority, since it uses a sort of constant policy in favor of either the X or Y dimension to resolve contentions. More importantly, those policies considering don't-care achieves much better performance than those without considering don't-care. The saturation throughput is improved about 24% from 0.6 to 0.745 (Figure 4(c)). Meanwhile the network delivery time decrements about 1 cycle by 19% (Figure 4(a)). Since considering don't-care packets reduces deflections, packets are delivered faster and more. As a result, the reduction of source queuing time $T_{src}$ is also significant (45%), as shown in Figure 4(b). We performed other experiments with different weighted values, for example, $w_a = 1$ and $w_d = 1$. Their performance are close to each other.




(a) Network delivery time (b) Latency (c) Throughput

**Figure 4: Performance of different deflection policies**

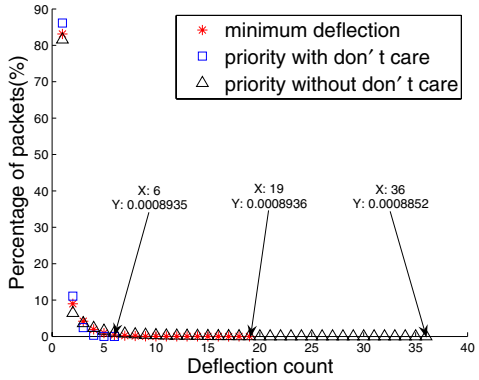



**Figure 5: Distribution of deflection count**

Given a topology, the deflection count $D$ is the most crucial factor for network performance (Equation 2). We compare the distribution graphs of deflection count when $r = 0.7$ for the Priority without don't-care, Priority with don't-care and Minimum deflection policies in Figure 5. The priority is given to deflection count in this case, i.e., $w_d = 1$. The distribution graphs are subject to the similar envelope. As can be seen, a very large percentage of packets are deflected only once. However, there exist packets experiencing much more deflections. The observed maximum deflection count is 36, 6, 19, respectively. Clearly, considering the don't-care preference of packets can improve the worst-case performance.

## 5. CONCLUSION

Determining the architecture of a deflection network is not a simple task because there exist many alternatives concerning network topology, routing algorithm and deflection policy. In this paper we have evaluated those alternatives that are currently being proposed for on-chip networks. We can conclude from our results that the network topology is the most significant factor, since it directly determines the deflection index and don't-care density. As long as a routing algorithm chooses the shortest path, the performance of different algorithms is close to each other. To improve the average-case, worst-case behavior and resolve livelock, packets should be prioritized, and a deflection rule should use the priority information and take packets' don't-care attribute into account. We believe these conclusions can be guidelines to select a deflection network architecture.

Our future work is to include application-specific traffic in the evaluation framework so that we can make guidelines for deflection networks targeting domain-specific applications.

## 6. REFERENCES


[1] P. Baran. On distributed computing networks. *IEEE Transactions on Communication Systems*, March 1964.

[2] A. Borodin, Y. Rabani, and B. Schieber. Deterministic many-to-many hot potato routing. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):587–596, 1997.

[3] J. T. Brassil and R. L. Cruz. Bounds on maximum delay in networks with deflection routing. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):724–732, July 1995.

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

[5] T. Dumitras and R. Marculescu. On-chip stochastic communication,. In *Proc. of Design, Automation and Test in Europe Conference*, March 2003.

[6] A. G. Greenberg and J. Goodman. Sharp approximate models of deflection routing in mesh networks. *IEEE Transactions on Communications*, 41(1), January 1993.

[7] W. D. Hills. The Connection machine. *Scientific American*, 256(6), June 1987.

[8] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits*, 19(12):1523–1543, December 2000.

[9] Z. Lu and A. Jantsch. Traffic configuration for evaluating networks on chips. In *Proceedings of the 5th International Workshop on System on Chip for Real-time applications*, July 2005.

[10] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the DATE Conference*, February 2004.

[11] E. Nilsson and J. Öberg. Reducing peak power and latency in 2D mesh NoCs using globally pseudochronous locally synchronous clocking. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, September 2004.

[12] D. Pamunuwa, J. Öberg, L.-R. Zheng, M. Millberg, A. Jantsch, and H. Tenhunen. A study on the implementation of 2D mesh based networks on chip in the nanoregime. *Integration - The VLSI Journal*, 38(2):3–17, October 2004.

[13] T. T. Ye, L. Benini, and G. D. Micheli. Packetization and routing analysis of on-chip multiprocessor networks. *Journal of Systems Architecture*, 50:81–104, February 2004.

# Paper 7

**Feasibility analysis of messages for on-chip networks using wormhole routing**

*Proceedings of the Asia and South Pacific Design Automation Conference*, pages 960-964, Shanghai, China, January 2005.

# Feasibility Analysis of Messages for On-chip Networks Using Wormhole Routing

Zhonghai Lu, Axel Jantsch and Ingo Sander
Royal Institute of Technology, Stockholm, 16440 Kista, Sweden
{zhonghai,axel,ingo}@imit.kth.se

**Abstract—The feasibility of a message in a network concerns if its timing property can be satisfied without jeopardizing any messages already in the network to meet their timing properties. We present a novel feasibility analysis for real-time (RT) and nonreal-time (NT) messages in wormhole-routed networks on chip. For RT messages, we formulate a *contention tree* that captures contentions in the network. For coexisting RT and NT messages, we propose a simple *bandwidth partitioning method* that allows us to analyze their feasibility independently.**

## I. Introduction

Network-on-Chip (NoC) [3, 4, 10] design starts with a system specification which can be expressed as a set or sets of communicating tasks. The second step is to map these tasks onto the nodes of a NoC instance. With a mapping, application tasks running on these nodes load the network with messages, and impose timing requirements. Timely delivery of messages is essential for performance and predictability. However, routing messages in a network is inherently nondeterministic because messages experience various contention scenarios which stem from sharing buffers at routers and links between the routers. These contentions cause indeterminate delay and jitter, leading to possibly the violation of the timing constraints of the messages. It is therefore important to conduct an analysis on messages to determine their feasibility. Given a set of already scheduled messages, a message is termed *feasible* if its own timing property is satisfied irrespective of any arrival orders of the messages in the set, and it does not prevent any message in the set from meeting its timing property [2]. In general, on-chip messages can be categorized as *real-time* (RT) and *nonreal-time* (NT) messages [10]. Messages with a deterministic bound, which must be delivered predictably even under worst case scenarios, are RT messages. Messages with a probabilistic bound, which request an average response time, are NT messages.

Wormhole flow control with lanes (virtual channels) is being advocated for NoCs due to its shorter latency, greater throughput and smaller buffering requirement [3, 10]. However, few studies have been performed to analyze the message feasibility for wormhole-routed networks. For real-time messages, the lumped link model [2, 5] is a path-based model in which all the links along a message $M_i$'s path are lumped into a single link. The message is scheduled on this link together with other competing messages. The feasibility test algorithms based on this model are efficient [2, 5]. However, due to lumping, all the competing messages must be scheduled in sequence. As a result, direct and indirect contentions are treated in the same way. Also, no concurrent use of the links on $M_i$'s path can be taken into account. In [6], Kim et al. used a blocking dependency graph to express the contentions a message may meet and derived the message's delivery upper bound. However, this graph does not reflect the possible concurrent use of links, too.

In the paper, we present a novel feasibility analysis for both RT and NT messages on wormhole-routed networks on chip. Section II describes the communication models delivering the RT and NT messages. In Section III, we first classify messages according to the type of performance bound and timing requirements on delay or jitter. Then, for the RT messages, we formulate a contention tree that can accurately reflect contentions and link usage. Specifically, it can distinguish direct and indirect contentions and captures concurrent use of links. Finally, we use a bandwidth partitioning method to test the feasibility of RT and NT messages coexisting in the network. The experiments are described in Section IV, followed by conclusions in Section V.

## II. The Communication Models

### A. The Nonreal-time Communication Model

In wormhole routing, a message is divided into a number of flits (flow control units) for transmission[1]. The head flit carrying routing and sequencing information governs the route. As the head flit advances, the remaining flits follow in a pipeline fashion. The message transmission is complete when its last flit is delivered to the destination. When required resources are unavailable, the messages are blocked in place. Wormhole routing manages two types of resources: the lanes and the physical link bandwidth. In conventional wormhole routers, the shared lanes are arbitrated on First-Come-First-Serve (FCFS), and they are multiplexed over the shared link bandwidth on demand [9]. This model is fair and produces good average-case latency results. But there is no guarantee that the messages are delivered before deadline. Therefore this communication model is suitable for the delivery of NT messages. With this NT model, the average network latency $T^{nt}$

[1]The effect of packetization is not considered in this study.

of delivering a message with $L$ flits is calculated by [1]:

$$T^{nt} = L/B^{nt} + HR + \omega = a + \omega \quad (1)$$

where $B^{nt}$ is the minimum link bandwidth allocated to the message along its route; $H$ denotes the number of hops the message passes; $R$ is the routing delay per hop. The first two terms represent the non-contentional or base latency $a$, which is the lower bound on $T^{nt}$; $\omega$ is the average contention delay due to the message being unable to access the shared lanes and link bandwidth.

### B. The Real-time Communication Model

Real-time messages must be served in such a way that the message delivery is predictable and guaranteed. Li and Mutka [7] developed a range of flow control schemes for real-time messages concerning priority mapping strategies, priority adjustment methods, and arbitration functions. In [2], based on a global priority, Preemptive Pipelined Circuit Switching for Real-Time (PPCS-RT) decouples the message delivery into two phases: path establishment and data delivery, where the path setup is preemptable. In [11], a flit-level preemption flow control is developed to resolve the priority inversion problem, i.e., a higher priority message is blocked by a lower priority message occupying shared resources. These real-time models complicate wormhole router design.

We assume a real-time (RT) message delivery model without a complicated router architecture and without a special service. All messages are globally prioritized (priority ties are resolved arbitrarily). This model arbitrates shared lanes and link bandwidth by priority. The priority, which may be assigned according to rate, deadline or laxity [5, 7], takes a small number of flits. With this RT model, assuming the same routing delay $R$ for the head flit and other flits, the worst-case latency $T^{rt}$ of delivering a message with $L$ flits is given by :

$$T^{rt} = (L + L_{pri})/B^{rt} + HR + \tau = c + \tau \quad (2)$$

where $B^{rt}$ is the minimum link bandwidth allocated to the RT message along its route; $L_{pri}$ is the number of flits taken by the message priority. The first term counts for the transmission time of all the message flits including that occupied by the priority; the sum of the first two terms is the non-contentional latency $c$, which is the lower bound on $T^{rt}$; the last term $\tau$ is the worst-case blocking time due to contentions.

## III. Feasibility Analysis

### A. The Message Model and Quality Classes

We consider messages or message streams that can be characterized by four parameters $M = (S,\ p,\ D,\ j)$, where $S$ denotes the maximum size of all the message instances; $p$ is the message period meaning that all the inter-arrival times of the message instances are never less than $p$; $D$ is the end-to-end delay constraint; $j$ is the jitter constraint. Though the delay $D$ is a constraint on the end-to-end communication latency, which is the sum of the latency due to the resource node $T_{node}$ and the network $T$, we focus on the network latency $T$. The effects of $T_{node}$ can be straightforwardly incorporated into the delay constraint resulting in a more stringent deadline.

Depending on the type of performance bound (deterministic or probabilistic) and that of timing requirement (delay or jitter), we define the Quality Class ($QC$) of a message, which can be viewed as an index representing the Quality of Service (QoS) requirement(s) of the message. For a probabilistic bound, we refer to constrain the bound to be an average response time. We define four quality classes as follows:

$QC_1$: jitter constrained, $D - j \leq T \leq D$.

$QC_2$: delay constrained, $T \leq D$, $j = D$.

$QC_3$: average jitter constrained, $D - j \leq T_{avg} \leq D$.

$QC_4$: average delay constrained, $T_{avg} \leq D$, $j = D$.

$QC_1$ and $QC_2$ messages are RT traffic while $QC_3$ and $QC_4$ are NT traffic. Also, $QC_2$ and $QC_4$ messages can be regarded as a special case of $QC_1$ and $QC_3$ messages when $j = D$, respectively.

### B. Real-Time Messages

According to Equation (2), a feasible real-time (RT) message $M_i$ satisfies its timing constraint:

$$\begin{array}{ll} \forall\ M_i \in QC_1 & D_i - j_i \leq c_i + \tau_i \leq D_i \\ \forall\ M_i \in QC_2 & c_i + \tau_i \leq D_i \end{array} \quad (3)$$

To estimate the worst-case latency of an RT message $M_i$, we must first determine all the contentions the message may meet.

