LECTURE NOTES IN ELECTRICAL ENGINEERING 34

Srinivasan Murali

Designing Reliable and Efficient Networks on Chips

Springer

Designing Reliable and Efficient Networks on Chips

Lecture Notes in Electrical Engineering
Volume 34

For other titles published in this series, go to
www.springer.com/series/7818

Srinivasan Murali

Designing Reliable and Efficient Networks on Chips

Springer

Dr. Srinivasan Murali
INF 331, Station 14, EPFL
1015 Lausanne
Switzerland
srinivasan.murali@epfl.ch

ISBN 978-1-4020-9756-0 e-ISBN 978-1-4020-9757-7

DOI 10.1007/978-1-4020-9757-7

Library of Congress Control Number: 2008944292

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Preface

The complexity of *Multiprocessor Systems on Chips* (*MPSoCs*) is growing rapidly with the advances in semiconductor technology. The number of processors, hardware cores, and memories on a single chip is increasing and a highly-scalable communication infrastructure is required to connect them. To effectively tackle the interconnect complexity of current and future MPSoCs, a communication-centric design approach, *Networks on Chips* (*NoCs*), has recently emerged. NoCs bring the networking principles for data transfer, such as those used in large area networks (e.g., the Internet), to the on-chip domain.

Developing NoC-based systems tailored to a particular application domain, satisfying the application performance constraints with minimum power-area overhead is a major challenge. With technology scaling, as the geometries of on-chip devices reach the physical limits of operation, another important design challenge for NoCs will be to provide dynamic (run-time) support against permanent and intermittent faults that can occur in the system.

The purpose of this book is to provide state-of-the-art methods to solve some of the most important and time-intensive problems encountered during NoC design. We present methods for topology synthesis, mapping of cores onto NoC topologies, crossbar sizing, route generation, resource reservation, achieving fault-tolerance, RTL code, and layout generation. We show how the different design methods can be integrated to make a complete tool flow for designing reliable and efficient NoCs for application-specific MPSoCs and chip multiprocessors. To have less design respins and faster time-to-market, we show how the architectural synthesis models can be integrated with back-end physical design tools and models, thereby bridging a big design gap in on-chip interconnect synthesis.

Key features of book:

- Presents in depth the state-of-the-art algorithms and optimization models for performing system-level design of NoCs
- Presents an integrated flow to design interconnect architectures that can lead to faster time-to-market and design closure
- Shows evolution of design methods from complex crossbar based buses to NoCs
- Presents static and run-time methods for achieving reliable operation of the NoC and the entire system

This book should be of interest to:

- System level architects and designers: The methods show how to improve design productivity and achieve design closure of SoCs.
- Communication architecture/interconnect designers: The methods show trade-off analysis and explorations of NoCs.

- Design automation engineers: The high-level synthesis methods and mathematical models presented in this book can be applied to solve several communication architecture issues. They are also of general interest to designers working in related fields, such as sensor, body-area, and automotive networks.

This book is based on my Ph.D. research work done at Stanford University. I am greatly indebted to my adviser Prof. Giovanni De Micheli and co-adviser Prof. Luca Benini (University of Bologna), as they were instrumental in shaping the ideas presented here. The work is a result of collaboration with many researchers. I thank all my collaborators: Dr. Federico Angiolini and Antonio Pullini of iNoCs, Prof. David Atienza (EPFL), Dr. Kees Goossens and his team (Dr. Andrei Radulescu, Martijn Coenen, Andreas Hansson) at NXP research, Prof. Davide Bertozzi (University of Ferrara), Rutuparna Tamhankar (Marvell Technology), Prof. N. VijayKrishnan, Prof. Mary Jane Irvin and Dr. Theocharis Theocharides at Pennsylvania State University, Prof. Salvatore Carta, Paolo Meloni and Prof. Luigi Raffo of University of Cagliari for their contributions to this work.

EPFL, Lausanne, Switzerland Srinivasan Murali

Contents

Chapter 1
Introduction

Multiprocessor Systems on Chips (*MPSoCs*) are high-complexity, high-value semiconductor chips comprising general-purpose processors, hardware cores, DSPs, and memory blocks [1, 5]. Today, the commercial MPSoCs have several tens of cores on a chip (e.g., the NEC's TCP/IP offload engine is powered by 10 Tensilica Xtensa processor cores [71]), and in the next few years technology will support the integration of several tens to hundreds of cores, making a large computational power available.

Full exploitation of the increased level of SoC integration requires new paradigms and significant improvements of design productivity, as current system architectures and design styles do not scale up to such dimensions and complexities. A relevant example regards the system architecture, whose paradigm is progressively shifting from computation-centric to communication-centric. In fact, MPSoC performance will be increasingly determined by the ability of the communication infrastructure to efficiently accommodate the communication needs of the integrated computation resources.

1.1 Networks on Chips: Scalable Interconnects for SoCs

In several application domains, such as multimedia processing, the bandwidth requirement between the cores in SoCs is increasing. The aggregate communication bandwidth between the cores is in the GB/s range for many video applications. In the future, with the integration of many applications onto a single device and with increased processing speed of cores, the bandwidth demands will scale up to much larger values [7]. As an example of a media processing application, a Video Object Plane decoder [8] is shown in Figure 1.1. Each block in the figure corresponds to a core and the edges connecting the cores are labeled with bandwidth demands of the communication between them. As seen from the figure, the bandwidth demands are in the order of hundreds of MB/s.

Traditionally, bus-based architectures have been used to interconnect the various cores of the MPSoCs. To meet the increasing communication demands, the bus-based architectures have evolved over time from a single shared bus to multiple bridged buses and to crossbar-based designs. Current state-of-the art bus architectures, such as the AMBA multilayer [2], enable the instantiation of multiple buses operating in parallel, thereby providing a crossbar architecture. However, such an architecture is inherently nonscalable for large number of cores in the design.

To effectively tackle the interconnect complexity of current and future MPSoCs, a micro-networks based interconnect architecture is needed to connect the cores.

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

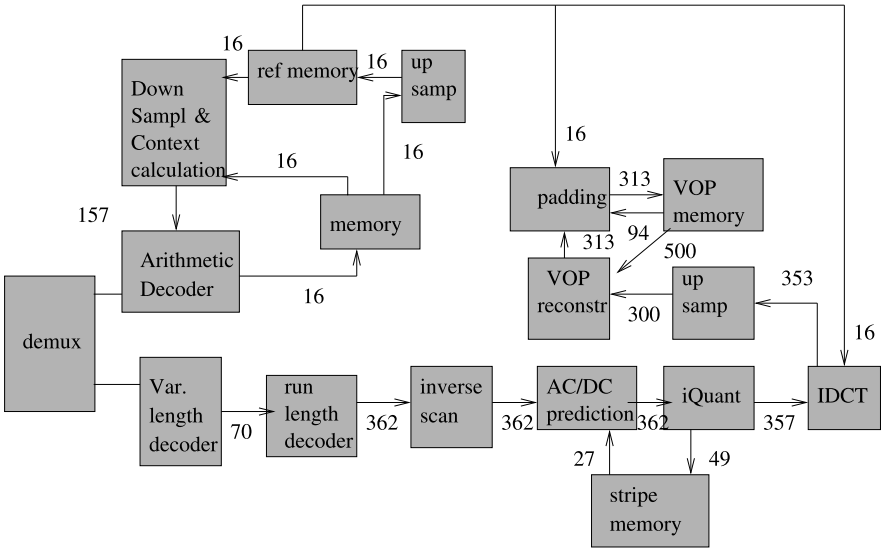


Fig. 1.1 Block diagram of Video Object Plane Decoder, with communication BW (in MB/s)

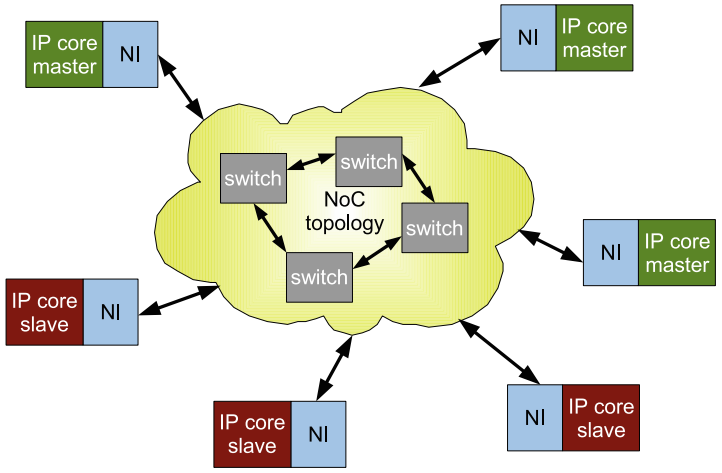


Fig. 1.2 Example NoC system with pipelined links

A communication-centric design approach, *Networks on Chips* (*NoCs*), has recently emerged as the design paradigm for designing such scalable micronetworks for MP-SoCs [20–23, 30, 31].

A typical NoC consists of switches, links, and Network Interfaces (NIs), as shown in Figure 1.2. A NI connects a core to the network and coordinates the transmission and reception of packets from/to the core. A packet is usually segmented into multiple *FLow control unITS* (*flits*). The switches and links are used to connect

the various cores and NIs together. To tackle the delay of long NoC links, a *latency insensitive design* approach in which the links are pipelined can be utilized [70]. Link pipelining increases the link throughput and decouples the cycle time of the communication system from the link length.

The use of a NoC to replace bus-based wiring has several key advantages:

- Better scalability at the architectural and physical levels. NoCs can add bandwidth as needed and segment wires as required.
- Better performance under high loads. NoCs can operate at high frequencies, cope with large bandwidth demands, and parallelize traffic streams.
- NoCs facilitate modularity by orthogonalizing the design of the communication architecture from the computation architecture, thereby leading to reduced design efforts.
- Quicker design closure. NoC are more predictable: They intrinsically provide wire segmentation, which helps ensuring that design respins will not be needed in the last phases of the design flow, when they are more costly.
- Higher energy-efficiency. To support the same traffic load, NoCs can operate at a lower frequency than bus-based systems and the data transfer can finish faster. These can lead to a reduction in energy consumption of the system.

Another effect of the shrinking feature size is that the power supply voltage and device V_t decreases and the wires become unreliable, as they are increasingly susceptible to various noise sources such as cross-talk, coupling noise, soft errors, and process variations [130]. The use of aggressive voltage scaling techniques to reduce the power consumption of the system further increases the susceptibility of the system to various noise sources. Moreover, wires are becoming thicker and taller, but their widths are not increasing proportionally, thereby increasing the effect of coupling capacitance on the delay of wires. As an example, the delay of a wire can vary between τ and $(1 + 4\lambda)\tau$ (where τ is the delay of the wire without any capacitive coupling and λ is the ratio of the coupling capacitance to the bulk capacitance) [136]. The wire delay for data transfer on a communication bus depends on the data patterns transferred on the bus. As presented in [121], the data-dependent variations in wire delay can be as large as 50% for the different switching patterns. With technology scaling, the device characteristics fluctuate to a large extent due to process variations and can cause significant variations in wire delay [122]. Wire delay is also affected by other forms of interference such as supply bounce, transmission line effects, etc. [123, 124]. Providing resilience from such transient delay and logic errors is critical for proper system operation.

The variability in process technology and temperature distribution (thermal hotspots) and the effect of various noise sources such as power supply fluctuations and electromagnetic radiations pose major challenges for the reliable operation of current and future MPSoCs. While some of these noise sources (such as thermal effects) cause intermittent or temporary failures in the system, some others (such as process variations) can cause permanent failures of hardware components. With the increased uncertainty of device operation, the time-to-failure period for the hardware components varies widely, with some components having a shorter

lifetime than expected. Therefore, new design methodologies and architectural solutions need to be developed to ensure proper system operation. NoCs facilitate the use of error recovery schemes developed for networks to achieve a reliable system operation.

1.2 NoC Design Challenges

Designing an efficient NoC architecture, while satisfying the application performance constraints is a complex process. The design issues span several abstraction levels, ranging from high-level application modeling to physical layout level implementation. Some of the most important phases in designing the NoC include: modeling the application traffic, synthesis of NoC topology for the application, mapping of cores onto the topology, finding paths and reserving resources, verifying performance of the system, developing simulation and synthesis models, and achieving reliable operation of the interconnect.

In order to handle the design complexity and meet the tight time-to-market constraints, it is important to automate most of these NoC design phases. To achieve design closure, the different phases should also be integrated in a seamless manner. The NoC design challenge lies in the capability to design hardware-optimized, customizable platforms for each application domain.

Computer-aided synthesis of NoCs is particularly important in the case of application-specific systems on chip, which usually comprise computing and storage arrays of various dimensions as well as links with various capacity requirements. Moreover, designers may use NoC synthesis as a means for constructing solutions with various characteristics that can be compared effectively only when a detailed model is available. Thus, synthesis of NoCs can be used for comparing prototypes. Needless to say, synthesis may also be very efficient for designing NoCs with regular topologies as, for example, multiprocessing systems with homogeneous cores.