In flit-buffered networks, the flits of a message $M_i$ are pipelined along its routing path. The message advances when it receives the bandwidth of all the links along the path. The message may directly and/or indirectly contend with other messages for shared lanes and link bandwidth. $M_i$ has a higher priority set $S_i$ that consists of a *direct contention* set $S_{D_i}$ and an *indirect contention* set $S_{I_i}$, $S_i = S_{D_i} + S_{I_i}$. $S_{D_i}$ includes the higher priority messages that share at least one link with $M_i$. Messages in $S_{D_i}$ directly contend with $M_i$. $S_{I_i}$ includes the higher priority messages that do not share a link with $M_i$, but share at least one link with a message in $S_{D_i}$, and $S_{I_i} \cap S_{D_i} = \emptyset$. Messages in $S_{I_i}$ indirectly contend with $M_i$. As an example, Fig. 1a shows a fraction of a network with four nodes and four messages. The messages $M_1$, $M_2$, $M_3$ and $M_4$ pass the links AB, BC, AB→BC→CD, and CD, respectively. A lower message index denotes a higher priority. The message $M_1$ has the highest priority, thus $S_1 = \emptyset$. For the message $M_2$, it directly contends with $M_3$, but it has a higher priority, thus $S_2 = \emptyset$. The message $M_3$ has a higher priority message set $S_3 = S_{D_3} = \{M_1, M_2\}$, $S_{I_3} = \emptyset$. For the message $M_4$, $S_{D_4} = \{M_3\}$ and $S_{I_4} = \{M_1, M_2\}$ because $M_1$ or $M_2$ may block $M_3$ which in turn blocks $M_4$.

To capture both direct and indirect contentions, we have formulated a *contention tree* defined as a directed graph $G$ :

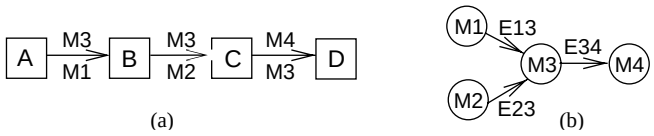



Fig. 1. Network Contentions and Contention Tree

$M \times E$. A message $M_i$ is a node $M_i$ in the tree, and vice versa. An edge $E_{ij}(i < j)$ directs from node $M_i$ to node $M_j$, representing the direct contention between $M_i$ and $M_j$. $M_i$ is called *parent*, $M_j$ *child*. Given a set $n$ of RT messages, after mapping to the target network, we can build a contention tree with the following three steps:

Step 1. Sort the message set in descending priority sequence with a chosen priority assignment policy.

Step 2. Determine the routing path for each of the messages.

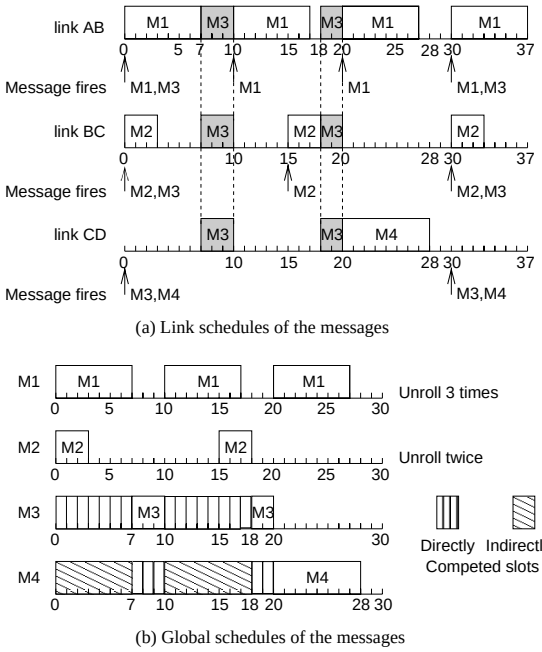
Step 3. Form a tree. If $M_i$ shares at least one link with $M_j$ where $i < j \leq n$, an edge $E_{ij}$ is created between them. Each tree node only maintains a list of its parent nodes.

In a contention tree, a direct contention is represented by a directed edge while an indirect contention is implied by a "walk" via parent node(s). A walk is a path following directed edges in the tree. The contention tree for Fig. 1a is shown in Fig. 1b, where the three direct contentions are represented by the three edges $E_{13}$, $E_{23}$ and $E_{34}$, and the two indirect contentions for $M_4$ are implied by the two walks $E_{13} \rightarrow E_{34}$ and $E_{23} \rightarrow E_{34}$ via $M_4$'s parent node $M_3$. Since knowing the routing path is a priori, creating a contention tree is more suitable for deterministic routing. For adaptive routing, it is difficult to figure out the worst-case routing path.

TABLE I
MESSAGE PARAMETERS AND LATENCY BOUNDS

| Message | Period $p$ | Deadline $D$ | Base latency $c$ | Lat. bound |
|---|---|---|---|---|
| $M_1$ | 10 | 10 | 7 | 7 |
| $M_2$ | 15 | 15 | 3 | 3 |
| $M_3$ | 30 | 30 | 5 | 20 |
| $M_4$ | 30 | 30 | 8 | 28 |

Table I shows the message parameters for Fig. 1, where the priority is assigned by rate, and deadline $D$ equals period $p$. The worst-case schedules[2] for the three links are illustrated separately in Fig. 2a. The latency bounds for the four messages are also listed in Table I. We can see that all the four messages are feasible. Looking into the schedules, we can observe that (1) $M_1$ and $M_2$ are scheduled in parallel. This concurrency is in fact reflected by the *disjoint* nodes in the tree. We call two nodes *disjoint* if no single walk can pass through both nodes. For instance, $M_1$ and $M_2$ in Fig. 1b are disjoint, therefore their schedules do not interfere with each other; (2) $M_3$ is scheduled on the overlapped empty time slots [8, 10] and [19, 20] left after scheduling $M_1$ and $M_2$. The competed slots [1,7] and [11,18] are occupied by $M_1$ or $M_2$. This is implied in the tree where $M_3$ has two parents, $M_1$ and $M_2$; (3) $M_4$ is scheduled only after $M_3$ completes transmission at time 20. The indirect contentions from $M_1$ and $M_2$, which are reflected via slots [1,7] and [11,18], *propagate* via its parent node $M_3$. For $M_3$, these slots are directly competed slots. For $M_4$, they become indirectly competed slots. The four message schedules are individually depicted in Fig. 2b. If the concurrent use of the two links, AB by $M_1$ and BC by $M_2$, was not captured, $M_3$ and $M_4$ would be considered infeasible since $M_2$ would occupy the slots [8, 10] and [18, 20], levaving only three empty slots before slot 30 for $M_3$ and $M_4$.




(a) Link schedules of the messages

(b) Global schedules of the messages

Fig. 2. Message Scheduling

In a contention tree, all levels of indirect contentions propagate via the intermediate node(s). This might be pessimistic since many of them are not likely to occur at the same time. If the number of shared lanes increases, the indirect contentions due to lane unavailability decrease. Also, a lower priority message can use the link bandwidth if a competing message with a higher priority is blocked elsewhere. To balance this pessimism, we have neglected priority inversion. As discussed in [2, 5], this problem can be alleviated by packetization.

### C. Nonreal-Time Messages

According to Equation (1), a feasible nonreal-time (NT) message $M_i$ satisfies its timing constraint:

$$\begin{array}{ll} \forall\ M_i \in QC_3 & D_i - j_i \leq a_i + \omega_i \leq D_i \\ \forall\ M_i \in QC_4 & a_i + \omega_i \leq D_i \end{array} \quad (4)$$

[2] A schedule is a timing sequence where a time slot is occupied by a message or left empty.

To analytically estimate the average contention delay $\omega_i$ is a difficult task because it is dependent on the network characteristics such as topology, routing algorithm, flow control, as well as the network communication patterns. Since this estimation is not the focus of this paper, we consider only special cases. To this end we use the closed form of contention delay [1] that Agarwal developed for random traffic $k$-ary $d$-cubes using dimension-order wormhole routing and unbounded internal buffers. For a 2D mesh network, $\omega_i$ is roughly calculated by: $\omega_i = \frac{3}{2} \cdot \frac{L_i}{B} \cdot \frac{\rho}{(1-\rho)} \cdot \frac{(H_i-1)}{H_i}$ , where $\rho$ is the network utilization calculated by $\rho = \sum_i (H_i - 1) q_i / C$, where $C$ is the network capacity measured in the total number of network links; $q_i$ is the probability of a network request a cycle.

Scheduling a new NT message leads to an increase in $\rho$. The timing constraints of the already scheduled messages must be met with the new $\rho$. Otherwise, the new message is infeasible.

### *D. Real-Time and Nonreal-Time Messages*

In a network supporting both RT and NT messages, estimating the values of worst-case blocking time $\tau$ and average blocking time $\omega$ becomes more complicated due to the possible interactions while delivering both classes of messages. For example, with respect to $\omega$, if the NT messages are allowed to use the unused bandwidth reserved by the RT messages, the RT messages may suffer from severe priority inversion problems, i.e., they may be blocked by the NT messages for an uncertain amount of time; with respect to $\tau$, the portion of the shared resources available to the RT messages may be dynamically changing, leading to intractability. This dynamic network behavior is not in accordance with our static analysis approach. In fact, such dynamic resource sharing schemes complicate the router design; for instance, it becomes too costly for the scheduler to adjust the allocated bandwidth. Therefore we have chosen to isolate the RT and NT traffic into two disjoint virtual networks. Such a nonwork-conserving service discipline has been discussed in [12].

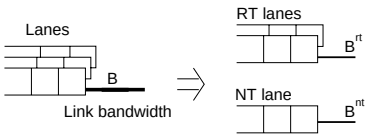



Fig. 3. Bandwidth Partitioning

Suppose the link bandwidth $B$ is normalized to 1, then each class of traffic has a weighted portion of $B$, as shown in Fig. 3. Let $B^{nt}$ and $B^{rt}$ be the bandwidth assigned to the NT traffic and RT traffic, respectively, $B^{nt} + B^{rt} = 1$. As a result, the link bandwidth is arbitrated by weighted round robin where the weights ($B^{nt}$ and $B^{rt}$) can be chosen a priori based on all types of traffic the router is designed to carry [8]. Concerning a network with uniform traffic, the same weights may be selected for all the routers. We can then apply our analysis method in Section III.B and III.C to the RT and NT traffic, respectively.

## IV. EXPERIMENTS

We have implemented a feasibility test algorithm based on the contention tree for RT messages and the bandwidth partitioning scheme for coexisting RT and NT messages. Then we conducted feasibility tests on messages in a 2D 8 X 8 mesh NoC with bidirectional links (the network capacity $C$ is $4 \times 8 \times (8-1) = 224$). The network uses wormhole flow control with dimension-order X-Y routing, which is a deterministic and deadlock-free algorithm. Lower dimension networks and deterministic routing algorithms are beneficial for NoCs in order to reduce the control complexity of the routers [4]. The purposes of our experiments are two-fold. First, we investigate how messages with a different Quality Class ($QC$) affect the NoC performance. Second, we examine the impact of a bandwidth partitioning on the system performance.

A message with the four parameters $(S,\ p,\ D,\ j)$ is randomly generated between a pair of nodes. The message size $S$ including protocol overhead randomly takes a value from $32$, $64$, $128$, and $512$ in flits. For each of the message sizes, the period $p$ takes a random value from $50\lambda$, $100\lambda$, $200\lambda$, and $800\lambda$, where $\lambda \in \{1, 2, 3\}$, respectively, and $p = D$. In this way, a longer message is likely to have a longer period. The routing delay per hop $R$ is chosen to be $2$.

The amount of traffic is *generated* given a threshold $\epsilon$ from $0.1$ to $1$ (normalized with the network capacity) with a step length of $0.1$. For any message generated, we must ensure that the link capacity is not violated. Let the probability of a network request of an RT and an NT message $M_i$ on any given cycle be $q_i^{rt}$ and $q_i^{nt}$, respectively. With a period of $p_i$, $q_i^{rt} = (L_i + L_{pri,i})/p_i$ and $q_i^{nt} = L_i/p_i$. Let $q_{ij}$ be the link bandwidth requirement of $M_i$ on link $j$, $q_{ij} = q_i$. For a link $j$ with $m$ RT and $k$ NT messages, the link constraint is:

$$\forall j \quad \sum_{i=1}^{m} q_{ij}^{rt} + \sum_{i=1}^{k} q_{ij}^{nt} \leq B^{rt} + B^{nt} = 1 \qquad (5)$$

If a new message generated does not lead to violate Inequality 5, the message is *offered* into the network; otherwise, it is discarded. By our traffic generation method, the *offered* traffic, which is the input of the feasibility test, is up to $62\%$ of the generated traffic as illustrated by the dashed line in Fig. 4. Also, we treat infeasible RT and NT traffic differently. If an RT message fails the feasibility test, it will not be considered any more. In contrast, all the offered NT messages are always involved. This is because a feasibility test needs to be conducted before admitting an RT message into the network while such a test is usually not necessary for an NT message. For each $\epsilon$, the simulation runs $50$ times to steady states and reports average results of *pass ratio*, i.e. the percentage of the messages that pass the feasibility test, and of the *network link utilization* of these feasible messages. In general, the more messages that fulfill their timing constraints, the higher the performance of the system. A higher utilization may imply a lower design cost while a lower utilization may imply an over-designed network.