NoC architectures are pushing the evolution of traditional circuit design methodologies to deal effectively with functional diversity and complexity. At the application level, the key design challenge is to expose task-level parallelism and to formally capture concurrent communication in models of computation [28]. Then high-level concurrent tasks have to be mapped to the underlying communication and computation resources. At this level, an abstract model of the hardware architecture is usually exposed to the mapping tool, so that area and power estimates can be given in the early design stage, and different objective functions (e.g., minimization of communication energy) can be considered to evaluate the feasibility of alternative mappings. In this context, a critical step in communication mapping is the NoC architecture synthesis for its significant impact on overall system performance, which is increasingly communication-dominated.

Finally, it is important to achieve a reliable NoC operation by providing resilience from permanent and transient delay and logic errors in the system. In order to protect the system from errors that occur in the communication subsystem, we can use error

recovery mechanisms that are used in traditional macro-networks. As the error detection/correction capability, area-power overhead, and performance of the various error detection/correction schemes differ, the choice of the error recovery scheme for an application involves multiple power-performance-reliability trade-offs that have to be explored.

1.3 Book Overview

In this book, we present methodologies to design reliable and efficient NoCs. We present algorithmic methods to solve many of the important NoC design problems. The novel, state-of-the-art optimization methods provide near optimal solutions for many of the NoC design problems. These methods will be useful for designers to tackle specific problems in NoC design or can even be applied to solve analogous problems in other domains. Most of the time-intensive steps of NoC design are automated and integrated into a complete tool flow. The tool flow can also be used to perform design space exploration of different communication architectures. The proposed tool bridges an important design gap that exists today, in building efficient communication architectures for MPSoCs.

In the rest of this section, a detailed overview of the book is presented.

1.3.1 NoC Design Methods

We first present (in Chapter 2) methods for synthesizing state-of-the-art crossbar based communication architectures. While methodologies that target the design of NoCs are required in the long run, providing design support for the state-of-the-art crossbar based bus designs pose an immediate and pressing problem. Also, as the NoC design process is more complex in nature, synthesis of crossbar-based communication architectures is an ideal starting point for illustration. Moreover, even in complex NoCs, the communication architecture will be hierarchical in nature, with local cores communicating through crossbars and the global communication taking place through a scalable network. In fact, this trend is already followed in many chip multiprocessors, such as the Stanford Smart Memories [101]. From Chapter 3 on, we present the design of NoC architectures.

The NoC topology defines the interconnection of the different network switches with the cores and among each other. The NoC topologies can be broadly classified into two main categories: standard and application-specific custom topologies. In the standard topologies, the interconnection structure ensures full connectivity between the cores: that is, any core is reachable from any other core. Examples of such topologies include mesh, torus, hypercube, Clos, and butterfly. In an application-specific custom topology, the interconnection between the switches and cores are optimized to match the application traffic patterns. If an application does not require

full connectivity between the cores, then the topology is optimized to provide only the required connectivity.

The use of a custom topology for an application almost always leads to a better performance and reduction in area/power overhead. However, there are some situations where a standard topology is desirable for the design:

- When the NoC is to be used across multiple product generations, a standard topology ensures that the same NoC can be reused easily across the different generations. However, when using a custom topology, the designer has restricted options when adding cores in the future, as the NoC may not provide full connectivity.
- When the cores are almost regular (similar sizes), the use of a standard topology leads to better wiring structure, as the floorplan is more predictable.

In this book, we address the design of both standard and application-specific custom topologies.

In Chapter 3, we present *Netchip*, a CAD tool flow for designing NoCs. The Netchip tool flow has three main phases and several tools integrated together:

- Front-End Design Phase: In this phase, several key NoC features such as the interconnect structure (or topology), routing scheme, paths for traffic flow, values for the NoC architectural parameters are determined. We present two tools: SUNMAP and SUNFLOOR to design application-specific standard and custom topologies, thereby automating this phase. The synthesis methods used in these tools are the subject of discussions in Chapters 4 and 5.
- Architectural Design Phase: In this phase, the RTL code of the NoC architecture is instantiated. For this, $\times$*pipes*, a library of soft-macros for the network components and $\times$*pipesCompiler*, a tool to generate the RTL design using the component library are developed. This is further explained in Chapter 3.
- Back-End Phase: In this phase, simulation, FPGA emulation, and layout generation of the NoC are carried out. For this, several standard industrial tool chains have been integrated with the tool flow, so that most of the back-end processes can be automatically obtained. This is also further explained in Chapter 3.

The NoC design process in Netchip is tuned to satisfy the requirements of the specific application that is to be run on the SoC. However, in today's systems, multiple applications (or use-cases) can run on the same chip. As an example, a set-top box SoC has multiple resolution video processing capabilities (like high definition, standard definition), multiple picture modes (like split-screen, picture-in-picture), video recording features, high speed Internet access, file transfer services, etc. In Chapter 6, we present the extensions to the synthesis process to handle the multiple use-case scenario. Even though, in Chapters 3–5, we present the NoC synthesis processes for the $\times$pipes architecture, the methods are quite general in nature and can be applied to any architecture. Toward this end, in Chapter 6, we show the process of designing the NoC for the Æthereal architecture (from NXP research).

So far, for the design process, we have assumed that the application traffic is statically known. However, what happens if there are large dynamic variations in traffic or if the traffic cannot be precharacterized at all? As an example, this would

be the case when tasks are assigned dynamically to the different cores. In Chapter 7, we present methods to design the NoC architecture for handling dynamic traffic patterns, while still yielding predictable performance.

Thus, in the first part of the book, we will be covering methods to synthesize NoCs under almost all possible design conditions.

1.3.2 NoC Reliability Mechanisms

In the second part, we will be presenting the different mechanisms that can be used to obtain a reliable NoC and system operation.

With technology scaling, the device characteristics fluctuate to a large extent due to process variations and can cause significant variations in wire delay [122]. Wire delay is also affected by other forms of interference such as supply bounce, transmission line effects, etc. [123, 124]. As such delay variations can affect multiple bits simultaneously, special mechanisms are needed to handle timing errors. In Chapter 8, we present *T-error*, a timing-error tolerant mechanism to make the interconnect resilient against timing errors arising due to such delay variations on wires.

Once the NoC components are made timing-error tolerant, we need to still handle other transient and permanent errors that can occur in the system, such as soft-errors. To handle such errors, we need support at the design level, as well as at the architectural level. In Chapter 9, we present an analysis of the power efficiency of traditional error detection/correction mechanisms, to choose the best scheme for the application, so that we can achieve the required reliability level with minimum area-power overhead. In Chapter 10, we present routing mechanisms that achieve an application-specific reliability level against temporary and permanent failures.

NoCs not only allow a reliable interconnect operation, but also facilitate achieving a reliable operation of the entire system. The high flexibility of NoCs allows the designer to add redundant cores in the same chip (e.g., processing elements, backup memories) without largely increasing the design complexity. In Chapter 11, we show how the NoC can be used to support the design of a reliable on-chip memory subsystem.

Finally, in Chapter 12, we conclude the book by integrating the reliability mechanisms with the design methods and showing how a complete NoC can be designed using the tool chain.

1.4 Related Work

In this section, we summarize some of the research that has been performed in the fields of NoC design and interconnect reliability.

The design issues in macro-networks (e.g., the LAN, WAN, Internet) have received unprecedented focus in the last several decades. In the last decade, the design

of chip-to-chip interconnection networks for parallel processing has also received considerable focus.

However, the challenges encountered in the design of on-chip networks for SoCs is quite different from the design of such macro-networks. Some major differences are: (1) The communication between the various cores can be statically analyzed for many SoCs and the NoC can be tailored for the particular application behavior. Whereas in the case of macro-networks, it is impossible to obtain a global knowledge of the traffic patterns of all the users. (2) The design objectives and constraints are different. As most SoCs are used in mobile and hand-held devices, having a network with minimum power consumption becomes an important design objective. Many SoCs also need to respond in real-time for certain inputs, for which the NoC has to support different criticality levels for the different traffic streams. (3) The design process should also consider VLSI issues, such as the structure (floorplan requirements) and wiring complexity of the resulting interconnect.

In this section, we present the state-of-the-art in the domain of NoC architectures, design methodologies, and fault-tolerant communication architectures.

1.4.1 NoC Architectures and Design Methods

The most advanced state-of-the-art SoC communication architectures represent evolutionary solutions with respect to shared buses. Sonics MicroNetwork [4] is a TDMA-based bus which can easily adapt to the data-word width, burst attributes, interrupt schemes, and other critical parameters of the integrated cores, while providing very high bandwidth utilization. STBus interconnect [3] is a high performance communication infrastructure that allows to instantiate shared buses as well as more advanced topologies such as partial or full crossbars. Although evolutionary from a topology viewpoint, these solutions can rely on advanced and highly automated design methodologies for the implementation of generic communication subsystems, allowing designers to rapidly assemble, synthesize, and verify their SoCs using the MicroNetwork or the STBUS interconnect as integration platforms.

However, the early works in [20, 22] pointed out the need for more scalable architectures for on-chip communication and, therefore, to progressively replace shared buses with on-chip networks. Many NoC architectures have therefore been proposed in the open literature so far, but in most cases, the design methodologies and tools are still in the early stage.

One of the earliest contributions in this area is the Maia heterogeneous signal processing architecture, proposed by Zhang et al., based on a hierarchical mesh network [24]. Unfortunately, Maia's interconnect is fully instance-specific. Furthermore, routing is static at configuration time and communication is based on circuit switching, as opposed to packet switching. In this direction, Dally and Lacy sketch the architecture of a VLSI multi-computer using 2009 technology [9]. A chip with 64 processor-memory tiles is envisioned. Communication is based on packet switching. This seminal work draws upon past experiences in designing parallel computers and reconfigurable architectures (FPGAs and their evolutions) [65, 66].

Most proposed NoC platforms are packet switched and exhibit regular structure. An example is a mesh interconnection, which can rely on a simple layout and the switch independence on the network size.

The *Scalable Programmable Integrated Network* (*SPIN*) described in [31] is a regular, fat-tree-based network architecture. It adopts cut-through switching to minimize message latency and storage requirements in the design of network switches.

The NOSTRUM network described in [32] also takes this approach: The platform includes both a mesh topology and the relative design methodology, wherein a concrete architecture is derived from a general NoC template, then application mapping follows.

The *Linkoeping SoCBUS* [69] is a two-dimensional mesh network which uses a packet connected circuit (PCC) to set up routes through the network: a packet is switched through the network locking the circuit as it goes. This notion of virtual circuit leads to deterministic communication behavior, but restricts routing flexibility for the rest of the communication traffic.

In [10], the use of *octagon* communication topology for network processors is presented. Instead, the implementation of a *star-connected* on-chip network supporting *plesiochronous communication* among system components is described in [27].

The *Æthereal* NoC design framework presented in [30] aims at providing a complete infrastructure for developing heterogeneous NoC with end-to-end quality of service guarantees. The network supports *guaranteed throughput* (*GT*) for real time applications and *best effort* (*BE*) traffic for timing unconstrained applications. Support for heterogeneous architectures requires highly configurable network building blocks, customizable at instantiation time for a specific application domain. For instance, the *Proteo NoC* [26] consists of a small library of predefined, parameterized components that allow the implementation of a large range of different topologies, protocols, and configurations. `×pipes` interconnect [61] and its synthesizer `×pipesCompiler` [62] push this approach to the limit, by instantiating an application specific NoC from a library of composable soft macros (network interface, link, and switch).

Today, several NoC architectures have been developed [42, 74–76] with each architecture having a different structure, switch/NI design, routing scheme, QoS support, and clocking methodology. In [73], the state-of-the-art in the NoC field is presented in detail.

The synthesis and instantiation of single bus and multiple bridged buses has been explored in many research works such as [35–38, 47, 77]. In [39], the authors present an approach for mapping the system's communication requirements and optimizing the protocols for a given communication architecture template. In [40], the use of communication architecture tuners to adapt to runtime variability needs of a system is presented. A floorplan aware method for designing point-to-point links and buses are presented in [77] and [53]. In [56], the authors present an exact approach to crossbar synthesis, where they integrate the NoC architecture parameter setting with the synthesis process.

Methods to collect and analyze traffic information that can be fed as input to the bus and NoC design processes have been presented in [39]. Mappings of cores onto

standard NoC topologies have been explored in [43, 44]. In [6], a unified approach to mapping, routing, and resource reservation has been presented.

Important research in macro-networks has considered the topology generation problem [41]. As the traffic patterns on these networks are difficult to predict, most approaches are tree-based (like spanning or Steiner trees) and only ensure connectivity with node degree constraints [41]. Hence, these techniques cannot be directly extended to address the NoC synthesis problem. Application-specific custom topology design has been explored in [25, 48–51]. In [49], a physical planner is used during topology design to reduce power consumption on wires. A method to obtain application-specific NoC topologies with floorplan estimation is presented in [78]. In [67], memory optimization in single chip network fabrics is explored. In [96], a tool flow to design NoCs with QoS guarantees is presented.

In [12], a low latency router architecture for supporting dynamic routing is presented. In [13], a routing scheme that switches between deterministic and adaptive modes, depending on the application requirements is presented. Several works in the multiprocessor field have focused on the design of efficient routing strategies [94]. In the Avici router [16], packets that need to be in-order at the receiver are grouped together into a *flow*. Packets of a single flow follow a single path, while different flows can use different paths. In the IBM SP2 network [17], source-based oblivious routing is used for a multi-stage interconnection network. In [14], the authors present a source-based dynamic routing algorithm for multi-stage networks. Building area and power models for on-chip networks has been addressed in [68, 79–81].

In this book, we present a streamlined design methodology for NoC topology synthesis that is completely integrated with the state-of-the-art commercial tools for back-end physical design. We present a floorplan aware topology design method for NoCs that leads to detecting timing violations on the NoC links early in the design cycle, with the resulting designs fully verified for timing correctness using standard place&route tools. Our custom NoC topology synthesis process guarantees a complete deadlock-free network operation without requiring special hardware mechanisms, which is critical for using NoCs in real designs. The topology synthesis process is integrated with NoC architectural parameter setting and uses accurate switch area, power models, and link power models that are obtained from layouts of the components. We address the design of both regular and custom NoC topologies and present methods to design NoCs under different application scenarios. Moreover, we also address the design of NoCs, when the application traffic characteristics cannot be predicted in advance. The presented design processes are both performance and power consumption aware, which are two of the important design objectives in SoC design.

1.4.2 Reliability Support for NoCs

The quest for reliable and energy efficient NoC architectures has been the focus of multiple researchers. Error protection can be applied at several levels within

a NoC design. For example, fault-tolerant routing algorithms have been proposed in [138, 140]. The use of nonintersecting paths for achieving fault-tolerant routing has been utilized in many designs, such as the IBM Vulcan [94]. The use of temporal and spatial redundancy in NoCs to achieve resilience from transient failures is presented in [15]. In this work, we present a fault-tolerant routing scheme and an associated design method for NoCs, which has low area-power overhead when compared to the existing schemes and is practical to be used in the on-chip domain.

A methodology for trading off power and reliability using error control codes for Systems on Chip (*SoC*) signaling is presented in [126, 147]. In [127], the energy behavior of different error detection and correction schemes for on-chip data buses is explored. In [129], a fault model notation is presented and the use of multiple encoding schemes for different parts of a packet is explored. In [128], the use of single error correction and parity based error detection schemes for NoCs is explored. Even though some of these works consider the use of error recovery schemes for NoCs, there is no framework available today for systematic analysis of the different error recovery schemes. To tackle this issue, we present a systematic power-reliability analysis methodology for the different error detection/correction schemes for NoCs. The presented method will be useful to NoC designers for choosing the appropriate error-recovery scheme for their applications.

In [139], the supply voltage is varied dynamically based on the error rate on the links. In [132], the data bus is monitored to detect adverse switching patterns (that increase the wire delay) and the clock frequency is changed dynamically to avoid timing errors on the bus. Many bus encoding techniques such as [134] have been proposed that decrease cross-talk between wires and avoid adversarial switching patterns on the data bus. There have been several approaches in the design space to detect and correct timing errors. The use of *double data sampling techniques* has been shown in self-checking testing circuits [116, 117] and for clock recovery in digital systems [120]. Recently, these techniques have been used for online timing and soft-error recovery in systems. The *TEAtime* [115] architecture tracks logic delay variations and dynamically adjusts the clock frequency to accommodate the changes in logic delay.

In *Razor* [113, 114], an aggressive, better than worst-case design approach is presented for processor pipelines. In this work, *double sampling* of data is used to control supply voltage (and hence power consumption) by monitoring the error rate. Favalli et al. [117] assume an encoded data signal which is checked by a small decoder present at the input of each flip-flop. In case of an error, the clock is delayed for one cycle, until the correct value of data settles. *Mousetrap* [118] is a high speed asynchronous pipeline which ensures correct data availability to consecutive stages. The *Iroc* [119] design uses a latch with delayed clock to detect transient faults due to soft errors. In [72], a method to reuse the scan flip-flops to achieve soft-error tolerance is presented. The method significantly reduces the soft-error rate of the system, with minimal overhead.

In this book, we present the application of the double sampling data technique to NoCs. By efficiently integrating the technique with the flow control of the NoCs, we show that large power/area savings can be achieved, when compared to the

general double sampling techniques. We also present ways to dynamically activate/deactivate the technique to adapt to the application error rates. We present novel methods for achieving error protection at both design level and architectural level. We also show how NoCs can be used to provide error resiliency for the entire SoC. Finally, we integrate the different error recovery methods presented in this thesis with the NoC design flow, thereby automating the design of fault-tolerant NoC architectures.

Part I
NoC Design Methods

Chapter 2
Designing Crossbar Based Systems

Over the last decade, the communication architecture of SoCs has evolved from single shared bus systems to multi-bus systems. Today, state-of-the-art bus based systems, such as the AMBA AXI [2] or the STBUS platform [3] supports the instantiation of crossbar matrices, where multiple buses operate in parallel, providing a high bandwidth communication infrastructure. While methodologies that target the design of NoCs are required in the long run, providing design support for the state-of-the-art crossbar based bus designs pose an immediate and pressing problem.

A crossbar matrix can be viewed as an evolutionary NoC architecture, where a single switch is used for the communication traffic flows. As the design process for building a general NoC is more complex in nature, synthesis of crossbar-based communication architectures is an ideal starting point for illustration of the design methods.

Despite some similarities, there is one important difference between the design of a crossbar matrix and a general NoC architecture. As the crossbar matrix design is simpler, exact algorithms can be utilized to build the system, thereby leading to fully optimum solutions. Even in cases where completely optimum solutions cannot be obtained, a large portion of the design space can be explored. Thus, we can design the crossbar system to handle more efficiently the local variations in traffic rates and burstiness in traffic flows, when compared to a general NoC system.

Even in complex NoCs, the communication architecture will be hierarchical in nature, with local cores communicating through crossbars and the global communication taking place through a scalable network. Thus, it is important to have efficient methods to design such crossbar systems.

In this chapter, we present the design of state-of-the-art crossbar based bus systems. We present methods to automatically design the *most power efficient crossbar* configuration for a MPSoC, satisfying the performance characteristics of the applications [52]. The communication architecture for the design should closely match the application traffic characteristics and performance requirements.

As an example, let us consider an image-processing MPSoC (detailed explanation of the MPSoC and experimental set-up is presented later in Section 2.5) with three different communication architectures used to connect the cores: a shared bus (all the cores are connected to a single bus), a full crossbar (each core is connected to a separate bus), and a partial crossbar (some of the cores share a bus). In Table 2.1, the average and maximum latencies incurred for a transaction (transfer of a single data word), obtained from SystemC simulation of the design using the different communication architectures are presented. The sizes of the crossbars (in terms of number of components used) normalized with respect to the size of the shared bus are also presented in the table. As seen from the table, as expected, both the average

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

Table 2.1 Crossbar performance and cost for an example image-processing MPSoC

Type	Average latency (cycles)	Maximum latency (cycles)	Size ratio
Shared bus	35	51	1
Full crossbar	6	9	11
Partial crossbar	10	20	4

and the maximum transaction latencies are much higher for a single shared bus than the partial or full crossbars. However, it is interesting to note that an optimal partial crossbar gives almost the same performance as a full crossbar, even though it uses fewer resources than a full crossbar.

The proposed design methodology is based on actual functional traffic analysis of the application, and the generated crossbar configuration is validated by cycle-accurate SystemC simulation of the application using that crossbar. Most previous works on bus generation and NoC topology generation (which are somewhat similar to crossbar generation) are either based on average communication traffic flow between the various cores or based on statistical traffic generating functions [43–51]. While the former approaches fail to capture local variations in traffic patterns (as the average bandwidth of communication is a single metric that is calculated based on the entire simulation time), the latter approaches are only based on approximations to the functional traffic.

The proposed design methodology differs from existing approaches [43–50] in the fact that it is based on the analysis of simulated traffic patterns in windows. The entire simulation period is divided into a number of fixed-sized windows. The crossbar is designed such that, within each window, the application communication requirements (such as the bandwidth requirements) are met. Moreover, the overlap among traffic streams mapped onto the same resource is minimized, thereby reducing the latency for data transfer. The criticality and real-time requirements of streams are also considered and the overlapping critical streams are mapped onto different crossbar resources.

The methodology spans an entire design space spectrum with the analysis based on average communication traffic (as done in many previous works [43–50]) and on-peak bandwidth (as done in [51]) being the two extreme design points. Thus, the methodology also applies to cases where application traces are not available and only rough estimates of the traffic flows between the various cores are known. The design point in the spectrum is varied by controlling the window size used for the traffic analysis and design.

We also integrate the setting up of several communication architecture parameters (such as the frequency of operation) with the crossbar synthesis phase. The wiring complexity of the interconnect should also be considered during the communication architecture synthesis procedure. For this, the floorplan of the design is performed, where the accurate physical locations of the cores and the crossbar matrix are determined. From the resulting floorplan, the wire-lengths in the design are

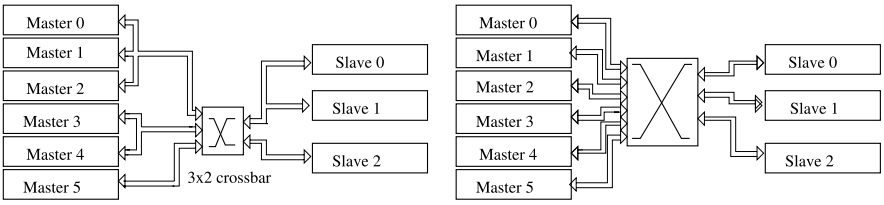
obtained. Based on the length of the wires and the operating frequency of the crossbar (which is automatically tuned by the synthesis procedure), any timing violations on the wires are obtained early in the design cycle. Thus, the crossbar architecture generated by the procedure is also validated for timing correctness, which is a key step to bridge the gap between the higher level architectural models and the actual physical design models of the crossbar architecture. From the wire-length estimates, accurate estimates of the power consumption of the interconnect wires are also obtained. The crossbar matrix power consumption values are based on the synthesis of the RTL models of the design, obtained using industry standard tools. From the wire and crossbar matrix power consumption, the total communication architecture power consumption is obtained, which is used to guide the synthesis procedure to obtain the most power efficient crossbar architecture.

We present experiments on several different MPSoC designs that show large reduction in power consumption of the communication architecture (45.3% on average) and total wire-length of the crossbar buses (38.0% on average) when compared to the traditional full crossbar based design approaches. Compared to the existing design methods, the proposed methodology results in crossbar platforms that lead to large reduction in transaction latencies (up to 7×). The experiments also show that the proposed approach is highly scalable to a large number of cores and to a large number of simulation windows in the design.

2.1 Problem Motivation and Application Traffic Analysis

2.1.1 Problem Motivation

There are three possible ways in which a crossbar can be instantiated: as a shared bus, a partial crossbar, or a full crossbar. The partial and full crossbars are actually composed of many buses to which the processor/memory cores are connected. Examples of partial and full crossbars are presented in Figure 2.1. In the partial crossbar architecture, some of the cores (such as the Master 0 and Master 1) share the same bus, while in the full crossbar, each core is connected to a separate bus. The objective of the crossbar synthesis procedure is to obtain an efficient clustering



(a) Partial crossbar (b) Full crossbar

Fig. 2.1 STbus crossbars

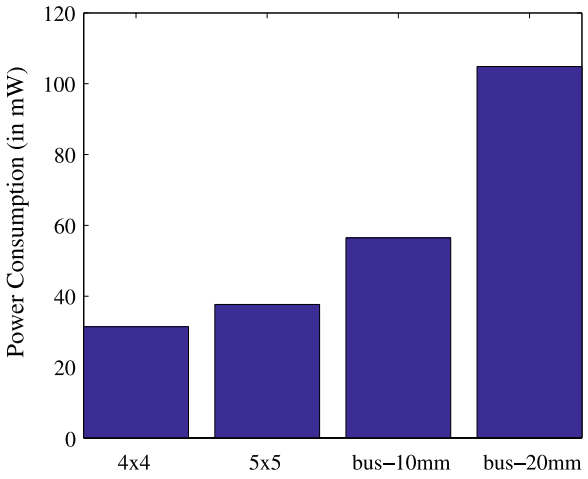
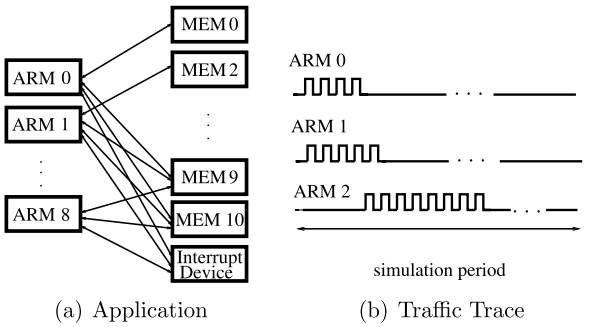


Fig. 2.2 Power consumption of switch matrix and wires

of the master and slave cores onto the crossbar buses, such that a communication architecture with low power consumption is obtained.

When choosing the most power efficient crossbar configuration, it is also important to account for the wiring complexity of the different configurations. As an example, the power consumption of the crossbar components (switch matrix and arbiters) for two different configurations and the power consumption of the wires for two different total wire-lengths (assuming a design with 30 cores and data width of 32-bits for the crossbar buses) are presented in Figure 2.2. For most MPSoC designs, the total length of the wires of the crossbar buses is of the order of few tens of millimeters. For the power consumption values presented in the figure, a 130 nm process technology is used, with an operating frequency of 500 MHz and an operating voltage of 1.2 V. The methods and assumptions used for estimating the power consumption of the crossbar matrices and wires are presented in detail in the experimental section. From the figure, we can infer that the wire power consumption is a significant fraction of the total communication architecture power consumption for crossbar based systems. Thus, it is important to consider the length of wires during the synthesis process, as the design point can be far from the optimum design point if such information is not accounted for. In order to have accurate wire-length estimates, we need to have accurate floorplan information of the design.