We designed three groups of experiments. The first two groups consider delay-constrained messages. The first (Fig. 4)

concerns only delay-constrained RT traffic ($QC_2$), and $B^{rt} = 1$ and $B^{nt} = 0$. An RT message with a shorter period has a higher priority. The overhead due to the priority is two flits. The second one (Fig. 5) concerns both delay-constrained RT ($QC_2$) and delay-constrained NT ($QC_4$) traffic with various values of bandwidth partitioning. The last one (Fig. 6) considers jitter-constrained traffic, i.e., $QC_1$ and $QC_3$ messages. The jitter $j$ is set to be $0.15p$; thus the network latency of a feasible message falls in the region $[0.85p, p]$.

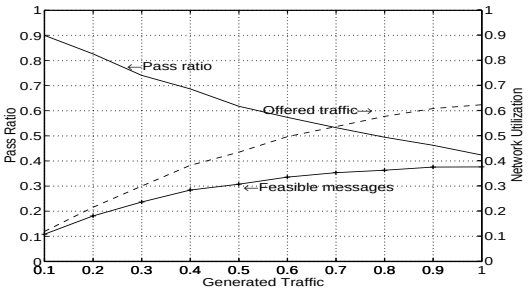



Fig. 4. Delay-constrained RT Traffic ($QC_2$)

In Fig. 4, as the generated traffic increases, the pass ratio decreases but the network utilization increases up to around $0.37$. Closing to this point, the network is near to saturation where the network latency increases exponentially but the throughput does not improve any more [1]. Therefore the gap between the offered traffic and the feasible traffic increases rapidly. Also, the pass ratio with this uniform traffic pattern is always below 1. For a *hard* real-time system that requires $100\%$ pass ratio, this means we need to find an application-specific mapping and our feasibility assessment can support such a mapping.

In Fig. 5, $QC_2$ and $QC_4$ messages are randomly generated; thus the number and message sizes of the RT and NT traffic have equal probability. With the value of $B^{nt} : B^{rt}$ increasing, the network tends to achieve higher pass ratio and utilization. In Fig. 6, $QC_1$ and $QC_3$ messages are also randomly generated. Comparing with Fig. 5, the corresponding pass ratio and network utilization are reduced. This is because a jitter constraint adds another condition ($D - j \leq T$) besides the deadline constraint ($T \leq D$), leading to fewer messages that pass the feasibility test.

## V. Conclusion

We have presented a feasibility analysis of messages in wormhole-routed networks on chip which is a crucial step in a NoC design flow. The contention tree we formulate can accurately reflect the network contentions but relies on deterministic routing. The static bandwidth partitioning method for coexisting RT and NT messages is simple but can illustrate some non-obvious results. From the experiments conducted, we can see that the feasibility analysis is useful for performance/cost tradeoff analysis of mapping messages with different QoS requirements on a NoC.

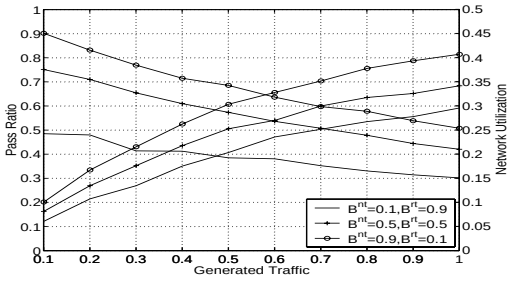



Fig. 5. Delay-constrained Traffic ($QC_2$-$QC_4$) with Bandwidth Partitioning

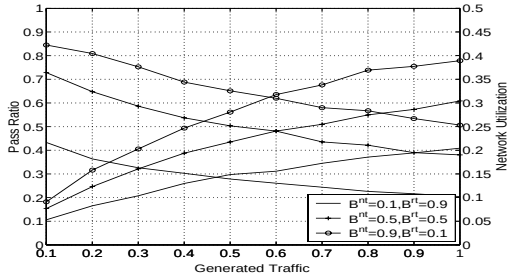



Fig. 6. Jitter-constrained Traffic ($QC_1$-$QC_3$) with Bandwidth Partitioning

Future work will investigate methods to enhance the pass ratio and/or network utilization by combining the feasibility assessment with task-to-node mappings.

## References


[1] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] S. Balakrishnan and F. Özgüner. A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):664–678, July 1998.

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

[4] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *DAC*, 2001.

[5] S. L. Harry and F. Özgüner. Feasibility test for real-time communication using wormhole routing. *IEE Proceedings of Computers and Digital Techniques*, 144(5), 1997.

[6] B. Kim, J. Kim, S. Hong, and S. Lee. A real-time communication method for wormhole switching networks. In *Proceedings of International Conference on Parallel Processing*, pages 527–534, Aug. 1998.

[7] J.-P. Li and M. W. Mutka. Real-time virtual channel flow control. *Journal of Parallel and Distributed Computing*, 32(1):49–65, 1996.

[8] J. W. S. Liu. *Real-time Systems*. Prentice Hall, 2000.

[9] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, February 1993.

[10] E. Rijpkema et al. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *DATE*, Mar. 2003.

[11] H. Song, B. Kwon, and H. Yoon. Throttle and preempt: a flow control policy for real-time traffic in wormhole networks. *Journal of Systems Architecture*, 45(8), Feb. 1999.

[12] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10), Oct. 1995.

# Paper 8

**Refining synchronous communication onto network-on-chip best-effort services**

In Alain Vachoux, editor, *Applications of Specification and Design Languages for SoCs*, Chapter 2, pages 23-38, Springer, 2006.

Chapter 2

# REFINING SYNCHRONOUS COMMUNICATION ONTO NETWORK-ON-CHIP BEST-EFFORT SERVICES

Zhonghai Lu, Ingo Sander, and Axel Jantsch
*Department of Electronic, Computer and Software Systems*
*Royal Institute of Technology, Sweden*
{zhonghai, ingo, axel}@imit.kth.se

**Abstract** We present a novel approach to refine a system model specified with perfectly synchronous communication onto a Network-on-Chip (NoC) best-effort communication service. It is a top-down procedure with three steps, namely, *channel refinement*, *process refinement*, and *communication mapping*. In channel refinement, synchronous channels are replaced with stochastic channels abstracting the best-effort service. In process refinement, processes are refined in terms of interfaces and synchronization properties. Particularly, we use *synchronizers* to maintain local synchronization of processes and thus achieve *synchronization consistency*, which is a key requirement while mapping a synchronous model onto an asynchronous architecture. Within communication mapping, the refined processes and channels are mapped to a NoC architecture. Adopting the *Nostrum* NoC platform as target architecture, we use a digital equalizer as a tutorial example to illustrate the feasibility of our concepts.

**Keywords:** Synchronous Model; Communication Refinement; Network-on-Chip.

## 1. Introduction

For system design, a synchronous design style is attractive since it allows us to separate timing from function. The designer can focus on the design of the system functionality without being distracted by unnecessary low-level communication details. This also facilitates the verification task, which is a key activity at the system level. Later, *refinement* explores the implementation space under constraints, making design decisions and filling in implementation details. Network-on-Chip (NoC) is an emerging SoC paradigm aimed to cope with the scalability problem of various buses in order to connect tens or perhaps even hundreds of microprocessor-sized heterogeneous resources, such as

processor cores, DSPs, FPGAs/ASICs, and memories. The complex integration is desired by ever-increasing functionality and enabled by the steady technology scaling. Nostrum ([11–13]) is our NoC architecture offering a packet-switched communication platform. To satisfy different performance/cost requirements, Nostrum provides two classes of communication services, namely, Best Effort (BE) and Guaranteed Bandwidth (GB) services. The BE service is connection-less where packets are routed without resource reservation. The GB service is connection-oriented where packets are delivered after enough bandwidth is reserved. It achieves better performance at the expense of higher cost.

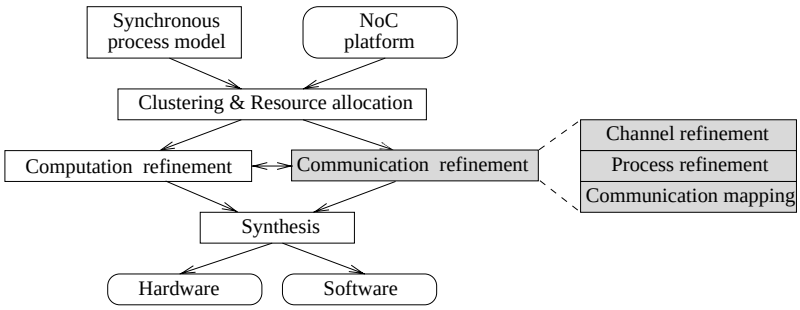



*Figure 2.1.* A NoC design flow

In this work, we are interested in mapping a system specified as a synchronous model onto a NoC. To this end, we propose a NoC design flow shown in Figure 2.1 where we concentrate on the communication problem. There are three communication-related tasks: *clustering & resource allocation*, *communication refinement* and *synthesis*. The clustering flattens the hierarchy in the model and groups processes into new processes with perhaps coarser granularity. With resource allocation, the grouped processes are allocated to network nodes, either HW or SW execution resources. Communication refinement bridges the gap between the communication model in the specification and the NoC communication implementation via adapters. With synthesis, these processes and adapters are synthesized into HW and/or SW.

We address the *communication refinement* that starts from a synchronous communication model and ends with the Nostrum NoC best-effort communication service. With the specification model, communication is perfectly synchronous with a global logical clock and cleanly separated from computation. With the NoC communication service, communication introduces variable delays and crosses multiple clock domains connected by a packet-switched network. Clearly the communication in the implementation domain is not synchronous, thus not consistent with that in the specification domain. Our contributions are (1) a novel approach to realize this communication refinement;

(2) a classification of process synchronization properties as *strict*, *nonstrict*, *strong*, and *weak* synchronization in order to formally analyze processes' local synchronization requirement(s) (Section 5.2); (3) using *synchronizers* (synchronization adapters) to maintain synchronization consistency during refinement (Section 5.3). We will focus on the synchronization issue while keeping the process computation untouched. Note that, this synchronization issue lies at the system modeling level, not at the lower implementation levels such as shared memory synchronization using locks or semaphores, as well as message passing synchronization using blocking or nonblocking semantics. We assume that, after a clustering, the resulting processes, more precisely, process networks, are top-level entities. Each process may comprise a hierarchy of sub-processes, which are intended to reside in a synchronous implementation domain. Besides, we consider that a resource maintains a local synchronous region. Consequently a process is to be mapped to one resource and one resource hosts exactly one process.

## 2. Related Work

Based on the isolation of communication from computation, a large body of work on communication refinement exists. Through the Virtual Component Interfaces (VCI) of the VSI Alliance ([9]), the COSY-VCC design flow ([3]) supports communication refinement from specification to performance estimation and to implementation. IPSIM ([5]) developed on top of SystemC 3.0 supports an object-oriented methodology and establishes two inter-module communication layers. The message box layer concerns generic and system-specific communication, while the driver layer implements higher level application-dependent communications. The SpecC methodology defines four levels of abstraction, namely at the specification, architecture, communication and implementation level, and the refinement transformations between them ([6]). These works do not assume a synchronous specification.

With synchronous communication, latency insensitive theory ([4]) targets synchronized HW design where synchronization can still be achieved using relay stations even if interconnecting synchronous IP blocks experiences indefinite wire latencies; Desynchronization for SW design was addressed in [1]. Furthermore, some mathematical frameworks were developed to support refinement-based design methods. Benveniste et al. present a theoretical framework for modeling heterogeneous systems, and derive sufficient conditions to maintain semantic-preserving transformations when deploying a synchronous specification onto GALS and the loosely time-triggered architectures ([2]). Another framework is proposed in [7] concerning the refinement of a polysynchronous specification, which allows the existence of multiple clocks

instead of a single clock. All these works are complementary to ours but none of them provides a detailed refinement approach targeting a NoC platform.

# 3. Refinement Overview

## 3.1 The perfectly synchronous model

The synchronous modeling paradigm is based on an elegant and simple mathematical model, which is the ground of synchronous languages such as Esterel, Signal, Argos and Lustre. The basis is the perfect synchrony hypothesis, i. e., both computation and communication take no observable time. A system is modeled as a set of concurrent communicating processes via signals. Processes use ideal data types and assume infinite buffers. Signals are ordered sequences of events. Each event has a time slot as a slot to convey data. If the data contains useful information, the event is *present* and called a *token*. Otherwise, the event is *absent* and modeled as a $\sqcup$ representing a clock tick. Each signal can be related to the time slots of another signal in an unambiguous way. The output events of a process occur in the same time slot as the corresponding input events. Moreover, they are instantaneously distributed in the entire system and are available to all other processes in the same slot. Receiving processes in turn consume the events and emit output events again in the same time slot. The medium a signal passes can thus be viewed as an ideal communication channel which has no delay for any event data types (unlimited bandwidth). A process specified in the synchronous paradigm is a synchronous process. For feedback loops, the perfect synchrony creates cyclic dependency between output and input, and thus leads to deadlock, which can be resolved with initial events in the specification. A synchronous model is deterministic, i. e., given the same input streams, it generates the same output streams.

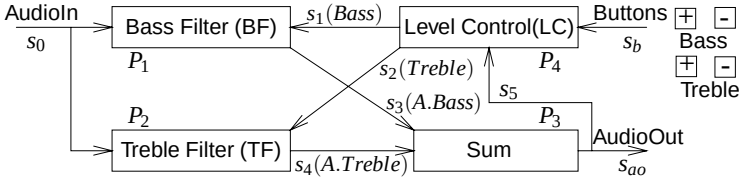



*Figure 2.2.* The digital equalizer

As a tutorial example, Figure 2.2 illustrates an equalizer model. It adjusts the bass and treble volume of the audio stream according to button control levels. In addition it prevents the bass level from exceeding a predefined threshold to avoid damaging the speakers. Its function can be described by the following set of equations, where the initial value '1' is used to resolve the feedback

loops:

$$
\begin{array}{lcl}
\mathit{AudioOut} & = & \mathit{Equalizer}(\mathit{Buttons}, \mathit{AudioIn}) \\
\quad \mathit{where} & & \\
\quad \mathit{AudioOut} & = & \mathit{Sum}(\mathit{AudioBass}, \mathit{AudioTreble}) \\
\quad (\mathit{Bass}, \mathit{Treble}) & = & \mathit{LevelControl}(\mathit{Buttons}, \mathit{AudioOut}) \\
\quad \mathit{AudioBass} & = & \mathit{BassFilter}(\mathit{AudioIn}, \mathit{init} : \mathit{Bass}) \\
\quad \mathit{AudioTreble} & = & \mathit{TrebleFilter}(\mathit{AudioIn}, \mathit{init} : \mathit{Treble}) \\
\quad \mathit{init} & = & 1
\end{array}
$$

This model is specified in the functional language Haskell and is executable.

## 3.2 Nostrum communication services

In Nostrum, each resource $R_i$ ($i = 1, 2, \cdots, n$) is equipped with a Resource-Network-Interface (RNI) in order to access the network, as shown in the lower part of Figure 2.3. The RNI and the network belong to the Nostrum protocol stack. Nostrum provides a message passing platform with two communication services, i. e., best-effort and guaranteed bandwidth. The BE service ( [12]) is connection-less. Packets are routed in the network without reserving network resources such as storage and link bandwidth. The end-to-end flow control, re-ordering, packetization and packet admission control are performed by RNIs. The BE service maintains message order, and is lossless and corruptless. It has no guarantee on timely delivery, but must have an upper bound on delivery time. To this end, we assume that the communication protocols can prevent the network from saturation and guarantee bounds on delay. The GB service is connection-oriented. Bandwidth is negotiated during a connection establishment phase. Packets are delivered after a connection is established. The GB service is implemented by using looped containers and temporally disjoint networks ([11]). The RNIs hide the service implementation details and make the services *transparently* accessible to applications. The access methods are communication primitives offered to the higher layer.

Within Nostrum, we define a set of basic communication primitives for message passing as follows:

**int open(int src, int dst, int service, struct bandwidth):** it opens a simplex channel between a source src process and a destination dst process. The service denotes the channel service class, 0 for the BE service, 1 for the GB service. The bandwidth is a user-defined record with three fields {int min_bw, int avg_bw, int max_bw} which specifies the minimum, average and maximum bandwidth (*bits*/*second*) requirement of the channel. The method returns a unique channel identity number (cid) upon successfully opening the channel. Otherwise, it returns various reasons of failure, such as a destination invalid, or performance not satisfied.

**bool write(int cid, void msg):** it writes msg to the specified channel cid. The size of messages is bounded. It returns the status of the write.

**bool read(int cid, void *msg):** it reads channel cid and writes the received data to the address starting at msg. It returns the status of the read.

We have implemented these primitives with the BE service using SystemC in our layered NoC simulator *Semla* ([13]). The write () and read () are presently implemented with nonblocking semantics. Semla is programmable as to network topology, process-to-resource mapping, routing algorithm, and traffic pattern. The current implementation opens channels statically during compile time and the opened channels are never closed during simulation.

## 3.3 The refinement procedure

Given a synchronous system specification, our objective is to refine the synchronous communication onto the Nostrum best-effort (BE) service. For this communication refinement, we propose a three-step procedure: *channel refinement*, *process refinement*, and *communication mapping*. We illustrate the procedure via a pair of producer-consumer processes in Figure 2.3. The three steps are marked by a circle with a step number inside it.

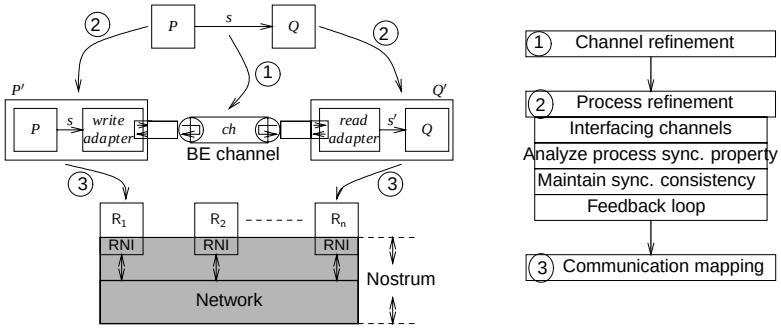



*Figure 2.3.* Communication refinement overview

**Step 1:** With **channel refinement**, we first abstract the behavior of the Nostrum BE service as that of stochastic channels which are then used to replace the ideal communication channels for passing signals. In Figure 2.3, the ideal channel for signal *s* between producer *P* and consumer *Q* is refined to a BE service channel *ch*. After being delivered via the service channel, signal *s* turns into signal $s'$, which is a derived version of *s*. Furthermore, *s* and $s'$ are not synchronous since different clock domains are involved in the service channel.

**Step 2:** With **process refinement**, we discuss how to connect a process to the service interface and how its synchronization property can be met by using adapters to wrap the process. Particularly, to guarantee a correct refinement, the process synchronization property must be consistent from the specification to the refined model. We classify and analyze the synchronization property of processes and then discuss how to maintain *synchronization consistency*. The process synchronization property can be annotated by designers on processes to enable automatically instantiating *synchronizers* to achieve synchronization consistency in the process refinement. Moreover, we consider design decisions to handle feedback loops, by which the process synchronization may be relaxed in order to optimize performance since a synchronous specification may over-specify the system. In Figure 2.3, $P$ and $Q$ are wrapped with a write and a read adapter, respectively. Note that an adapter contains both a component to interface with the service channel (writer/reader) and component(s) to achieve synchronization consistency (synchronizers) whenever necessary.

**Step 3:** Finally, together with a process-to-resource allocation scheme, the **communication mapping** is to implement the adapters and map the service channels on a NoC, in this case, the Nostrum simulator Semla. In Figure 2.3, the refined processes $P'$ and $Q'$ are mapped to the resources $R_1$ and $R_n$, respectively. Accordingly, the service channel *ch* is implemented via the interfaces provided by the RNIs of the resources $R_1$ and $R_n$.

## 4. Channel Refinement

The Nostrum BE service provides in-order, lossless and bounded-in-time communication between processes. However, its performance is *nondeterministic* since the message delivery experiences dynamic contentions in the RNIs and network. To capture its characteristics, we resort to a stochastic approach. Formally, we develop a unicast BE service channel as a point-to-point *stochastic* channel: given an input signal of messages $\{m_1, m_2, \cdots, m_n\}$ to the service channel, the output signal is $\{d_1, m_1, d_2, m_2, \cdots, d_n, m_n\}$, where message $m_i$ $(i = 1, 2, \cdots, n)$ is bounded in size; $d_i$ denotes the delay of $m_i$, which may be expressed as the number of absent ($\sqcup$) values and is subject to a distribution with a minimum $d_{min}$ and maximum $d_{max}$ value. The actual distribution, which may differ from channel to channel, is irrelevant. We do not make any further assumptions about this. If $d_i = n$ ($n$ is a positive integer), it means there are $n$ absent values between $m_{i-1}$ and $m_i$. We can identify two important properties of the generic service channel behavior: (1) $d_i$ is vary-

ing; (2) $d_i$ is bounded. This behavior is purely viewed from the perspective of application processes and its implementation details are hidden.

Replacing the ideal channel (zero delay and unlimited bandwidth) with a stochastic channel (varying delay and limited bandwidth) leads to the violation of the synchrony assumption. In the specification, a channel is ideal so that we can use a *single* signal $s$ to connect a producer to a consumer process. After replacing the ideal channel with a service channel, the signal $s$ can be seen as being *split* into a pair of signals, the original signal $s$ and its derived signal $s'$, as shown in Figure 2.3. For a process with two synchronous input signals, for example, the *Sum* process of the equalizer (Figure 2.2), if both signals $s_3$ and $s_4$ are delivered via a service channel, they are split, resulting in two derived signals $s_3'$ and $s_4'$, which are now the input signals to the *Sum* process. Apparently, the two pairs of signals, $s_3$ and $s_3'$, $s_4$ and $s_4'$, and the two derived signals $s_3'$ and $s_4'$ are not synchronous. A synchronous system becomes globally asynchronous, leading to possibly nondeterministic behavior which deviates from the specification. It is therefore important for a refinement to maintain synchronization consistency for functional correctness.

# 5. Process Refinement

We first describe how to interface with the service channels in general, and then discuss the synchronization property of processes followed by methods to achieve synchronization consistency. At the system level (a composition of processes), we discuss feedback loops.

## 5.1 Interfacing with the service channels

Once an ideal channel is replaced by a service channel, the processes can not be directly connected to the interface of the service channel. They must be *adapted* in terms of data and control because (1) the input/output data type of a service channel is of a bounded size while a signal in the specification assumes an ideal data type, whose length is finite but arbitrary, e. g., a 32/64-bit integer, a 64-bit floating point or a user-defined 256-bit record type, etc.; (2) the service channel has bounded buffers and limited bandwidth while a signal uses unlimited resources. The sending and receiving of messages use shared resources and thus control functionality has to be added to allocate shared resources, schedule multiple threads and achieve thread-level synchronization. These adaptations are achieved by a writer and reader process. Specifically, to interface with the service channels, a producer needs to be wrapped with a *writer*, a consumer with a *reader*.

## 5.2 Process synchronization property

In the system model, all signals of processes are synchronous. However, whether or not the input signals of a process must be synchronous is subject to the evaluation condition of processes, specifically, the local condition(s) to *evaluate* the input events. Because of the tight synchronization in the model, some processes may be over specified, limiting the implementation alternatives. During the refinement, the designer(s) must inspect and determine the synchronization property of the processes.

Inspired by [8], we use *firing rules* to discuss the synchronization property of *synchronous processes*. For a synchronous process with $n$ input signals, $PI$ is a set of $N$ input patterns, $PI = \{I_1, I_2, \cdots, I_N\}$. The input patterns of a synchronous process describe its firing rules, which give the conditions of evaluating input events at each event cycle. $I_i$ ($i \in [1, N]$) constitutes a set of event patterns, one for each of $n$ input signals, $I_i = \{I_{i,1}, I_{i,2}, \cdots, I_{i,n}\}$. A pattern $I_{i,j}$ contains only one element that can be either a token wildcard $*$ or an absent value $\sqcup$, where $*$ does not include $\sqcup$. Based on the definition of firing rules, we propose four levels of process synchronization properties as follows:

**Strict synchronization.** All the input events of a process must be present before the process evaluates and consumes them. The only rule that the process can fire is $PI = \{I_1\}$ where $I_1 = \{[*], [*], \cdots, [*]\}$.

**Nonstrict synchronization.** Not all the input events of a process are absent before the process fires. The process can *not* fire with the pattern $I = \{[\sqcup], [\sqcup], \cdots, [\sqcup]\}$.

**Strong synchronization.** All the input events of a process must be either present or absent in order to fire the process. The process has only two firing rules $PI = \{I_1, I_2\}$, where $I_1 = \{[*], [*], \cdots, [*]\}$ and $I_2 = \{[\sqcup], [\sqcup], \cdots, [\sqcup]\}$.

**Weak synchronization.** The process can fire with any possible input patterns. For a 2-input process, its firing rules are $PI = \{I_1, I_2, I_3, I_4\}$ where $I_1 = \{[*], [*]\}$, $I_2 = \{[\sqcup], [\sqcup]\}$, $I_3 = \{[*], [\sqcup]\}$ and $I_4 = \{[\sqcup], [*]\}$.

We can identify processes with a *strict*, *strong*, and *weak* synchronization property in the equalizer (Figure 2.2). The *Bass Filter* ($s_0$ and $s_1$) and *Treble Filter* ($s_0$ and $s_2$) have a strict synchronization. Both filters are composed of a FIR filter and an amplifier. The FIR filter is specified as an FSM, whose state transition is sensitive to time, thus a $\sqcup$ value in an audio stream can change the values of its output sequence. Meanwhile, the amplifier must have an amplification level, thus a $\sqcup$ value makes the amplifier undefined. The *Sum* process ($s_3$ and $s_4$) has a strong synchronization. It is a combinational process and thus tolerable to events with a $\sqcup$ value. However, the two events of $s_3$ and $s_4$ must be

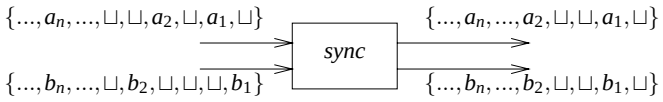
synchronized before being processed since they represent the low and high frequency components of the same audio sample. The *Level Control* ($s_b$ and $s_5$) process has a weak synchronization. It can fire even when either or both of the events of $s_b$ and $s_5$ are absent since pressing buttons happens irregularly and the bass level surpassing the threshold occurs only aperiodically.