Another point worth noting is that in many crossbar architectures, the underlying protocol may not support pipelining of the buses (as an example, the Type 1 protocol of STbus [3]). In this case, the frequency of operation of the communication architecture is limited by the length of the longest bus in the design. For a chosen frequency point, it is then important to evaluate whether the length of the wires are lower than the threshold limit, so that they can be traversed in one clock cycle. We would also require the accurate floorplan and wire-length estimates to apply such feasibility checks.



(a) Application (b) Traffic Trace

Fig. 2.3 Application traffic analysis

2.1.2 Application Traffic Analysis

In this subsection, we explore the traffic characteristics of applications to model the performance constraints to be satisfied by the crossbar designed for the system. As an example, consider the 21-core image processing application, shown in Figure 2.3(a). In this example, there are 9 ARM cores, 11 on-chip memories, with some of the memories used for interprocessor communication and an interrupt device. The ARM cores act as masters and the memory cores act as slaves. The ARM cores run a set of image processing benchmarks that involve accesses to different memories. A cycle-accurate simulation of the system with a full crossbar design was performed, using the STbus crossbar architecture. A small trace of the traffic to three of the cores is shown in Figure 2.3(b).

Even though the aggregate traffic (measured over the entire simulation period) to the three cores is lower than that can be supported by a single bus, using a single bus to connect all three cores will lead to high average and peak latency due to overlap in traffic patterns during some regions of the simulation. Another related point is that if overlaps are not considered, connecting ARM 0 and ARM 1 on to the same bus is better than connecting ARM 0 and ARM 2 onto the same bus, as the former results in lower bandwidth needs. However, the latter solution will result in better performance (reduced transaction latency) while still satisfying the bandwidth needs. Note that using peak bandwidth instead of the average bandwidth will solve this problem, but lead to an over-design of the crossbar (in terms of number of buses needed or their frequency of operation). The design methodology needs to consider overlap among the various traffic streams into account and should consider local variations in traffic rates. Also, some of the traffic streams can be critical and to facilitate providing real-time guarantees; real-time traffic streams that overlap in time should not be mapped onto the same crossbar bus.

2.2 Design Methodology

The design flow for the crossbar design is shown in Figure 2.4, which consists of four distinct phases. In the first phase, the application is initially designed using a

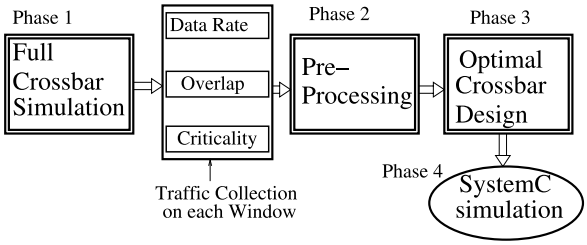


Fig. 2.4 Crossbar design methodology

full crossbar communication architecture and a SystemC simulation of the design is carried out. As the full crossbar architecture is nonblocking in nature (no contention between the cores if they are accessing different cores), it helps in modeling the application traffic requirements under ideal operating conditions. For the simulations, we use the *MPARM* simulation environment [57] that allows interconnection of ARM cores to several interconnection platforms (such as AMBA, STbus, . . .) and to perform cycle accurate simulations for a variety of benchmark applications.

To effectively capture local variations in traffic patterns and to perform overlap calculations, we define a window-based traffic analysis. The entire simulation period is divided into a number of windows and the traffic characteristics to the various cores in each window are obtained. The traffic characteristics recorded include: the amount of data sent and received by each core in every window, amount of pair-wise overlap between the traffic streams between the different cores in every window, the real-time requirements of traffic streams, etc. Without loss of generality, in the rest of this chapter, we assume that all the windows are of equal size, although the methodology also applies to windows with varying sizes. The size of the window is parameterizable and depends on the application characteristics and performance requirements.

After the data collection phase, a preprocessing phase is carried out in which the cores that have traffic flows with large overlaps in any window and need to be put on different buses are identified. In this phase, the overlapping critical streams that need to be on separate buses are also identified.

In the next phase, the optimal crossbar configuration for the application, satisfying the performance constraints is synthesized. To generate the optimal crossbar configuration, we use the traffic information collected in each window and check whether the bandwidth, overlap, and criticality constraints are satisfied in each window. In the final phase, the designed crossbar matrix is instantiated in the MPARM environment and SystemC simulations are carried out.

The details of the crossbar synthesis phase are presented in Figure 2.5. In the outer loop of the synthesis process, the communication architectural parameters (such as the frequency of operation and bus width) are varied in several user defined steps. The interesting range for the parameters are obtained from the user. For each architectural parameter point, the most power efficient crossbar configuration is synthesized. For synthesis, we present two approaches: one approach is based on solving the problem exactly using *Integer Linear Program* (*ILP*) formulation, which is applicable for small problem instances, and the other is a more scalable approach based on fast and efficient heuristics. In the next step of the synthesis phase,

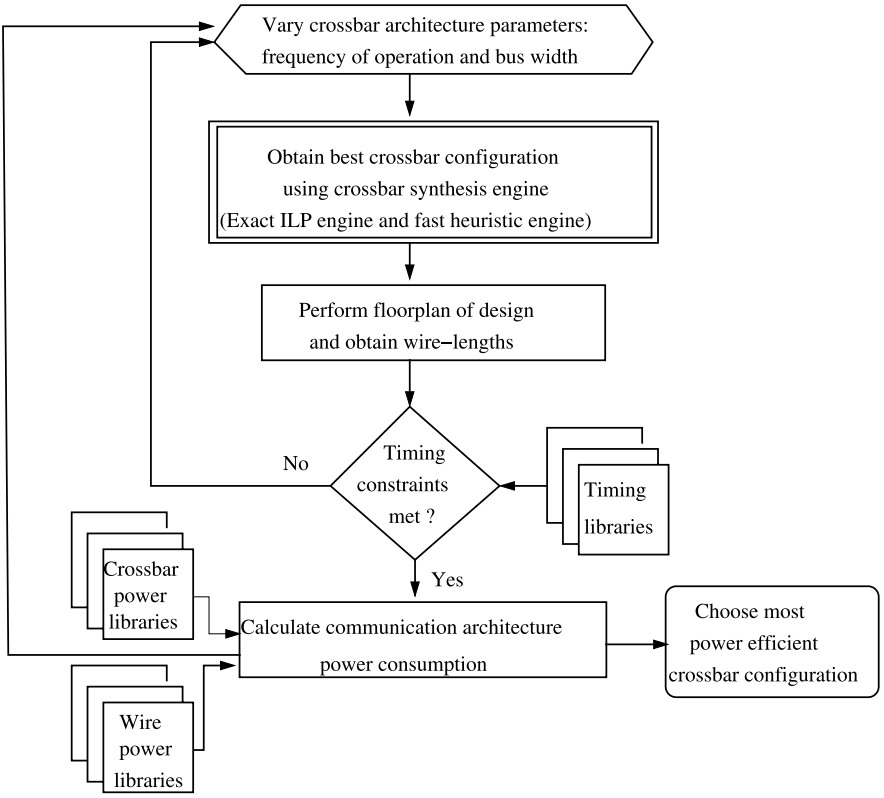


Fig. 2.5 The crossbar synthesis phase

floorplan of the synthesized design is performed. Floorplanning is the process of determining the exact 2D positions of the different cores and the switch matrix in the design. For obtaining the floorplan, we use Parquet [92], a fast and accurate floorplanner that minimizes the design area as well as the average wire-length. As the cores in the MPSoC are usually predesigned hardware blocks, we realistically assume that the size of the cores (either the width and height or the aspect ratio and area) are provided as an input to the synthesis process.

From the floorplan of the design, the length of the wires (based on the Manhattan distance), and hence the power consumption on the wires are obtained. In the next step, for the chosen frequency point, the wire-lengths are checked to see whether the maximum wire-length exceeds the length that the data can traverse in a single clock cycle. In the next step, from the switch matrix power consumption and the wire power consumption, the power consumption of the synthesized communication architecture is obtained. From the set of generated crossbar architectures for each architectural design point, the most power efficient architecture that satisfies the performance and timing constraints is chosen.

2.3 Exact Approach to Crossbar Synthesis

In this section, we formulate the mathematical models of the crossbar design problem and present the exact ILP formulation to synthesize the most efficient architecture for a chosen architectural parameter design point.

2.3.1 Problem Formulation

Definition 1 The set of all cores in the design is represented by the set T. The set of all windows used for traffic analysis is represented by the set W, with the bandwidth available (product of frequency of operation and bus width) in each window represented by WS. The set of buses used in the crossbar is represented by the set B.

Definition 2 The bandwidth requirement of each core t_i, $\forall i \in 1, \ldots, |T|$, in every window m, $\forall m \in 1, \ldots, |W|$, is represented by $comm_{i,m}$.[1] The amount of data overlap between every pair of cores (t_i, t_j) in each window m is represented by $wo_{i,j,m}$.

The overlap between every pair of cores t_i and t_j, over the entire simulation period is obtained by summing the overlap between them in all the windows and represented by the entries of the overlap matrix OM:

$$om_{i,j} = \sum_{m} wo_{i,j,m} \quad \forall i, j \tag{2.1}$$

In the preprocessing phase of the design flow (refer to Figure 2.4), those pair of cores that have overlap exceeding the threshold value (which is parameterizable) in any window are identified. By mapping the traffic flows of such cores onto separate buses, the maximum and average latency of data transmission can be reduced, and in some cases can also speed up the process of finding the optimal crossbar configuration. Also, in this preprocessing step, the real-time traffic streams that overlap with each other in any window are identified. Such cores with overlapping real-time streams should not be placed on the same bus, as real-time communication guarantee to the streams cannot be given in this case. Also, as noted earlier, most crossbar architectures do not allow masters and shared slaves of the design to be mapped onto the same bus. The set of all cores that cannot be on the same bus by the conflict matrix is defined by:

$$c_{i,j} = \begin{cases} 1, & \text{if } t_i \ \& \ t_j \text{ should be on different buses} \\ 0, & \text{otherwise} \end{cases} \quad \forall i, j \tag{2.2}$$

The performance constraints that need to be satisfied by the crossbar configuration in each window are modeled as constraints of an ILP.

[1] In the rest of this chapter, we follow the convention that variables i and j are defined for $1, \ldots, |T|$, variable k is defined for $1, \ldots, |B|$, and m for $1, \ldots, |W|$.

Definition 3 The set X represents the set of binding variables $x_{i,k}$, such that $x_{i,k}$ is one when core t_i is connected to the bus b_k and zero otherwise.

In the crossbar design, each core has to be connected to a single bus (while a single bus can connect multiple cores). This is implemented by the following constraint:

$$\sum_k x_{i,k} = 1 \quad \forall i \tag{2.3}$$

In every window used for traffic analysis, the individual buses of the crossbar have to support the traffic through them in that window. By evaluating the bandwidth constraints over a smaller sample space of a window (which is typically a few hundred or thousand cycles), instead of the entire simulation sample space (which can be millions of cycles), we are better able to track the local variations in the traffic characteristics.

This window-based bandwidth constraint is represented by the equation:

$$\sum_i comm_{i,m} \times x_{i,k} \leq WS \quad \forall k, m \tag{2.4}$$

Definition 4 The set SB represents the set of sharing variables $sb_{i,j,k}$, such that $sb_{i,j,k}$ is one when cores t_i and t_j share the same bus b_k and zero otherwise. The set S represents the set of sharing variables $s_{i,j}$, such that $s_{i,j}$ is one when cores t_i and t_j share any of the buses of the crossbar and zero otherwise.

The $sb_{i,j,k}$ can be computed as a product of $x_{i,k}$ and $x_{j,k}$. However, this results in nonlinear (quadratic) equality constraints. To break the quadratic equalities into linear inequalities, the following set of inequalities are used:

$$\begin{aligned} & sb_{i,j,k} \in \{0, 1\} \\ & x_{i,k} + x_{j,k} - 1 \leq sb_{i,j,k} \\ & 0.5x_{i,k} + 0.5x_{j,k} \geq sb_{i,j,k} \quad \forall i, j, k \end{aligned} \tag{2.5}$$

and the $s_{i,j}$ are computed using the equation:

$$s_{i,j} = \sum_k sb_{i,j,k} \quad \forall i, j \tag{2.6}$$

The condition that certain cores are forbidden to be on the same bus, obtained from equation (2.2), is represented by:

$$c_{i,j} \times s_{i,j} = 0 \quad \forall i, j \tag{2.7}$$

The fact that all the integer variables introduced above take values of either 0 or 1 only, is represented by:

$$x_{i,k}, s_{i,j}, c_{i,j} \in \{0, 1\} \quad \forall i, j, k \tag{2.8}$$

2.3.2 Exact Crossbar Synthesis Algorithm

The exact algorithm for the crossbar design has two major steps: the first is to find the best crossbar configuration that satisfies the performance constraints (that were presented in the above subsection) and the second step is to find the optimal binding of the cores to the chosen crossbar configuration.