## 5.3 Achieving synchronization consistency

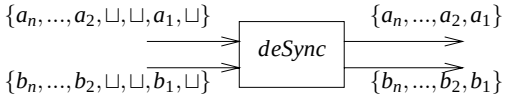
Apparently, for processes with a strict or strong synchronization, their synchronization properties can not be satisfied if any of their input signals passes through a service channel since the delays via the channel are stochastic. Although globally asynchronous, the processes can be locally synchronized by using *synchronizers* to satisfy their synchronization properties. To achieve strong synchronization, we use an align-synchronization process *sync*; to achieve strict synchronization, we use three processes, *sync*, *deSync* and *addSync*. We use a two-input process to illustrate these processes in Figure 2.4. An align-synchronization process *sync* aligns the tokens of its input events, as shown in Figure 2.4(a). It does not change the time structure of the input signals. A desynchronizer *deSync* removes the absent values, as shown in Figure 2.4(b). All its input signals must have the same token pattern, resembling the output signals of the *sync* process. Removing absent values implies that the process is *stalled*. The desynchronizer changes the timing structure of the input signals, which must be recovered in order to prevent from causing unexpected behavior of other processes that use the timing information. An add-synchronizer *addSync* adds the absent values to recover the timing structure, as shown in Figure 2.4(c). It must be used in relation to a *deSync* process. If the input events of the *deSync* is a token, the *addSync* reads one event from its internal buffers for each output signal; otherwise, it outputs a $\sqcup$ event. The two processes *deSync* and *addSync* are used as a pair to assist processes to fulfill strictness.

We can now use these synchronizers in connection with the *reader* and *writer* processes to wrap the original processes to interface with the service channels and maintain the synchronization consistency from the specification model to the refined model. For instance, as shown in Figure 2.5, we use a *sync* process and a pair of *reader*/*writer* processes to wrap the *Sum* process in the equalizer to maintain its strong synchronization. We use the three processes, *sync*, *deSync* and *addSync*, and a pair of *reader*/*writer* processes to wrap the *Bass/Treble Filter* process (Figure 2.2) to maintain their strict synchronization.

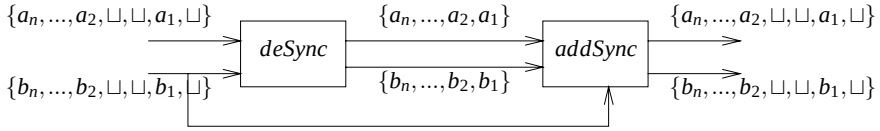
The refinement of processes with a nonstrict synchronization should be individually investigated according to their firing rules.



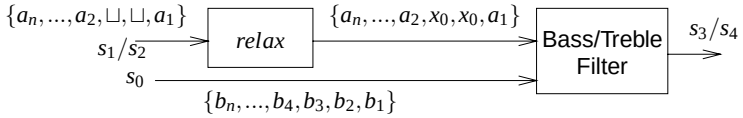

a) An align-synchronization process




b) A de-synchronization process




c) An add-synchronization process




d) A relax-synchronization process

*Figure 2.4.* Processes for synchronization

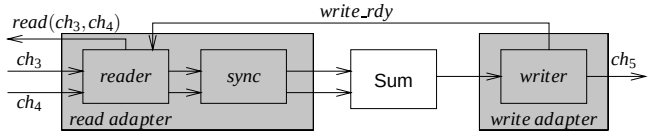



*Figure 2.5.* Read/Write adapters for a process with strong synchronization

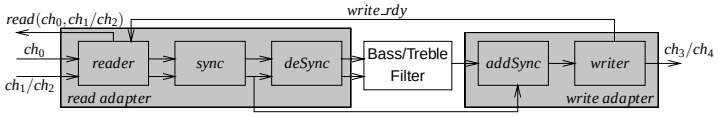



*Figure 2.6.* Read/Write adapters for a process with strict synchronization

## 5.4 Feedback loops

In the specification, feedback loops are resolved by using initial events. If the feedback signals pass through a service channel, the delays are nondeterministic. If following the initial event approach in the refinement procedure, we encounter a problem since we are not certain how many initial events are

required to resolve the deadlock. Consider the *Bass/Treble Filter*, if the tokens of $s_1/s_2$ are not available, it can not fire. This implies it may not be able to process enough audio samples in time, leading to violate the system's performance constraint. However, if the amplification level signals, $s_1$ (*Bass*) and $s_2$ (*Treble*), are delayed and thus not available, the amplifiers should continue functioning by, for example, using the previous amplification level or simply using a constant level like 1. In this case, the effect of pressing buttons may be delayed several cycles. This is tolerable since the human sensing of the changes in the audio volume is not instantaneous.

By this observation, we can in fact *relax* the strict synchronization of the processes *Bass/Treble Filter*, using a relax-synchronization process *relax* illustrated in Figure 2.4(d). If the input event is a token, it outputs the token; otherwise, a token $x_0$ is emitted. The exact value of $x_0$ is application dependent. Relaxing synchronization is a design decision leading to behavior discrepancy between the specification and the refined model. Care must be taken to validate the resulting system.

## 6. Communication Mapping

The inputs to this task are the refined model as well as a process-to-resource allocation scheme; the output is a communication implementation on Semla.

### 6.1 Channel mapping

With a resource allocation scheme, all processes are allocated to resources in a one-to-one manner. Note that this is not a limitation but due to the assumption on the clustering and resources (refer to Section 1). With such a clustering, inter-process signals, which represent inter-resource communications, are mapped to service channels. Since the processes may be hierarchical, we need to flatten the hierarchy to the level that each signal mapped to a service channel can be uniquely identified with a pair of a producer and a consumer process with *finer* granularity. For simplicity, we do not consider mapping multiple service channels to one implementation channel. Mapping channels is thus straightforward. Each pair of processes communicating via a service channel in the refined model results in its dedicated unicast implementation channel, which is mapped to the open channel primitive open(). For example, with the producer-consumer case, a BE channel setup is fulfilled by a single line of code: int ch[1]=open(P,Q,BE_SERVICE,NULL).

### 6.2 Communication process mapping

After the process refinement, a refined process consists of the original computational process, the writer and reader, and perhaps the synchronizer(s) to satisfy their synchronization properties. Our refinement keeps the original

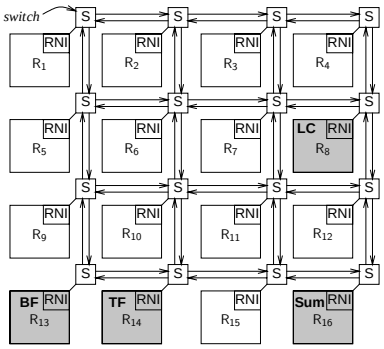
processes intact. Therefore, the tasks of communication process mapping are to implement the writer/reader, and the synchronizers such as *sync*, *deSync*, *addSync* and *relax*, and to coordinate the writing and reading operations.

In SystemC, processes are implemented as modules. The reader/writer may be implemented as separate modules or in the same modules as processes. We implement a process and its adapter(s) in a single module. For implementation, execution control in the module must be considered. Suppose the module has a single thread of control, we need to find a Periodic Admissible Sequential Schedule (PASS) for process executions ([10]). For the process in Figure 2.6, a PASS could be PASS={*reader*, *sync*, *deSync*, *Bass/Treble Filter*, *addSync*, *writer*}. Besides, a control signal *write_rdy* must be asserted by the *writer* to the *reader* to enable reading the channel(s) for the next-round PASS execution, as shown in Figure 2.6. This leads to a local feedback loop, and we adopt the initial event approach to deal with it. In this case, *write_rdy* is initially asserted. Using the communication primitives defined in Section 3.2, the SystemC module for Figure 2.6 is sketched as follows, with each component explained briefly in commentary:

```
//initially write_rdy=1;
//read_ch0_rdy=0; read_ch1_rdy=0
//sync_rdy=0; compute_done=0;
if (write_rdy==1){
//(1) reader: nonblocking read ch1 and ch2
 if (read_ch0_rdy==0)
   if ((read(ch[0],&r_msg1))==true)
      read_ch0_rdy=1;
 if (read_ch1_rdy==0)
   if ((read(ch[1],&r_msg2))==true)
      read_ch1_rdy=1;
//(2) sync: synchronize the two events
if (read_ch0_rdy==1 && read_ch1_rdy==1)
   sync_rdy=1;
else sync_rdy=0;
//(3) deSync: desynchronization by guard
if (sync_rdy==1 && compute_done==0){
   //process computation
   //return w_msg and set compute_done to 1
   w_msg=compute(r_msg1,r_msg2);
   write_rdy=0; compute_done=1;}
}
//(4) addSync: fill synchronization
 if (sync_rdy==1 && compute_done==1) {
//(5) writer: nonblocking write ch3
```

```
    if (write_rdy==0)
      if (write(ch[3],w_msg)==true){
          write_rdy=1;
          sync_rdy=0; compute_done=0;
          read_ch0_rdy=0; read_ch1_rdy=0;}}
}
```

In the implementation domain, whether to emit and pass $\sqcup$ via a service channel either as a special message or using one bit to indicate *presence* and *absence* can be a design decision. To preserve the semantics, $\sqcup$ must be transported. However, this incurs too much overhead on computation and communication, and may be meaningless since its value is useless. Therefore $\sqcup$ is usually neglected. Only in cases where the timing information carried by $\sqcup$ is used by other processes, it must be emitted and passed. In the equalizer case, $\sqcup$ is neglected since its timing information is not used by any of the four processes.




(a) Resource allocation

| Load (%) | Avg. Delay (cycles) | Throughput (samples/cycle) |
|---|---|---|
| 9.06 | 15.00 | 0.0667 |
| 16.96 | 16.53 | 0.0605 |
| 25.16 | 18.52 | 0.0540 |
| 29.97 | 20.24 | 0.0494 |
| 35.36 | 22.59 | 0.0443 |
| 43.28 | 35.14 | 0.0285 |

(b) Performance

*Figure 2.7.* The equalizer mapped on a NoC

We have implemented the equalizer in Semla. The purpose is to validate the concepts of our refinement approach. Figure 2.7(a) illustrates the mapped equalizer in a $4 \times 4$ mesh NoC. All the five inter-resource signals $s_1, s_2, \cdots, s_5$ in Figure 2.2 use the BE service. The resources and the network run with the same speed. The switches operate synchronously with the switching per hop taking one cycle. The message streams on $s_3$ and $s_4$ are injected into the network conservatively so that a new audio sample will not be processed by the filters until the previous sample has been handled by the *Sum* process. This implies that the audio samples are not processed in a pipeline fashion in the network. In addition, we inject background traffic with uniformly distributed random destinations in the network. The motivation is to load the network with

reasonable amount of traffic since the equalizer example can only make use of a small fraction of the network capacity. Figure 2.7(b) shows the equalizer performance, where the network load is the average percentage of active links per cycle. The process computations are function calls and complete instantly. We observe the average delay that is the time (in cycles) to process one sample. Since the audio processing is not pipelined, the throughput (samples/cycle) is simply the inverse of the average delay. In Figure 2.7(b), the first row shows the case where there is no background traffic. As expected, when the network is increasingly loaded, the average delay is increased and the throughput decreased. The average delay can be seen as the time to respond to a button press or to activate bass control. We noted that the audio output sequences are different from those observed from the specification due to relaxing the synchronization for the feedback loops. We conducted other experiments in which we removed the feedback loops, and could validate that the output sequences agree with each other in all traffic setting cases.

## 7. Conclusions and Future Work

Communication refinement is a crucial step in a NoC design flow. We have presented a refinement approach that allows us to map a perfectly synchronous communication model onto the NoC best-effort service accessible through communication primitives. Particularly we classify the synchronization properties of processes and describe methods to achieve synchronization consistency during the refinement upon the violation of the perfect synchrony hypothesis. For feedback loops, we relax the synchronization with the tolerance of system requirements. In this paper we use Nostrum as target, but with few adjustments, this approach is also applicable for other NoC platforms.

In future work, we plan to develop formalism for synchronization consistency and realize automatically analyzing the synchronization properties of processes. During refinement, we take either automatic analysis that yields correct synchronization and system behavior, or manual analysis with design decisions on the synchronization refinement combined with a systematic verification of the resulting implementation. For the refinement of feedback loops, we intend to use the Nostrum GB service to reach a systematic solution.

## References

[1] A. Benveniste, B. Caillaud, and P. L. Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.

[2] A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction de-

ployment. In *Proceedings of the Third International Conference on Embedded Software*, 2003.