In order to find the best crossbar configuration, we vary the number of buses in the design, from the maximum number (equal to the number of cores in the design, modeling a full crossbar) to one (modeling a single shared bus), in a binary search manner. For each configuration of bus count, we check whether a feasible solution that satisfies the constraints of the ILP (formed by the set of inequalities from equations (2.3) to (2.8)) exists. Once the minimum number of buses have been identified from applying the ILP, possibly multiple times, the buses used by the masters and slaves of the design are separated, thereby generating the optimal crossbar configuration.

Once the best crossbar configuration is obtained in the next step, the optimal binding of the cores onto buses of the crossbar is obtained. A binding of cores to the buses that minimizes the amount of overlap of traffic on each bus will result in lower average and peak latency for data transfer.

For this, the above ILP is solved with the objective of reducing the maximum overlap on each of the bus (the maximum overlap over all the buses is represented by the variable *maxov*), and satisfying the performance constraints as follows:

$$\begin{aligned} &\min\ maxov \\ &\text{s.t.} \quad \sum_i \sum_j om_{i,j} \times sb_{i,j,k} \leq maxov \quad \forall k \\ &\qquad \text{and subject to equations (2.3) to (2.8)} \end{aligned} \tag{2.9}$$

By splitting the problem into two ILPs, the execution time of the algorithm is reduced, as solving ILP 1 for feasibility check is usually faster than solving the ILP 2 with objective function and additional constraints. The ILPs are solved using the CPLEX package [59].

2.4 Heuristic Approach to Crossbar Synthesis

As the exact ILP approach is not scalable to large problem instances, either when the number of cores in the design is large or when the number of simulation windows used for analysis is large, in this section we present fast and efficient heuristic approach for crossbar synthesis.

The problem of assigning cores to the minimum number of buses, subject to the performance constraints is a special instance of the general problem of *constrained bin-packing* [60]. There are several efficient heuristics that have been developed for

the bin-packing problem [60]. In this work, we use an approach that is based on the *first-fit* heuristic to bin-packing. We chose this heuristic for several reasons. When the performance constraints are removed, the heuristic procedure is theoretically guaranteed to provide solutions that are within two times the optimum solution that would be obtained by an exact algorithm [60]. Practically, we found that the solutions obtained by the heuristic are close to the optimum solution possible for experiments on several SoC benchmarks. Moreover, the heuristics are relatively simple to implement and have a very low run-time complexity, making the approach scalable to large designs and allowing the use of large number of simulation windows for analysis.

The heuristic algorithm for crossbar synthesis is presented in Algorithm 1. In the first step of the algorithm, the bandwidth available in each simulation window is calculated. In the next step, all the cores are initialized as unmapped, as they are yet to be mapped onto buses. Then the number of buses in the crossbar is initialized to zero (step 5). In steps 6 to 25, the assignment of the cores onto the buses of the crossbar is performed. The basic approach used is the following: We try to map as many cores as possible onto a single bus. While mapping the cores, from the set of all cores that satisfy the bandwidth and conflict constraints, we choose the one that minimizes the pair-wise traffic overlap with the cores that have been already mapped onto the current bus. When no more cores can be assigned to the current bus, either because the bandwidth of the bus in any of the simulation window has been saturated, or because of conflicts with the cores already mapped onto the bus, a new bus is instantiated. The process is repeated until all the cores in the design have been mapped onto a bus.

From the resulting number of buses, the buses onto which masters are attached and those onto which the slaves are attached are separated. From this, the efficient crossbar configuration for the design is obtained.

Example 1 Let us consider a small example with 5 cores, with 3 of them being masters and the rest being slaves. For illustrative purposes, let us assume that two simulation windows are used for analysis (although in real systems usually several thousand windows are used). The communication traffic rates for each of the cores (in MB/s) for the two simulation windows are presented in Table 2.2 and the amount of traffic overlap between the different cores over all the windows is presented in Table 2.3. Let us assume that the current frequency design point is 100 MHz and the bus width is 32 bits, which are automatically tuned by the crossbar synthesis procedure (as presented in Figure 2.5). In the first step of the heuristic algorithm, the bandwidth of the bus in each simulation window is calculated to be 400 MB/s (frequency $\times$ data-width). Initially, a single bus is instantiated and core_0 is chosen to be mapped onto the bus, as it has the maximum bandwidth requirements of the different cores, across all the simulation windows (see Figure 2.6(a)).

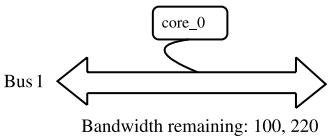
Then from the set of all cores, those cores that satisfy the bandwidth and conflict constraints are chosen. As cores that are masters and slaves are not allowed to be mapped onto the same bus (specified as part of the conflict constraints), the set of assignable cores to the bus are core_1 and core_2. From these two, core_2 is

Algorithm 1 Heuristic-synthesis(*frequency*, *buswidth*)

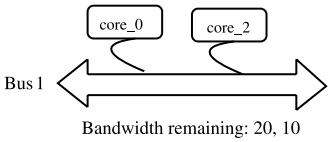
1: Bandwidth available in each window, $WS = frequency \times buswidth$
2: **for** $i = 1, \ldots, |T|$ **do**
3: mapped$(i) =$ false
4: **end for**
5: Initialize number of buses used, k to 0
6: **while** $\exists i \in 1, \ldots, |T|$, such that mapped$(i) =$ false **do**
7: Increment the bus count k by 1 and instantiate new bus. Initialize bandwidth available on bus on all windows: $BW(k, m) = WS, \forall m \in 1, \ldots, |W|$
8: Choose unmapped core $i, \forall i \in 1, \ldots, |T|$, with maximum bandwidth requirements on any window and map it onto bus k
9: Initialize the set *chosen_set* to ϕ
10: **for** $i = 1, \ldots, |T|$ **do**
11: **if** mapped$(i) =$ false and core i does not have conflicts with cores already mapped onto bus k **then**
12: $bw\_satisfied =$ true
13: **for** $m = 1, \ldots, |W|$ **do**
14: **if** $BW(k, m) < comm_{i,m}$ **then**
15: $bw\_satisfied =$ false
16: **end if**
17: **end for**
18: **if** $bw\_satisfied =$ true **then**
19: $chosen\_set = chosen\_set \cup i$
20: **end if**
21: **end if**
22: **end for**
23: Choose core $i, \forall i \in 1, \ldots, |chosen\_set|$, with minimum overlap with cores mapped onto bus k and map it to bus k. Update available bus bandwidth as: $BW(k, m) = BW(k, m) - comm_{i,m}, \forall m \in 1, \ldots, |W|$
24: Repeat steps 9–23 until *chosen_set* is empty
25: **end while**
26: Separate the buses onto which masters and slaves are mapped and generate the crossbar configuration

Table 2.2 Communication requirements of example system: (M-Master, S-Slave)

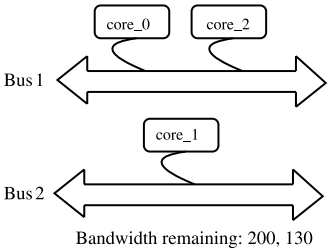
Name	Type	BW (win 1) MB/s	BW (win 2) MB/s
core_0	M	300	180
core_1	M	200	270
core_2	M	80	210
core_3	S	60	110
core_4	S	150	70



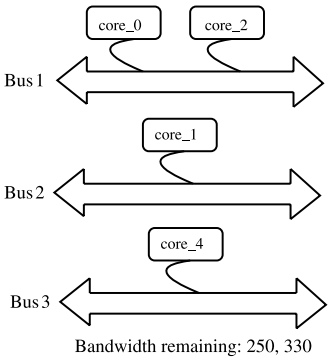
(a) Step 1. core 0 is chosen



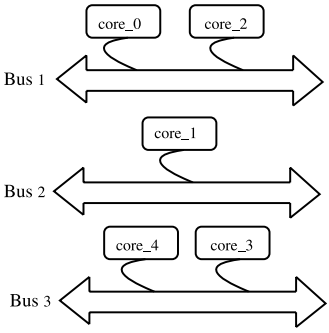
(b) Step 2. From emaining cores, core 2 is chosen.



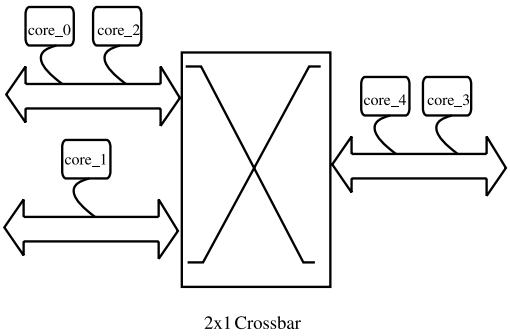
(c) Step 3. New bus is instantiated, core 1 mapped.



(d) Step 4. Another bus is instantiated.



(e) Step 5. The remaining core, core 4 is mapped.



(f) Step 6. The crossbar configuration, 2x1, is obtained from the instantiated buses.

Fig. 2.6 Example application of the heuristic algorithm

Table 2.3 Amount of traffic overlap between cores (in MB/s) of example system

	core_0	core_1	core_2	core_3	core_4
core_0	×	30	10	×	×
core_1	30	×	27	×	×
core_2	10	27	×	×	×
core_3	×	×	×	×	15
core_4	×	×	×	15	×

chosen, as it has minimum overlap with the cores already mapped onto the bus (i.e., with core_0) and assigned onto this bus (Figure 2.6(b)). When no more cores can be assigned to the current bus, a new bus is instantiated. The different steps of the procedure for the 5-core example are presented in Figures 2.6(a)–(f). At the end of the procedure, those buses that are used by the masters and those that are used by the slaves are separated, which gives the best crossbar configuration. In this example, we have 2 buses used by the masters and 1 used by the slaves, resulting in a 2×1 crossbar design, as shown in Figure 2.6(f).

2.5 Experiments and Case Studies

In this section, we present the experimental case studies performed to validate the proposed crossbar design methodology.

2.5.1 Experimental Platform and Power Models

For performing the SystemC simulations on MPSoC benchmarks, we use the MPARM simulation platform [57]. The platform is a representative of a large class of multiprocessor SoC platforms and consists of a configurable number of 32-bit ARM processors, memory cores, hardware devices or traffic generators, and a hardware interrupt unit. The platform allows the use of different interconnect architectures, such as the AMBA, STbus to interconnect the various hardware cores. It also supports a variety of MPSoC benchmarks that have been efficiently parallelized to run on the ARM cores.

For power consumption estimations of the switch matrix, we implemented several configurations of the AMBA multilayer crossbar, varying the number of input and output ports of the matrix. The different configurations were implemented using the AMBA DesignWare libraries obtained from Synopsys CoreAssembler tool [98]. The tool generates RTL code of the different configurations, which were then synthesized using Synopsys Design Compiler [98]. For synthesis, we utilize a 130 nm process technology, an operating voltage of 1.2 V and an operating frequency of 500 MHz. Based on the power consumption values obtained from the synthesis process, analytical models for the switch matrix power consumption are built using linear regression. During the crossbar design process, the power numbers from the analytical models are linearly scaled, based on the crossbar operating frequency (which is automatically tuned by the design process). We estimate the wiring capacitance and wire power consumption based on the models from [58]. The power consumption values of some of the crossbar components were presented earlier in Figure 2.2.

2.5.2 Application Benchmark Analysis

We apply the crossbar design methodology on several SoC designs implemented using the MPARM platform: *IMage Processing design 1* (*IMP1*-25 *cores*), *IMage Processing design-2* (*IMP2*-21 *cores*), *FFT based SoC* (*FFT*-29 *cores*), *Data Processing SoC* (*DP*-15 *cores*) *and SoC implementing a DES encryption system* (*DES*-19 *cores*). The traffic characteristics of the applications were scaled to project the traffic requirements of future MPSoCs, as presented in [31]. For traffic analysis, we use 1000 simulation windows for the different designs, with each simulation window accounting for few hundred simulation cycles.

The interesting range of operating frequencies and bus data-widths are obtained as inputs from the designer. Practically, the data-width of the bus is set-up based on the data-widths of the different processors in the design. In the SoC designs used here, all the processors have the same data-width (of 32-bits), and hence we feed this value as an input to the synthesis engine. The interesting range of operating frequencies are defined to be between 100 MHz to 500 MHz, with each frequency point being a multiple of 100 MHz. With this set-up, we apply the heuristic synthesis engine to design the crossbar architecture for the designs.