[3] J.-Y. Brunel, W. Kruijtzer, H. Kenter, F. Petrot, L. Pasquier, E. de Kock, and W. Smits. COSY communication IP's. In *Proceedings of the 37th Design Automation Conference*, Los Angeles, California, June 2000.

[4] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, Sept. 2001.

[5] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. IPSIM: SystemC 3.0 enhancements for communication refinement. In *Proceedings of Design Automation and Test in Europe*, 2003.

[6] R. Dömer, D. D. Gajski, and A. Gerstlauer. SpecC methodology for high-level modeling. In *Proceedings of the Ninth IEEE/DATC Electronic Design Processes Workshop*, April 2002.

[7] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–303, December 2003.

[8] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 1995.

[9] C. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee. Standards for system-level design: practical reality or solution in search of a question? In *Proceedings of Design Automation and Test in Europe*, 2000.

[10] Z. Lu, I. Sander, and A. Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis*, October 2002.

[11] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the Design Automation and Test Europe Conference (DATE)*, 2004.

[12] E. Nilsson, M. Millberg, J. Öberg, and A. Jantsch. Load distribution with the proximity congestion awareness in a network on chip. In *Proceedings of the Design Automation and Test Europe (DATE)*, pages 1126–1127, 2003.

[13] R. Thid, M. Millberg, and A. Jantsch. Evaluating NoC communication backbones with simulation. In *Proceedings of the IEEE NorChip Conference*, 2003.

# Paper 9

**Towards performance-oriented pattern-based refinement of synchronous models onto NoC communication**

*Proceedings of the 9th Euromicro Conference on Digital System Design (DSD'06)*, pages 37-44, Dubrovnik Croatia, August 2006.

# Towards Performance-oriented Pattern-based Refinement of Synchronous Models onto NoC Communication

Zhonghai Lu, Ingo Sander and Axel Jantsch

Department of Electronic, Computer and Software Systems
Royal Institute of Technology, Sweden
{zhonghai,ingo,axel}@imit.kth.se

## Abstract

*We present a performance-oriented refinement approach that refines a perfectly synchronous communication model onto Network-on-Chip (NoC) communication. We first identify four basic forms of NoC process interaction patterns at the process level, namely, producer-consumer, peers, client-server, and multicast. We propose a three-step top-down refinement method: channel refinement, protocol refinement and channel mapping. For the producer-consumer pattern, we describe it in detail. In channel refinement, we deal with interfacing multiple clock domains and use a stochastic process to model channel delay and jitter. In protocol refinement, we show how to refine communication towards application requirements such as reliability and throughput. In channel mapping, we discuss channel convergence and channel merge arising from channel overlapping. All the refinements have been conducted and validated as an integral design phase towards implementation in ForSyDe, a formal system-level design methodology based on a synchronous model of computation.*

## 1 Introduction

Network-on-Chip (NoC) is deemed to be a paradigm to tackle System-on-Chip (SoC) design challenges in the billion transistor era. Due to lack of scalability, a bus-based (single level or multi-level) architecture is becoming performance bottleneck to interconnect tens or even hundreds of microprocessor-sized heterogeneous resources. Most probably a bus-based design will be used at the local resource level and complemented with a network platform at the chip global level. Meanwhile the deep submicron technology limits the maximum synchronous region on a chip to a local resource area. Globally Asynchrony Locally Synchrony (GALS) is regarded as a future SoC synchronization mechanism. From the design methodology perspective, raising design abstraction to system level is considered to be indispensable to cope with relentlessly increasing design complexity.

Keutzer et al. discuss system-level design in [12]. They point out, that "to be effective a design methodology that addresses complex systems must start at high levels of abstraction". Particularly, a design methodology should separate (1) function (what the system is supposed to do) from architecture (how it does it) and (2) communication from computation. They "promote to use formal models and transformations in system design so that verification and synthesis can be applied to the advantage of the design methodology". These arguments not only support but also establish the foundations of ForSyDe. The ForSyDe [13, 14] methodology addresses the design of SoC applications. Starting with a formal system specification model that captures the system functionality at a high abstraction level, it provides formal transformation methods to refine the system model into an implementation model , which serves as a starting point for synthesis into HW and SW.

In this paper we present the top-down communication refinement method towards NoC communication in ForSyDe, focusing on techniques to satisfy communication reliability and to leverage throughput and network utilization. The related work is briefed in section 2. We then introduce the ForSyDe methodology in section 3, and the process communication patterns, our NoC platform and its services in section 4. In section 5, we discuss our incremental communication refinement steps, *channel refinement*, *protocol refinement* and *channel mapping*. A tutorial example is shown in section 6, followed by conclusion in section 7.

## 2 Related Work

Based on the isolation of communication from computation, a large body of work on communication refinement exists in the literature. Through the Virtual Component Interfaces (VCI) of the VSI Alliance [10], the COSY-VCC design flow [3] supports communication refinement

Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD'06)
0-7695-2609-8/06 $20.00 © 2006 IEEE

from specification, to performance estimation and to implementation. IPSIM [5] developed on top of SystemC 3.0 supports an object-oriented methodology and establishes two inter-module communication layers. The message box layer concerns generic and system-specific communication, while the driver layer implements higher level application-dependent communications. The SpecC methodology defines four levels of abstraction, namely at the specification, architecture, communication and implementation level, and the refinement transformations between them [6]. In the course of communication refinement, methods to allow architecture exploration and protocol selection can be found in [11] and [9], respectively. These works do not assume a synchronous specification, thus are not applicable to our context.

With synchronous communication, latency insensitive theory [4] targets synchronized HW design where synchronization can still be achieved even if interconnecting synchronous IP blocks experiences indefinite wire latencies; De-synchronization for SW design was addressed in [1]. Furthermore, some mathematical frameworks were developed to support refinement-based design methods. Benveniste et al. present a theoretical framework for modeling heterogeneous systems, and derive sufficient conditions to maintain semantic-preserving transformations when deploying a synchronous specification onto GALS and the loosely time-triggered architectures [2]. Another theoretical framework is proposed in [8] concerning the refinement of a polysynchronous specification, which allows multiple clocks instead of a single clock. All these works are complementary to our work but none of them provides a detailed refinement approach targeting a NoC platform. Furthermore, this paper concentrates on refinement techniques to satisfy performance requirements based on process interaction patterns.

# 3 The ForSyDe Methodology

## 3.1 The Design Process

The ForSyDe [13, 14] design process starts with the development of an abstract functional specification expressed in the functional language Haskell. This model is then refined inside the functional domain by a stepwise application of well defined design transformations into an efficient implementation model. As the implementation model is a refined version of the specification model, the same validation and verification methods can be applied to both models. In the partitioning phase, the implementation model is partitioned into HW and SW blocks, which are mapped on architectural components. Only now, in the code generation phase, we leave the functional domain to generate VHDL or C code for the HW and SW.

## 3.2 The Specification Model

The specification model follows the synchronous modeling paradigm. This paradigm is based on an elegant and simple mathematical model, which is the ground of synchronous languages such as Esterel, Signal, Argos and Lustre. The basis is the perfect synchrony hypothesis, i.e., both computation and communication take no observable time. In order to formally describe our synchronous computational model, we follow the denotational framework of Lee and Sangiovanni-Vincentelli [16]. They define signals as a set of events, where each event $e$ has a tag $t$ and a value $v$, i.e. $e = (t,v) \in T \times V$. As our system model is synchronous, $T$ is the set of natural numbers, and all signals have the same set of tags. In order to model the absence of an event, a data type $D$ can be extended into a data type $D_\perp$ by adding the special value $\perp$, which is used to model the absence of a value. Absent values are used to establish a total order of events when dealing with signals with different or aperiodic event rates.

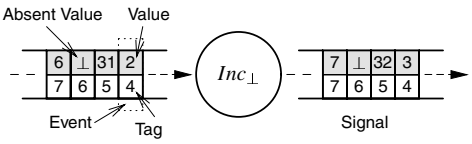


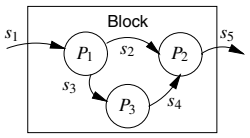

Figure 1: Modeling of signals and processes

Figure 1 illustrates the modelling of signals and the behavior of processes. At the event cycle $n$ a process evaluates the events of each signal with the tag $n$ and outputs the result at the same tag $n$.

We implement the synchronous computational model with the concept of *process constructors*. A *process constructor* is a higher order function that takes *combinational functions*, i.e. functions that have no internal state, and *values* as input and produces a process as output. There is a clean separation between *synchronization* (captured by process constructors) and *computation* (implemented by combinational functions). In addition, each process constructor has a structural *hardware* and *software* semantics which is used to translate the implementation model into a hardware/software implementation.

As an example, the process constructor *mealySY* models a finite state machine of Mealy type. It takes a function *ns* to calculate the next state as first argument, a function *out* to calculate the output as second argument and a value $s_0$ for the initial state as last argument. Thus a process $Mealy = mealySY(ns, out, s_0)$ implements the behavior of a finite state machine.

Processes can be glued together to build a network of processes. Such a network is called a block. Figure 2 shows how a block is formed by a network of processes. The function of a block is expressed by a set of equations.




$$
\begin{aligned}
Block(s_1) = \; & s_5 \\
\text{where} \quad & (s_2, s_3) = P_1(s_1) \\
& s_5 = P_2(s_2, s_4) \\
& s_4 = P_3(s_3)
\end{aligned}
$$

Figure 2: A network of processes

In the same way, blocks can be composed into higher level blocks, subsystems and eventually a hierarchical system.

### 3.3 Refinement of the Specification Model

The specification model abstracts from implementation details, such as buffer sizes and low-level communication mechanisms. This enables the designer to focus on the functional behavior of the system rather than structure and architecture. This abstract nature leaves a wide space for further design exploration and design refinement, which is supported by our transformational refinement technique. During the refinement phase the specification model is stepwise refined into a final optimized implementation model.

## 4 NoC Communication

### 4.1 Process Communication Patterns

A NoC application can be represented as a process network with a set of functional equations in ForSyDe. According to the interactions among processes [7], we identify the basic forms of inter-process communication patterns as follows:

- Producer-consumer. This is a one-to-one pattern where a producer generates data which in turn are consumed by a consumer. The consumer may or may not send back acknowledgments depending on application requirements.
- Peers. Similar to the producer-consumer, but both processes send/receive data and perhaps acknowledgments to/from each other.
- Client-server. This is a multiple-to-one pattern. Multiple clients send requests to a server which responds with various services. It works in a request-response manner in that a server will not respond to a client until a valid request is asking for service. The server may offer a uniform service or multiple services. For example, a memory only serves data read/write service. A microprocessor may provide various computing services such as remote procedure calls.
- Multicast. This is a one-to-multiple pattern. The group master actively reads/writes data to its multicast group members.

Next, we will take the producer-consumer pattern to demonstrate our communication refinement approach.

### 4.2 The NoC Platform and its Services

Our NoC platform [15] is a mesh structure composed of switches where each switch is connected to a resource, as shown in figure 3. The resources, which may work with different clock rates, are placed on the slots formed by the switches. The area of a resource is constrained within the maximal synchronous region in a given technology. The Resource-Network-Interfaces (RNIs) offer network communication services to resources.

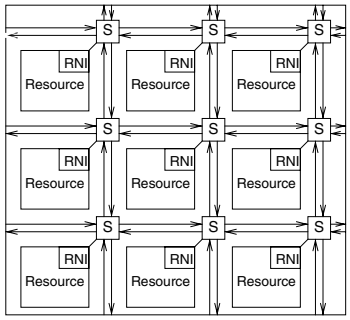



Figure 3: A NoC of mesh structure with 9 nodes

Our NoC platform provides two kinds of services. One is *best-effort* or connection-less delivery of messages. The other is connection-oriented *virtual circuit* that provides a resource-to-resource connectivity. In best-effort service, messages are routed in the network. The data sequence is maintained by re-ordering and data will not be lost. Neither the bandwidth nor the latency can be guaranteed. In virtual circuit services, messages are reliably delivered in order with guaranteed bandwidth. In terms of latency predictability, the services can be further classified as two kinds: *virtual circuit with latency commitment* and *virtual circuit with relaxed latency commitment*. In both cases, there is a bounded range with minimum and maximum latency value. For the second case, the upper bound is the worst case latency along the virtual circuit path.

Our NoC architecture provides a message passing platform. Processes communicate in the platform via channels which use one of the services. A message passing procedure between processes consists of three phases:

channel setup, data transmission and channel tear-down. If a channel intends to use the best-effort service, the channel setup is handled locally relying on an admission protocol in order not to saturate the network. If a channel asks for a virtual circuit service, the initiating process sends a setup message using the best-effort service to negotiate with the network for bandwidth and delay during the channel setup phase. Once the request is granted, the circuit path is fixed, and the bandwidth is reserved. The data transmission unit from process to process is message. A message comprises a channel identity number and payload. After the data transmission phase, the channel has to be explicitly torn down.