We first briefly analyze the quality of the crossbar design obtained for the IMP2 SoC design. The communication between the cores of the IMP2 design was presented earlier in Figure 2.3(a). The communication requirements of some of the cores for the first few simulation windows are presented in Table 2.4. In this benchmark, there are 9 ARM cores (ARM 0 to ARM 8), 11 memory cores (MEM 0 to MEM 10), and an interrupt device (INT). The ARM cores act as masters and the others are slave cores that respond to the requests of the masters. There is substantial temporal overlap between the traffic flows from the various ARM cores to the memories, as the ARM cores perform similar computations, and thus access their memories at almost the same time. The power consumption of the synthesized crossbar designs for the different frequency design points are plotted in Figure 2.7. As the maximum bandwidth requirements of most of the cores were above 800 MB/s, the minimum frequency design point that gives a feasible solution is 300 MHz (at 200 MHz, the available bus bandwidth of 800 MB/s cannot support the requirements of most cores). At lower operating frequencies (such as 300 MHz), a larger crossbar

Table 2.4 Traffic characteristics of IMP2

Core	Win. 1 (MB/s)	Win. 2 (MB/s)
ARM 0	810	210
ARM 1	740	234
MEM 0	790	150
MEM 1	730	220
MEM 9	180	50
MEM 10	180	50

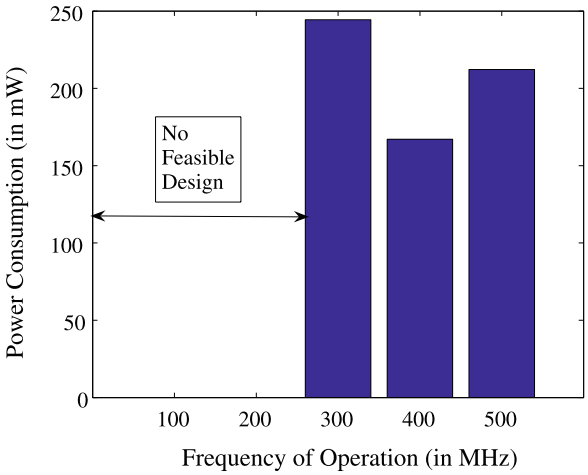


Fig. 2.7 Power consumption for different crossbar frequencies

configuration is required to satisfy the bandwidth constraints. A larger crossbar configuration usually also leads to an increased wiring complexity. These two factors coupled together results in larger power consumption for the communication architecture. At very high operating frequencies, the power consumption of the communication architecture is higher, as the power consumption increases linearly with the operating frequency of the system. For the IMP2 design, the crossbar architecture with lowest power consumption is obtained at 400 MHz.

The synthesized crossbar architecture (a 5×6 crossbar) for the IMP2 design is presented in Figure 2.8. In order to satisfy the window bandwidth constraints, only few of the cores can share a single bus, and thus each of the buses used in the crossbar have at most 2 cores attached to them. The bindings are such that the cores with highly overlapping streams are placed on different buses. As a result, the designed crossbar has acceptable performance (in terms of average and maximum latency constraints) with $1.9\times$ reduction in the number of buses used, when compared to a full crossbar. The floorplan of the IMP2 SoC with the designed crossbar, as obtained from the Parquet floorplanner is presented in Figure 2.9.

The size and power consumption of the synthesized crossbar architectures for the different SoC designs and for full crossbar configurations are reported in Table 2.5. The power consumption of both the switch matrix and the crossbar bus wires are reported in the table. The methodology results in a large reduction in the crossbar architecture power consumption (45.3% on average) when compared to the traditional full crossbar based systems. The synthesized crossbar configurations also lead to large reduction in the total length of the buses used in the design (38.0% on average, refer to Figure 2.11), as there are fewer buses in the design. Reducing wiring congestion is essential to have a faster physical design process and to achieve faster design closure.

The normalized average and maximum read/write transaction latencies (to read or write one data word) for the designs obtained using the methodology based on average traffic flows and using the proposed methodology (referred to as "slot" in the

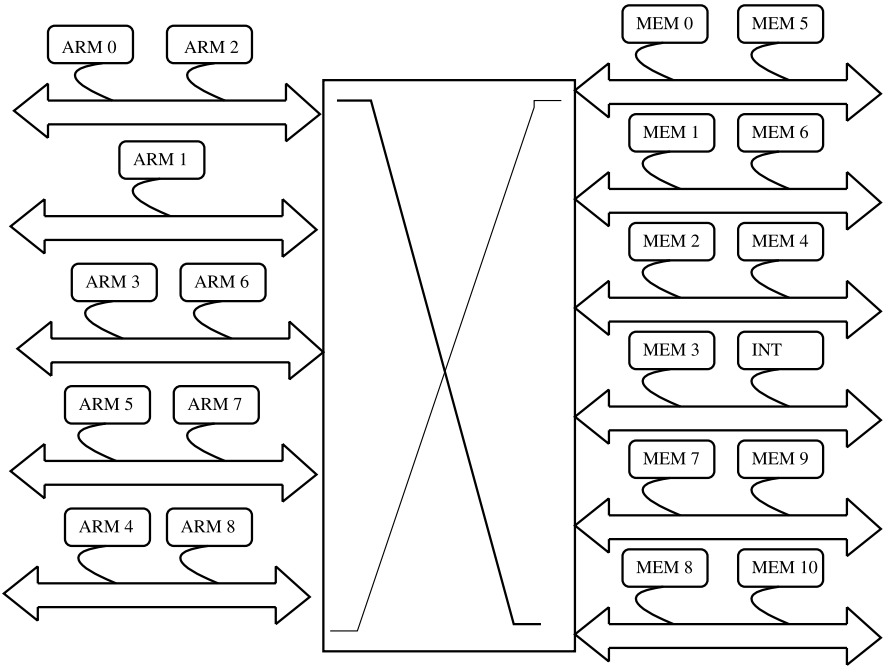


Fig. 2.8 Synthesized crossbar for the IMP2 SoC design

Fig. 2.9 Generated floorplan and buses for the IMP2 SoC design

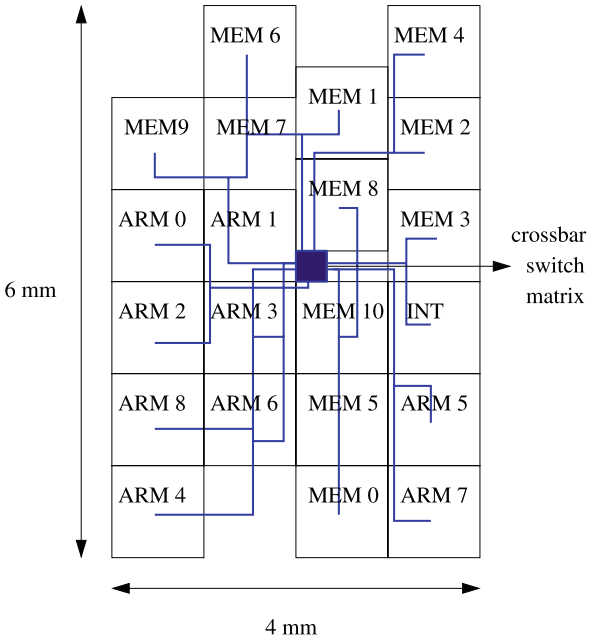
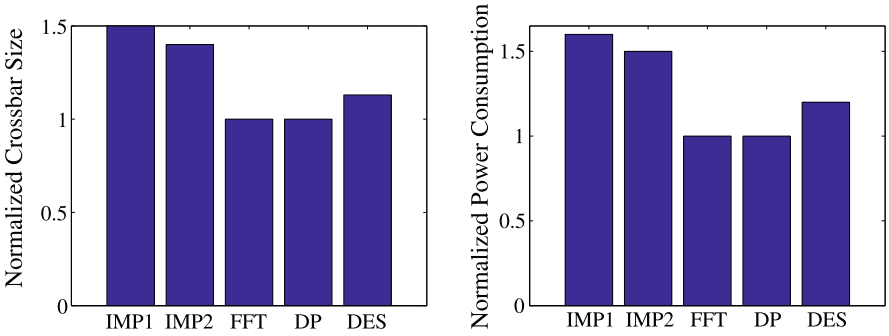


Table 2.5 Crossbar size and power consumption for SoC designs

Design	Full crossbar size	Synthesized crossbar size	Full crossbar power consumption (mW)			Synthesized crossbar power consumption (mW)		
			Matrix	Wire	Total	Matrix	Wire	Total
IMP1	11×14	6×7	156.7	228.0	384.7	60.2	146.1	206.3
IMP2	9×12	5×6	128.4	198.2	326.6	45.2	125.0	170.2
FFT	13×16	7×8	175.1	301.4	476.5	75.9	191.8	276.7
DP	6×9	3×5	38.7	51.3	90.0	12.1	36.9	49.0
DES	8×11	4×6	56.0	82.1	138.1	18.8	54.1	72.9



(a) Crossbar Size (b) Crossbar Power Consumption

Fig. 2.10 Comparisons of heuristic engine vs. exact engine

figures, signifying the use of the proposed slot or window based methodology), are presented in Figures 2.12(a) and (b). As seen from the figures, the latencies incurred by crossbar designs based on average traffic flows are $4\times$ to $7\times$ higher than the crossbars designed using the presented scheme. Also, the latencies incurred in the designs generated by our scheme are within acceptable bounds from the minimum possible latencies (of a full crossbar). Moreover, depending on the design objective, crossbar size-performance trade-offs can be explored in this approach by tuning the analysis parameters (such as the window size, overlap threshold, etc.), as explained in further subsections.

2.5.3 Comparisons of Heuristic Engine with the Exact Engine

In this subsection, we explore the quality of the solutions produced by the heuristic engine with respect to the exact ILP engine. As the exact engine takes several hours to compute solutions for designs with more than few hundred windows, we

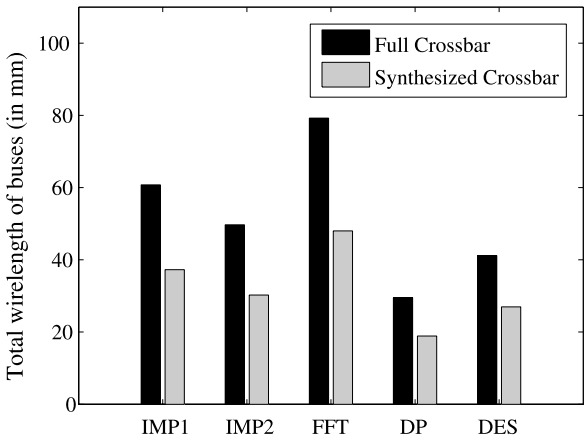


Fig. 2.11 Average wire-length of the crossbar buses for the designs

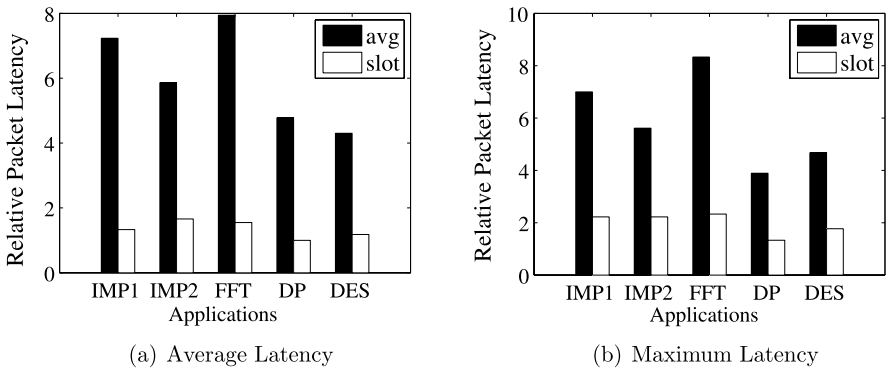


Fig. 2.12 Application relative latencies

reduced the number of windows to 100 for the designs and applied the two engines for the SoC designs. The size (total number of buses) of the crossbar synthesized by the heuristic engine normalized with respect to the size of the crossbar synthesized by the exact engine for the different designs is presented in Figure 2.10(a). The normalized power consumption of the synthesized crossbar designs for the different SoC designs is presented in Figure 2.10(b). Compared to the exact solutions, the solutions obtained by the heuristic engine incur only a modest increase in crossbar size (1.21× on average) and power consumption (1.26× on average).

The total run-time of the heuristic engine (including the time for performing floorplanning) for the biggest SoC design (the FFT SoC) for different number of window sizes is presented in Table 2.6. The experiments were performed on a Linux workstation, with 3.2 GHz processor, and 4 GB RAM. The run-time also includes the time to perform the sweep over the architectural parameters (frequency of operation and bus width) of the crossbar design. As seen from the table, the algorithms have a very low run-time complexity even for large designs and when large number

Table 2.6 Heuristic procedure run-time for FFT design

Number of windows	Run-time (in s)
1000	4.85
10000	5.46
50000	8.31
100000	11.91
500000	41.40

Table 2.7 Run-time for different number of cores

Number of cores	Run-time (in s)
29	41.40
40	67.22
50	96.73
60	130.03

of windows are used for analysis. On the other hand, the exact ILP procedure did not produce results in reasonable time for the design when more than few hundred windows were used for analysis. To show the scalability of the heuristic procedure with the number of cores in the design, we produced synthetic benchmarks based on the scaled versions of the FFT SoC. The execution times of the engine for the different benchmarks (with the number of windows set to 500,000) are presented in Table 2.7. From the table, we see that even for a very large design (60 cores with 500,000 windows used for analysis), the heuristic process completes in few minutes, thereby showing the scalability of the procedure.

2.5.4 Window Sizing

The size of the window used during the design process is an important parameter that determines the efficiency of the design methodology to capture the application performance parameters. A small window size results in much finer control of the application performance parameters and the resulting crossbars have lower latencies. However, a very small window size will lead to over-design of the network components. On the other hand, a large window size results in lesser control over the performance parameters of the application, but results in a more conservative design approach where higher transaction latencies can be tolerated.