# 5 NoC Communication Refinement

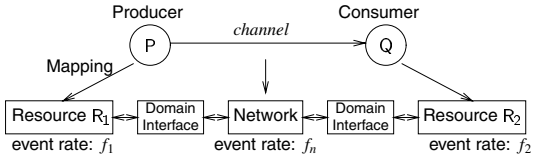
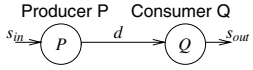



Figure 4: The producer-consumer and the target NoC

In figure 4, the upper part shows a producer-consumer pattern where the producer P communicates with the consumer Q via a *logical channel* [1]. The lower part shows a NoC instance with two resources $R_1$, $R_2$. In a GALS architecture, the resources $R_1$, $R_2$ and the network may work in different clock domains $f_1$, $f_2$ and $f_n$, respectively. We assume that all clocks have the same phase. The time structures from these clock domains have to be arbitrated when cross-domain communication is incurred. To this end we have explicitly highlighted the two domain interfaces, though they may be implemented as part of RNIs on resources. Our task is to map the producer-consumer onto the NoC. As for this pattern, the fundamental problem is data loss which occurs when the data-producing speed is higher than the data-consuming speed. Our refinement is not solving this problem. Instead we assume that the data-producing speed is not higher than the data-consuming speed. Also, the only meaningful read scheme for the consumer in this case is blocking read since the consumer can not react if no data is received.

In ForSyDe this pattern is initially modeled with a network of two processes, $P$ and $Q$, as shown in figure 5. Process $P$ models the producer P. Process $Q$ models the consumer Q. The two processes communicate in a perfectly synchronous manner via the signal $d$. A *signal* in the specification is to be mapped to a *service channel*.

[1] For convenience, we also call an arc connecting a pair of interacting processes a *channel*. It is logical and not associated with a service yet.




$$\text{where} \quad \begin{aligned} s_{out} &= QP(s_{in}) \\ d &= P(s_{in}) \\ s_{out} &= Q(d) \end{aligned}$$

Figure 5: The producer-consumer model in ForSyDe

## 5.1 Refinement Overview

Our objective is to refine this perfectly synchronous producer-consumer model onto the NoC communication services. The service selection is subject to the channel characteristics. The resultant producer and consumer model should fulfill application requirements such as reliability and throughput. For reliability, the producer asks from the consumer for acknowledgment for each message sent. For throughput, the producer-consumer has to make full use of the channel bandwidth honored during the channel setup. If the channel is not established, nothing will happen. Therefore we concentrate our refinement on the data transmission phase.

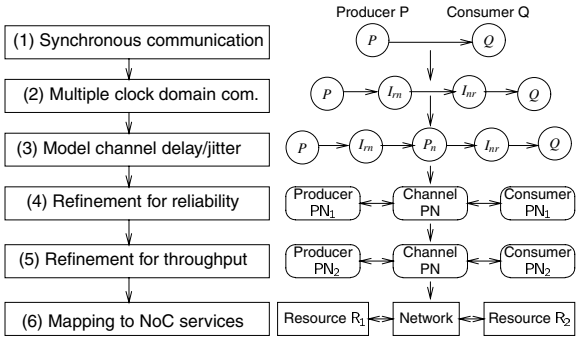



Figure 6: Refinement to NoC communication services

From now on we assume that the channel in figure 4 is granted with either of the two virtual circuit services. During the refinement steps we focus on how the process networks (PNs) will evolve. The overall refinement steps and the resultant process networks are illustrated in figure 6. The initial model (figure 5) is the perfectly synchronous model where there is only one clock domain. In step (2) we consider different clock domains and interfacing the clock domains. The process $I_{rn}$ models the domain interface connecting a resource to the network. The process $I_{nr}$ models the domain interface connecting the network to a resource. In step (3) we model the channel delay and jitter with the process $P_n$. We call the steps (2) and (3) *channel refinement* covered in section 5.2. In fact the channel refinement builds the channel model for refining the producer P and the consumer Q. In steps (4) and (5)

the process networks are refined to satisfy the reliability and throughput, respectively. We call the steps (4) and (5) *protocol refinement* covered in section 5.3. In step (6) covered in section 5.4 we discuss channel convergence and channel merge while mapping the channel to NoC communication services.

## 5.2 Channel Refinement

### 5.2.1 The clock domain interfaces

First of all we build models for the two clock domain interface processes $I_{rn}$ and $I_{nr}$. Introducing a synchronous sub-domain into the system model was presented in [13] where the clock rate of the sub-domain is $\frac{1}{n}$ ($n$ is a positive integer) of the main domain. Here we consider a generic domain interface that connects a clock domain with event/clock rate $f_1$ to another clock domain with event rate $f_2$. The simplest form of the fraction $\frac{f_1}{f_2}$ is $\frac{m}{n}$. The generic interface is constructed as $I_{f_1 \to f_2} = P_{dn}(m) \circ P_{up}(n)$, where $\circ$ is the composition operator. The processes, $P_{up}(n)$ and $P_{dn}(m)$, are formally defined as follows:

$$P_{up}(n)(\{x_1,x_2,\ldots\}) = \{\underbrace{\perp,\ldots,\perp}_{n-1},x_1,\underbrace{\perp,\ldots,\perp}_{n-1},x_2,\ldots\}$$

$$P_{dn}(m)(\{\underbrace{x_1,x_2,\ldots,x_m}_{m},\underbrace{x_{m+1},\perp,\ldots,\perp}_{m},\ldots\}) = \{x_m,x_{m+1},\ldots\}$$

The *up-sampling* process $P_{up}(n)$ samples out $n$ times of the input events, and does not result in event loss. The *down-sampling* process $P_{dn}(m)$ samples out $\frac{1}{m}$ times of the input events. At each down-sampling cycle, $m-1$ events are discarded and only the last valid event value (non-absent value) is kept. The interface first does up-sampling and then down-sampling. If $f_1 \leq f_2$, no event drop, hence no data is lost. If $f_1 > f_2$, events are cyclically dropped. But data may or may not be lost because the data rate may not match the event rate. If there is no data at an event cycle, only an event with absent value $\perp$ is inserted into the signal.

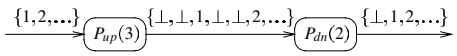



Figure 7: A clock domain interface

Figure 7 shows the interface process network for connecting the clock domain $f_1$ to the clock domain $f_2$ with the ratio $\frac{f_1}{f_2} = \frac{2}{3}$. Knowing the clock rates of the resources and the network, we can similarly build the interface processes $I_{rn}$ and $I_{nr}$. Our assumption is that the data-producing speed is not higher than the data-consuming speed. Besides, the NoC communication services guarantee that no data will be lost at the network. The two conditions guarantee that there is no data loss at the interfaces $I_{rn}$ and $I_{nr}$.

### 5.2.2 Model channel delay/jitter

We have assumed that the channel uses either of the two virtual circuit services fulfilling the bandwidth requirement. If viewing from a process's perspective, the net effect of delivering messages is delay and delay variances called jitter. To model the channel delay/jitter, we introduce stochastic characteristics to the network process $P_n$. The stochastic process $D_{[min,max]}$ generates a random delay within a given range $[min,max]$ for each event. A delay is modeled as an event with the absent value $\perp$. Figure 8 shows a stochastic delay process with the jitter range $[0,3]$.

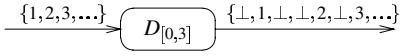



Figure 8: A stochastic delay process

After inserting the stochastic process, we receive a channel-refined producer-consumer, as shown in figure 9.

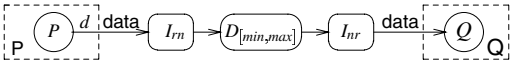



Figure 9: The channel-refined producer-consumer model

## 5.3 Protocol Refinement

### 5.3.1 The acknowledged producer-consumer

Although the channel is lossless and errorless, the consumer may be out of function or experience buffer overflow. In such a case, it is necessary for the producer to receive an acknowledgment before sending the next message in order to prevent the producer from overloading the network. This results in a feedback loop from the consumer to the producer shown in figure 10. If no acknowledgment is received, the producer will wait and not feed more data to the network, and the incoming data from the process $P$ will be silently dropped.

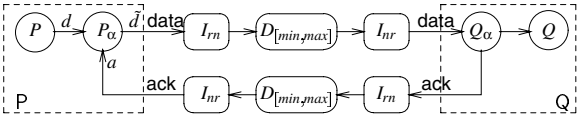



Figure 10: The acknowledged producer-consumer model

The processes $P_\alpha$ and $Q_\alpha$ in figure 10 implement the acknowledgment protocol. The process $P_\alpha$ has two states, *Idle* and *Wait*. It is modeled as a mealy FSM with the process constructor `mealySY` as follows:

$$\begin{array}{rrcl} & \tilde{d} & = & P_\alpha(d,a) \\ \text{where} & P_\alpha & = & mealySY(ns,out,Idle) \end{array}$$

The process $Q_\alpha$ receives data from the channel, then passes the data to the process $Q$ and generates acknowledgment.

At this step, the producer P is refined into the two processes $P$ and $P_\alpha$. The consumer Q is refined into the two processes $Q$ and $Q_\alpha$. The reliability is achieved through acknowledgment.

### 5.3.2 Buffering

In figure 10, during waiting for acknowledgment, the next incoming data will be silently dropped. To avoid this, a bounded FIFO buffer process $P_{buffer}$ is inserted between the process $P$ and the process $P_\alpha$. Data produced by the process $P$ is first pushed into the buffer. The process $P_\alpha$ is refined into the process $P_\beta$ that has an additional signal *readBuf* to read the buffer, as shown in figure 11. When the previously sent data is acknowledged, the process $P_\beta$ reads the buffer until successfully fetching data.

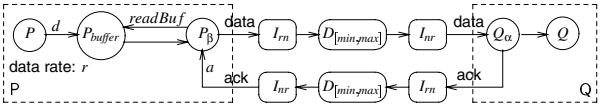



Figure 11: The acknowledged producer-consumer after buffering

It is easily seen that, if the data-emitting speed is higher than that of receiving acknowledgment, any bounded buffer will eventually overflow. The fastest data-emitting speed $r_{max}$ without buffer overflow is governed by the following formula:

$$\exists\, r_{max},\ r_{max} = \frac{f}{1+2\cdot D_{min}\cdot f}$$

where $D_{min}$ is the minimum channel delay and $f$ the producer's clock frequency. The minimum buffer size is 1.

At this step, the producer P is refined into the three processes $P$, $P_{buffer}$ and $P_\beta$ shown in figure 11. The consumer Q has no change.

### 5.3.3 Data pipelining

In figures 10 and 11, each message is individually acknowledged. The data transmission speed is limited by the variable channel delay. This leads to a waste of channel bandwidth, and, in some cases, data loss at the producer due to the buffer overflow. To solve this problem we elaborate the protocol. Instead of generating one acknowledgment for one received message, we can acknowledge a batch of data altogether. After sending a batch of data with size $w$, the producer waits for an acknowledgment from the consumer. Upon receiving the acknowledgment for the $w$ data, the producer starts to emit the next batch of data into the channel. In this way, the channel utilization is largely improved. The maximum allowable data-emitting speed without buffer overflow can be increased with a factor of nearly $w$, but no more than the channel capacity. The window size $w$ is affected by the channel delay and bandwidth, and the consumer buffer size. It is initially determined during the channel setup phase. Later it may be dynamically adjusted in case of network congestion control. Further, we can even improve the channel throughput by *prediction*. We assume an acknowledgment will come at the right time, thus we can first emit $2\times w$ size of data before waiting for the acknowledgment. If the acknowledgment for the first $w$ data comes in time, the producer starts to emit the third batch, and so forth. Otherwise, the producer has to wait. Compared with the previous pipelining, this will improve the channel throughput by up to a factor of two leading to a fully data-pipelined channel.

To accomplish the data pipelining, we need a counter process at both the producer and the consumer side, shown in figure 12. The counter process $Q_{counter}$ at the consumer side counts up to the window size $w$ and generates one acknowledgment. The counter process $P_{counterRst}$ at the producer side counts the number of sent data. If the window size $w$ or $2w$ is not reached, more data can be fetched from the buffer. In contrast to the process $Q_{counter}$, the process $P_{counterRst}$ is reset upon receiving an acknowledgment. That means, if an acknowledgment comes, it restarts to count from 0.

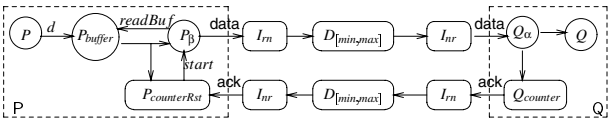



Figure 12: The acknowledged producer-consumer after windowing

At this step, the producer is refined into the four processes, and the consumer is refined into the three processes, shown in figure 12. Now the acknowledged producer-consumer can efficiently use the channel bandwidth. Our protocol refinement objective is thus achieved.

## 5.4 Channel Mapping

### 5.4.1 Channel convergence

After the protocol refinement, the producer and the consumer are mapped to their allocated resources. The channel uses the network communication services via the RNIs on the resources. Figure 13 shows two pairs of the non-acknowledged producer-consumer, $P_1$ and $Q_1$, $P_2$ and $Q_2$. The two producers $P_1$ and $P_2$ are mapped to the resource $R_1$. The two consumers $Q_1$ and $Q_2$ are mapped to the resource $R_2$. The two channels $ch_1$ and $ch_2$ use the virtual circuits $vc_1$ and $vc_2$ via $RNI_1$ and $RNI_2$, respectively.