To illustrate these effects, we applied the design methodology with different window sizes for a synthetic benchmark with 20 cores. Please note that we use a synthetic benchmark for this experiment (instead of the real SoC designs), so that we can vary the burst sizes (we refer to a burst as a stream of words generated by the

same core) in the application to study its impact on the crossbar synthesis process. The typical burst sizes for the benchmark is initially set to 100 cycles. When the window size is much smaller than the burst size, the size of the crossbar generated is very close to that of a full crossbar (refer Figure 2.13). When the window size is around few times that of the burst size (from 1–4 times), the synthesized crossbar has much smaller size (typically around 25%) and acceptable latencies (around 1.5×) of that of a full crossbar. For aggressive designs, the window size can be set closer to the burst size and for conservative designs (where larger transaction latencies can be tolerated), the window size can be set to few times the typical burst size. The acceptable window sizes for various burst sizes is presented in Figure 2.14. It can be seen from the plot that the window size varies almost linearly with the burst size, consolidating the above arguments.

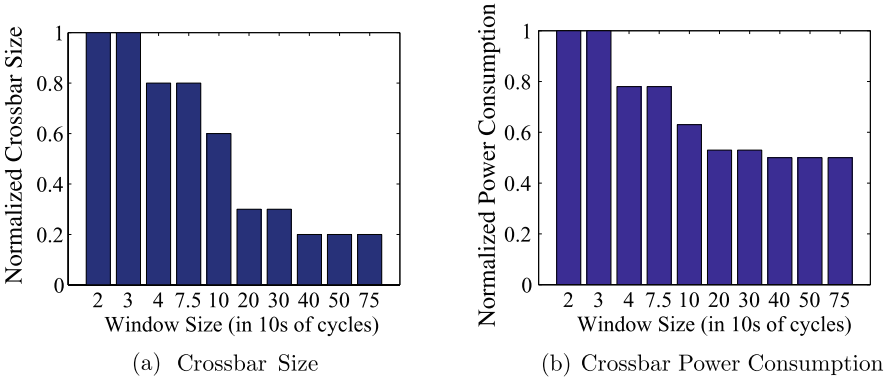


Fig. 2.13 Effect of window size on crossbar size and power consumption

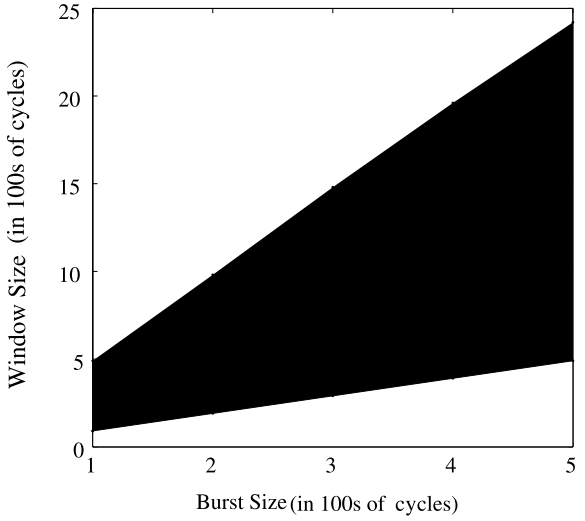


Fig. 2.14 Burst vs. window size

2.5.5 Real-Time Streams & Effect of Binding

In each simulation window, the critical traffic streams that require real-time guarantees are recorded. During the preprocessing step of the design flow (refer to Figure 2.4), the real-time traffic streams that overlap with each other in any window are identified. In order to provide real-time guarantees to such streams, the cores with critical streams that have temporal overlap are placed onto separate buses of the crossbar. Experimental results on the benchmark applications show a very low transaction latency (almost equal to the latency of perfect communication using a full crossbar) for such streams. Please note that in order to provide hard real-time guarantees, the underlying crossbar architecture should also provide support for having priorities for the different traffic streams, so that the real-time streams are given higher priorities over other streams. In many crossbar architectures, such as the ST-bus, such support is provided in the crossbar architecture by utilizing priority based arbitration mechanisms.

After finding the best crossbar configuration, we do an optimal binding of the cores onto the buses of the crossbar, minimizing the total overlap on each bus. By minimizing the overlap on each bus, the transaction latencies reduce significantly. To illustrate this effect, we compare the crossbars designed using the proposed approach with two binding schemes: random binding of cores onto the buses, satisfying the design constraints (equations (2.3)–(2.8)) and optimal binding that minimizes overlap on each bus, satisfying the design constraints. The average latency incurred by the random binding scheme for the benchmark applications was on average 2.1× higher than that incurred by the optimal binding scheme.

2.5.6 Overlap Threshold Setting

By varying the two parameters: window size and overlap threshold, the crossbar can be designed such that the average and the maximum transaction latencies incurred in the design are acceptable. The effect of the overlap threshold parameter on the size and power consumption of the crossbar generated for the synthetic benchmark are presented in Figures 2.15(a) and (b). The crossbar size and power numbers are normalized with respect to the case when the overlap threshold is set to 0%, which leads to a full crossbar configuration (as no two cores can share a bus in this case). The plots end at 50% overlap between cores because, if the pair-wise overlap between two cores exceeds 50% of the window size (in any of the windows), then the window bandwidth constraints cannot be satisfied. So, the maximum value of the overlap parameter can be set at 50% of the window size. This will also speed-up the process of finding the best crossbar configuration, as such overlapping cores will be identified in the preprocessing phase (refer to Figure 2.4) and will be forbidden to be on the same bus of the crossbar. From experiments, we found that for aggressive designs (where there are tight requirements on the maximum latencies) the threshold can be set to around 10% and for conservative designs, the threshold can be set to 30%–40% of the window size.

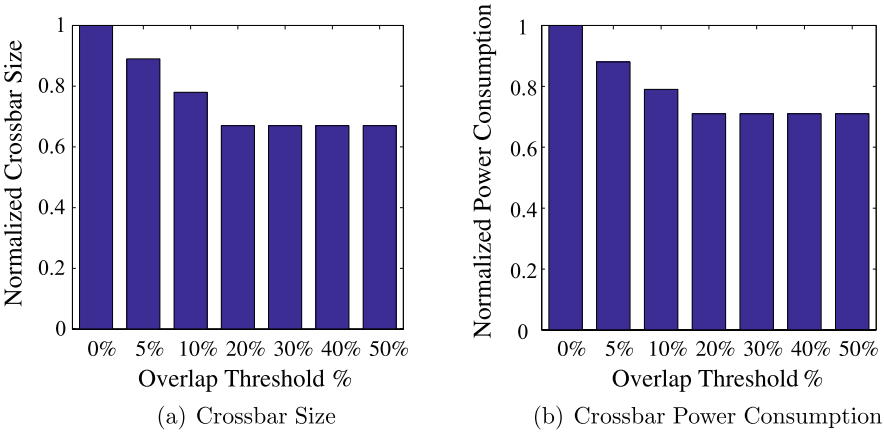


Fig. 2.15 Effect of overlap threshold parameter

2.6 Summary

Today, a streamlined methodology to design crossbar based architectures is not yet fully developed. Toward this end, in this chapter, we have presented methods that address this important problem of designing optimal crossbar based systems for SoCs. The approaches consider the real application traffic, accounting for the local variations and temporal overlap of the traffic streams. As a crossbar system is wiring dominated, it is important to consider the wiring complexity during the design of the architecture. We consider this by utilizing physical design aware methods that consider the layout of the design. We have validated the methods using two state-of-the-art industrial platforms: STBus and AMBA AXI, that are widely deployed in several industrial designs. Equipped with this knowledge of designing bus-based systems, in the subsequent chapters, we will proceed to design general NoC systems that can handle the global traffic requirements of SoCs.

Chapter 3
Netchip Tool Flow for NoC Design

The crossbar based systems we considered in the previous chapter can provide a very high bandwidth communication infrastructure. However, they are still inherently nonscalable, as all the cores need to connect to a single crossbar matrix. To provide a scalable infrastructure, we need to utilize many such crossbar matrices in the design. NoCs can be viewed as a logical extension of this concept, where multiple switches are used to connect the cores of the SoC. The switches, while providing the functionality of a crossbar matrix, also support decentralized control of the traffic flows.

A NoC consists of three main components: *switches*, *Network Interfaces* (*NIs*), and *links*. The NoC is instantiated by deploying a set of these components in an arbitrary topology and by configuring them.

In this chapter,[1] we present *NetChip*, a design flow [33] that automates most of the complex and time-intensive design steps in NoC synthesis. It provides design support for application-specific standard and custom network topologies and, therefore, lends itself to the implementation of both homogeneous and heterogeneous system interconnects. Netchip assumes that the application has already been mapped onto cores by using preexisting tools and the resulting cores together with their communication requirements are taken as an input. The tool-assisted design and generation of a customized NoC-based communication architecture is the ultimate goal of Netchip.

The design flow of Netchip is presented in Figure 3.1. The Netchip tool flow has 3 main phases and several tools integrated together:

- Front-End Design Phase: In this phase, several key NoC features such as the interconnect structure (or topology), routing scheme, paths for traffic flow, values for the NoC architectural parameters are determined.
- Architectural Design Phase: In this phase, the RTL code of the NoC architecture is instantiated.
- Back-End Phase: In this phase, simulation, FPGA emulation, and layout generation of the NoC are carried out.

3.1 Front-End Design Phase

We have developed two tools: SUNMAP [11, 45, 46], and SUNFLOOR [82, 83] to design application-specific standard and custom topologies, thereby automating this phase. Based on the user's choice, either a custom topology is synthesized using *SUNFLOOR* or mapping onto a regular topology is performed using *SUNMAP*.

[1] We would like to acknowledge Dr. Federico Angiolini for his contributions.

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

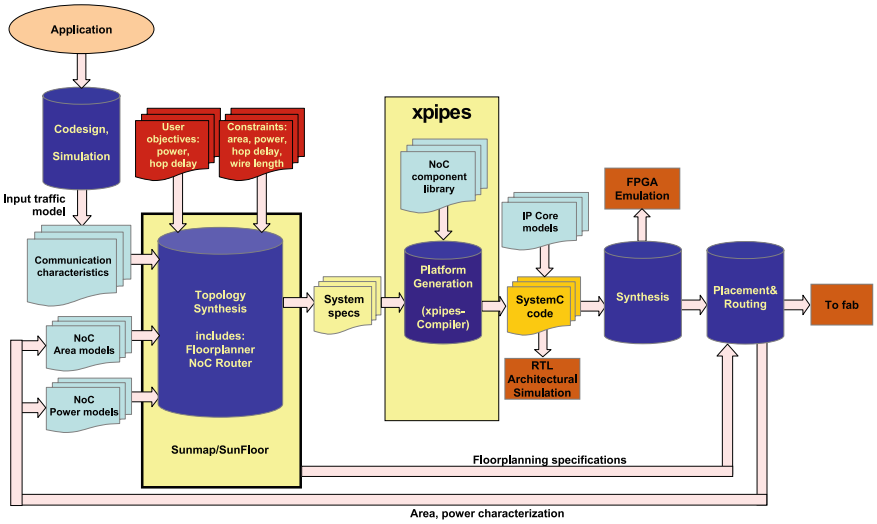


Fig. 3.1 Design flow of Netchip

SUNMAP maps the input core graph onto various standard topologies (*mesh*, *torus*, *hypercube*, *Clos,* and *butterfly*) defined in the topology library. This tool is further explained in Chapter 4.

The SUNFLOOR tool is used to synthesize a custom irregular topology that is tailor-made for a specific application. This tool is further explained in Chapter 5.

3.2 Architectural Design Phase: The ×pipes NoC Library

Among the many NoC architectures proposed in the literature, we choose the ×*pipes* NoC architecture, which incorporates features that have been successful in many NoC designs and represents a reasonable design point. The ×pipes NoC [34, 88] is an example of a highly flexible library of component blocks (Figure 3.2).

To show the generality of the methods, we also apply them to the Æthereal architecture [30] (which is presented later in this chapter). We chose this architecture, as it represents a different design point in the NoC spectrum. It supports a predictable communication behavior, by providing connections with throughput guarantees. This is in contrast with the ×pipes architecture, where the packets are routed in a best-effort manner.

The backbone of the NoC consists of switches, whose main function is to route packets from sources to destinations. Arbitrary switch connectivity is possible, allowing for implementation of any topology. Switches provide buffering resources to lower congestion and improve performance; in ×pipes, output buffering is chosen, i.e., FIFOs are present on each output port. Switches also handle flow control [90] issues (we use the ACK/NACK protocol in this thesis), and resolve conflicts among packets when they overlap in requesting access to the same physical links.