One pre-condition for message passing is that the channel has to be established before communicating. Since no NoC platform can provide unlimited bandwidth, the number of channels which can be opened simultaneously is always limited. The channel setup may become the communication bottleneck. On the other hand, during the mapping of the producers and the consumers onto the NoC platform, some producers may be mapped to one resource and some consumers may be mapped to another resource, leading to overlapped channels. In some cases, if one virtual circuit can satisfy the latency requirement of these channels, and can provide enough bandwidth, the producers and the consumers can in fact share the virtual circuit. We call this *channel convergence*. In figure 13, if the latency of the virtual circuit $vc_1$ or $vc_2$ satisfies the latency requirements of the two channels, $ch_1$ and $ch_2$, and its bandwidth is not less than the sum of the two channel bandwidth, the two channels can share the virtual circuit $vc_1$ or $vc_2$.

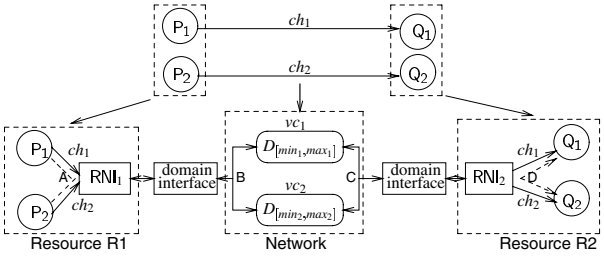



Figure 13: Channel mapping and channel convergence

### 5.4.2 Channel merge

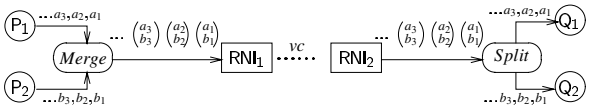



Figure 14: Channel merge and split

Further, if the message format can contain the payloads from the two channels, $ch_1$ and $ch_2$, there is a possibility of merging the two channels into one channel. One additional requirement is that the merged message format should be transparent to the destination processes, and be correctly split. In our refinement, we use the process *Merge* to realize merge, and the process *Split* to realize split, as illustrated in figure 14. This may decrease the overhead of arbitrating resource sharing, for example, at $\mathsf{RNI}_1$. In particular, it may benefit for synchronizing the two consumers $\mathsf{Q}_1$ and $\mathsf{Q}_2$. And one virtual circuit suffices.

# 6 A Tutorial Example

## 6.1 The Audio Amplifier

We use an audio amplifier as a tutorial example to illustrate our refinement steps. The amplifier regulates the audio input signal in response to the button levels. It is structurally decomposed into three functional blocks illustrated in figure 15. The audio sampling rate is 64K bps. There are two buttons "+" (Up) and "-" (Down) used to increase and decrease the amplification ratio, respectively. The maximum rate of button press is once per second.

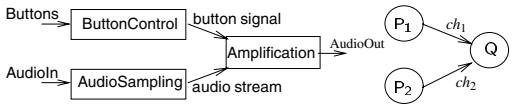



Figure 15: The audio amplifier and its process network

There are two channels $ch_1$ and $ch_2$. Both work as the producer-consumer pattern. The system requires acknowledgment for the audio channel $ch_1$. The button channel $ch_2$ does not need acknowledgment, but the button signals must be delivered in order (to keep causality) within tolerable period. In the system specification model, the audio output responds to the button press synchronously.

## 6.2 The Distributed Amplifiers

Analyzing the channel characteristics, we know that the two channels, $ch_1$ and $ch_2$, need to use a virtual circuit service. As we have refined the general producer-consumer model onto the virtual circuit services, we can choose one of the refined models for both channels. According to the system requirements, we adopt the producer-consumer model after channel refinement (figure 9) for the button channel $ch_1$, and the acknowledged producer-consumer with windowing (figure 12) for the audio channel $ch_2$.

Then we map the two refined producer-consumer models onto a NoC. Since the button channel uses less bandwidth, and the button signals contain less information bits, we assume that it is possible to converge and merge it with the audio channel. As a result, there are four choices for channel mapping:

(1) Map the producers $\mathsf{P}_1$, $\mathsf{P}_2$ and the consumer $\mathsf{Q}$ to three resources $\mathsf{R}_1$, $\mathsf{R}_2$ and $\mathsf{R}_3$, respectively. Both channels, $ch_1$ and $ch_2$, have their own virtual circuit, $vc_1$ and $vc_2$, respectively.

(2) Map the producers $\mathsf{P}_1$ and $\mathsf{P}_2$ to the resource $\mathsf{R}_1$, the consumer $\mathsf{Q}$ to another resource $\mathsf{R}_2$. Both channels, $ch_1$ and $ch_2$, maintain their own virtual circuit, $vc_1$ and $vc_2$, respectively.

(3) Mapping is the same as in case (2). But the two channels share one virtual circuit.

(4) Mapping is the same as in case (2). But the two channels are merged into one channel.

We only take case (2) to show the whole refined system model in figure 16. Ideally the model should be plugged into the interface models offered by the network service layer, specifically the RNI's model and the switch's model. Different models yield different results. In our experiments, we choose a parallel-to-serial conversion process $P/S$ if there is a need to arbitrate resource sharing, the points A and C. We choose a one-input two-output router process $R$ to separate channel messages according to the channel id, the points B and D. The refined models for the other three cases can be derived accordingly.

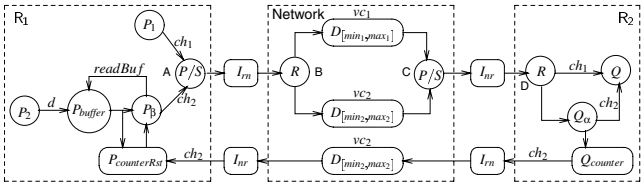



Figure 16: The refined amplifier model (Case (2))

Given the same set of parameters such as channel delays and window sizes, we have compared the four cases in terms of the average response delay of a button press to the amplification. In cases (1) and (2), the button signals may arrive ahead of the audio stream sampled during the buttons pressed, resulting in amplifying the previous sent audio data. This is because the two channel messages are delivered in different virtual circuits independently. In cases (3) and (4) the two channels can be synchronized since they share one virtual circuit. We can at least conclude that sharing virtual circuit may facilitate synchronization between channels. However, the virtual circuit bandwidth has to be high enough (case (3)), and the message format must be able to contain enough information (case (4)). Alternatively, data compression and de-compression may be introduced.

# 7 Conclusion

With the producer-consumer interaction pattern, this paper presents refinement procedures from perfectly synchronous communication onto NoC communication. During the refinements, application requirements such as reliability and throughput are satisfied. In ForSyDe, all the refinements are conducted within the functional domain, and are an integral design phase towards implementation.

The future work will focus on the refinements for the other three process interaction patterns defined in section 4.1, and consider the mixed effects in an application process network comprising two or more of the four process interaction patterns.

# References


[1] A. Benveniste, B. Caillaud, and P. L. Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.

[2] A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *Proc. of 2003 Conference on Embedded Software*.

[3] J.-Y. Brunel, W. Kruijtzer, H. Kenter, F. Petrot, L. Pasquier, E. de Kock, and W. Smits. COSY communication IP's. In *Proceedings of the 37th Design Automation Conference*, June 2000.

[4] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):18, September 2001.

[5] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. IPSIM: SystemC 3.0 enhancements for communication refinement. In *Proceedings of Design Automation and Test in Europe*, 2003.

[6] R. Dömer, D. D. Gajski, and A. Gerstlauer. SpecC methodology for high-level modeling. In *Proceedings of the Ninth IEEE/DATC Electronic Design Processes Workshop*, April 2002.

[7] G. R. Andrews. Foundations of Multithreaded Parallel and Distributed Programming *Addison Wesley Longman, Inc.*, 2000.

[8] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–303, December 2003.

[9] P. 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.

[10] C. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee. Standards for system-level design: practical reality or solution in search of a question? In *Proceedings of Design Automation and Test in Europe*, March 2000.

[11] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Depretter. System level design with SPADE: an M-JPEG case study. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2001.

[12] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[13] I. Sander and A. Janstch. Transformation Based Communication and Clock Domain Refinement for System Design. In *Proc. of the 39th Design Automation Conference*, 20(1):281–286, June 2002.

[14] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17-32, Jan. 2004.

[15] S. Kumar et. al.. A Network on Chip Architecture and Design Methodology, In *IEEE Computer Society Annual Symposium on VLSI*, 2002.

[16] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

A List of Doctoral Theses in **Electronic System Design** from the Department of Electronic, Computer and Software Systems, KTH:

- Tawfik Lazraq. **Design Techniques and Structures for ATM Switches**, ISBN 91-7170-703-4, 1995.
- Bengt Jonsson. **Switched-Current Circuits: from Building Blocks to Mixed Analog-Digital Systems**, ISRN KTH/ESD/AVH–99/1–SE, 1999.
- Johnny Öberg. **ProGram: A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols**, ISRN KTH/ESD/AVH–99/3–SE, 1999.
- Mattias O'Nils. **Specification, Synthesis and Validation of Hardware/Software Interfaces**, ISRN KTH/ESD/AVH–99/4–SE, 1999.
- Peeter Ellervee. **High-Level Synthesis of Control and Memory Intensive Applications**, ISRN KTH/ESD/AVH–2000/1–SE, 2000.
- Henrik Olson. **Algorithm-to-Architecture Refinement for Digital Baseband Radio Receivers**, ISRN KTH/ESD/AVH–2000/2–SE, 2000.
- Bengt Oelmann. **Asynchronous and Mixed Synchronous/Asynchronous Design Techniques for Low Power**, ISRN KTH/ESD/AVH–2000/6–SE, 2000.
- Yonghong Gao. **Architecture and Implementation of Comb Filters and Digital Modulators for Oversupplying A/D and D/A Converters**, ISRN KTH/ESD/AVH–2001/1–SE, 2001.
- Lirong Zheng. **Design, Analysis and Integration of Mixed-Signal Systems for Signal and Power Integrity**, ISRN KTH/ESD/AVH-2001/3-SE, 2001.
- Per Bjuréus. **High-Level Modeling and Evaluation of Embedded Real-Time Systems**, ISRN KTH/IMIT/LECS/AVH–02/3–SE, 2002.
- Imed Ben Dhaou. **Low Power Design Techniques for Deep Submicron Technology with Application to Wireless Transceiver Design**, ISRN KTH/IMIT/LECS/AVH–02/4–SE, 2002.
- Ingo Sander. **System Modeling and Design Refinement in ForSyDe**, ISRN KTH/IMIT/LECS/AVH-03/03–SE, 2003.
- Andreas Göthenberg. **Modeling and Analysis of Wideband Sigma-Delta Noise Shapers**, ISRN KTH/IMIT/LECS/AVH-03/04–SE, 2003.
- Dinesh Pamunuwa. **Modelling and Analysis of Interconnects for Deep Submicron Systems-on-Chip**, ISRN KTH/IMIT/LECS/AVH-03/07–SE, 2003.
- Bingxin Li. **Design of Multi-bit Sigma-Delta Modulators for Digital Wireless Communications**, ISRN KTH/IMIT/LECS/AVH-03/10–SE, 2003.

- Li Li. **Modelling, Analysis and Design of RF Mixed-Signal Mixer for Wireless Communications**, ISRN KTH/IMIT/LECS/AVH-04/12–SE, 2004.
- Abhijit Kumar Deb. **System Design for DSP Applications with the MASIC Methodology**, ISRN KTH/IMIT/LECS/AVH-04/10–SE, 2004.
- Steffen Albrecht. **Sigma-Delta Based Techniques for Future Multi-Standard Wireless Radios**, ISRN KTH/IMIT/LECS/AVH-05/07–SE, 2005.
- Meigen Shen. **Concurrent Chip and Package Design for Multi Radio and Mixed-Signal Systems**, ISRN KTH/IMIT/LECS/AVH-05/09–SE, 2005.
- Xinzhong Duo. **System-on-Package Solutions for Multi-Band RF Front-End**, ISRN KTH/IMIT/LECS/AVH-05/08–SE, 2005.
- Wim Michielsen. **VCOs for Future Generation of Wireless Radio Transceivers**, ISRN KTH/IMIT/LECS/AVH-05/10–SE, 2005.
- Yi-Ran Sun. **Generalized Bandpass Sampling Receivers for Software Defined Radio**, ISRN KTH/ICT/ECS/AVH-06/01–SE, 2006.
- Adam Strak. **Timing Uncertainty in Sigma-Delta Analog-to-Digital Converters**, ISRN KTH/ICT/ECS/AVH-06/12–SE, 2006.
- Petra Färm. **Integrated Logic Synthesis Using Simulated Annealing**, ISRN KTH/ICT/ECS/AVH-07/01–SE, 2007.
- Zhonghai Lu. **Design and Analysis of On-Chip Communication for Network-on-Chip Platforms**, ISRN KTH/ICT/ECS/AVH-07/02–SE, 2007.