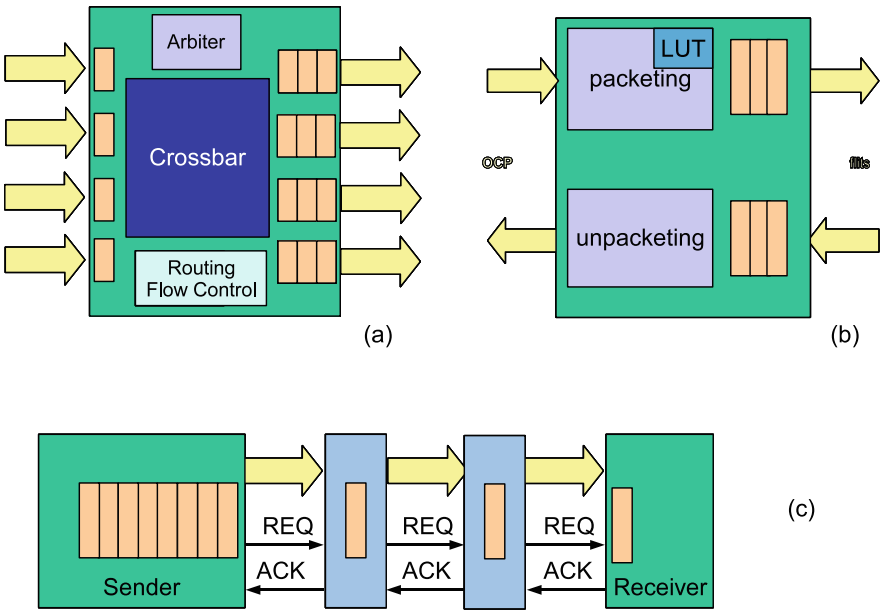


Fig. 3.2 ×pipes building blocks: (**a**) switch, (**b**) NI, (**c**) link

An NI is needed to connect each IP core to the NoC. NIs convert transaction requests/responses into packets and vice versa. Packets are then split into a sequence of *flits* (*FLow control unITS*) before transmission, to decrease the physical wire parallelism requirements. In ×pipes, two separate NIs are defined, an *initiator* and a *target* one, respectively, associated to system masters and system slaves. A master/slave device will require an NI of each type to be attached to it. The interface among IP cores and NIs is point-to-point as defined by the *Open Core Protocol OCP 2.0* [91] specification, guaranteeing maximum reusability. NI *Look-Up Tables* (*LUTs*) specify the path that packets will follow in the network to reach their destination (source routing). Two different clock signals can be attached to NIs: one to drive the NI front-end (OCP interface), the other to drive the NI back-end (×pipes interface). The ×pipes clock frequency must be an integer multiple of the OCP one. This arrangement allows the NoC to run at a fast clock even though some or all of the attached IP cores are slower, which is crucial to keep transaction latency low. Since each IP core can run at a different divider of the ×pipes frequency, mixed-clock platforms are possible.

In the *topology generation* phase, Netchip reads the topology and routing information file and generates SystemC description of network components for the topology using ×pipesCompiler. The ×pipesCompiler instantiates a network of building blocks from the ×pipes library.

Once the SystemC code is available, it can be used in multiple ways. To get accurate simulation in a flexible environment, we integrate the NoC in the MPARM simulation platform [33]. MPARM allows for accurate injection of functional traffic

patterns as generated by real IP cores (processors, DMA engines, etc.) during a benchmark run. Further, it provides facilities for debugging, statistics collection, and tracing.

The RTL code of the platform can also be used to synthesize it, either on FPGA or on a custom chip via the use of technology libraries.

After synthesis phase, a place&route of the design can be performed using standard tools, such as Synopsys Astro or Cadence SoC encounter. The placement tool is fed with a floorplan specification file automatically generated by SUNFLOOR. This file contains information about the layout *fences*, i.e., sets of constraints on where the cells of each NoC module and the black boxes representing the IP cores can be placed. This approach lets the designer skip, if desired, the tedious activity of manually placing blocks on the floorplan, and iteratively improving the result by means of trial-and-error tighter packing. The tool automatically places the cells within the fences, and subsequently performs the wire routing steps. The final output is a complete layout of the NoC design that can be sent to a foundry.

3.3 Summary

In this chapter, we presented the basics of the Netchip design flow that automates the design of application-specific NoCs. The design flow integrates the front-end design phase, where the NoC topology is synthesized, with the architectural and back-end phases. The front-end phase has two major tools: SUNMAP and SUNFLOOR to design regular and custom topologies. In the subsequent chapters, we will present a detailed description of these tools.

Chapter 4
Designing Standard Topologies

The NoC topology defines the interconnection of the different network switches with the cores and among each other. The NoC topologies can be broadly classified into two main categories: standard and application-specific custom topologies. In the standard topologies, the interconnection structure ensures full connectivity between the cores: that is, any core is reachable from any other core. Examples of such topologies include mesh, torus, hypercube, Clos, and butterfly. In an application-specific custom topology, the interconnection between the switches and cores are optimized to match the application traffic patterns. If an application does not require full connectivity between the cores, then the topology is optimized to provide only the required connectivity.

The use of a custom topology for an application almost always leads to a better performance and reduction in area/power overhead. However, there are some situations where a standard topology is desirable for the design:

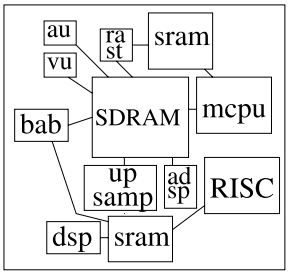
- When the NoC is to be used across multiple product generations, a standard topology ensures that the same NoC can be reused easily across the different generations. However, when using a custom topology, the designer has restricted options when adding cores in the future, as the NoC may not provide full connectivity.
- When the cores are almost regular (similar sizes), the use of a standard topology leads to better wiring structure, as the floorplan is more predictable.

In this chapter, we present *SUNMAP*, a tool for synthesizing the best standard topology for applications.

SoCs are aggressively designed to meet the performance requirements of diverse applications that need to be supported. In most cases, the cores in the SoC are heterogeneous in nature with each core performing a set of specialized functions in order to maximize performance and satisfy design constraints such as *Quality-of-Service* (*QoS*) for the applications. As an example, consider an efficient design of an *MPEG4 decoder* shown in Figure 4.1(a) [8]. In this design, there are several processors (e.g., `RISC`), several hardware cores (e.g., `Upsampler`), and memory cores (e.g., `SDRAM`). Each core has different functionality, size, and communication requirements. Some of the cores are *hard cores*, with size fixed during design (e.g., `RISC`) and some of the cores are *soft cores*, whose size can be varied with some restrictions on the aspect ratios (e.g., `Upsampler`).

Figure 4.1(b) shows the design area for the best mappings of the *MPEG4* onto a mesh topology for two schemes: in the first scheme, the mapping of the cores is done logically (without considering the physical planning of the cores) followed by a separate physical planning phase, and in the second scheme the mapping and physical planning are done together, so that the mapping process takes the physical planning information, i.e., the position of the cores and network components (e.g., switches,

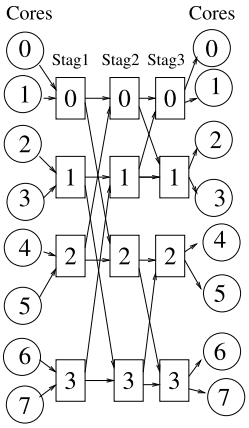
S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,



(a) MPEG4

Scheme	Area (sq mm)
1. Mapping Then Physical Planning	36.0
2. Integrated Mapping & Physical Planning	20.25

(b) Mapping Schemes



(c) Butterfly Network

Fig. 4.1 MPEG4 mapping schemes and an example of butterfly topology

links) and the size of soft cores and switches in the 2D plane. There is significant area improvement in the second scheme where mapping and physical planning are integrated together.

This improvement will be even more pronounced for indirect topologies such as the *butterfly* network shown in Figure 4.1(c). In a butterfly topology, logically, the switches are arranged as stages with the switches in the first and last stages connected to the cores. Ideally, we would like to distribute the switches around the cores so that performance of the NoC is maximized and mappings onto the butterfly should take this physical planning information into account.

Another important design consideration for SoCs is to guarantee *Quality-of-Service* (*QoS*) for the application. As an example, in many video applications, data should be communicated in such a way that the system supports a predetermined frame rate (e.g., 30 frames/s in many video displays). The network should support the QoS requirements of the applications satisfying the delay constraints of the traf-

fic streams. It should also provide support for real-time communication. These QoS guarantees need to be considered during the mapping process. Moreover, the burstiness in the traffic streams (that makes providing QoS guarantees harder) needs to be considered.

In SUNMAP, we provide an integrated approach to mapping and physical planning, where we determine the 2D position of the cores and network components and the size of soft cores and switches during the mapping process. The physical planning phase also automatically computes the switch buffers needed to support the application traffic and integrates this in the switch size computation. We also present a method to provide QoS guarantees for the application during the mapping-physical planning phase. For QoS guarantees, we consider the burstiness in the application traffic, delay/jitter constraints of the individual traffic streams, and provide support for real-time communication. The additional power-area overhead in obtaining the QoS guarantees is negligible. The mapping and physical planning of the cores is applied to several topologies defined in a topology library and the best topology for the application is automatically selected. In the resulting topology, the switches and links are optimized for the traffic characteristics, followed by automatic instantiation of the topology. Thus, the integrated design methodology automates *mapping, physical planning*, and *topology selection* for an application providing *QoS guarantees*, thereby bridging an important design gap in building NoCs based on standard topologies.

4.1 On-Chip Traffic Modeling

In this section, we develop traffic models to characterize the application traffic, providing QoS guarantees for the application. As an example, consider the traffic flowing between the *Filter core* and the *ARM core* in a *DSP Filter* application (refer to Figure 4.2). Without loss of generality, assume that the packet size is such that a packet is sent in one cycle, although the following discussion also applies when a packet is sent over multiple cycles (i.e., when a packet has multiple flits). There are three important features to be noted from Figure 4.2(b).

- *Bursty Traffic Flows*: The application traffic from *Filter* to *ARM* core is bursty in nature, with a burst period of 100 cycles followed by 900-cycles of silence period. The peak-bandwidth of the traffic (100 packets/100 cycles) is an order of magnitude higher than the average bandwidth (100 packets/1000 cycles).
- *Delay/Jitter Constraints*: Each burst from the *Filter* core has a delay constraint by which it should reach the *ARM* core. In this example, we assume that the burst B1 has to reach the *ARM* core by 500 cycles, which is obtained from the application characteristics.
- *Real Time Constraints*: The *ARM* core issues a control stream to the *Filter* which is assumed to be critical and needs to reach the *Filter* as quickly as possible. These real-time requirements need to be satisfied by the network.

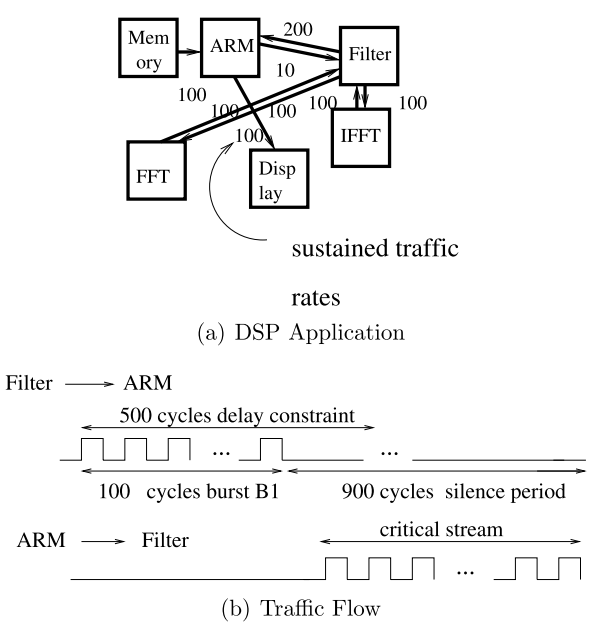


Fig. 4.2 *DSP Filter* application and traffic flow between *ARM* & *Filter* cores

Table 4.1 Link implementation

Scheme	BW (pk/cy)	Delay (cyles)
1. Avg.	100	1000
2. Peak	1000	100
3. Opt.	200	500

Consider three implementations of the communication link (refer to Table 4.1) between the *Filter* core and *ARM* core (for illustrative purposes assume other cores do not send traffic on this link). In the first case, the link is designed to support the average bandwidth of traffic flowing between *Filter* and *ARM*. As seen from Table 4.1, the delay incurred in this scheme for the burst B1 violates the delay constraint for the stream. In the second case, the link is designed for the peak bandwidth requirements and the delay constraints are met. However, the link is over-designed with 5× the capacity that is needed to support the delay constraints of the burst. In the third case, the link is optimally designed to support the burst without violating the delay constraints.

From this example, it is clear that the communication links should be designed optimally in a way such that they support the traffic flowing through them, satisfying the delay/jitter constraints of the traffic streams. Moreover, there should be a mechanism that ensures that each core sends traffic so that the links can support the traffic and the delay constraints are met. Clearly, these two objectives complement each other and to ensure that the objectives are met we propose the use of traffic reg-

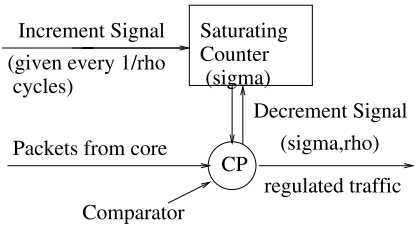


Fig. 4.3 A (σ, ρ) regulator

ulators for NoCs. Traffic regulators are widely used in *ATM* networks to guarantee QoS to applications [94]. A traffic regulator can be abstracted as a hardware block with two parameters: σ and ρ. The parameter ρ represents the bandwidth required to support the traffic streams so that the delay constraints are met and the parameter σ represents the variations permitted over the ρ value. Such a regulator is also called as a (σ, ρ) regulator [94]. The traffic flow between each source-destination is represented by a (σ, ρ) value. As an example, the *Filter* to *ARM* communication is represented by (0, 0.2), which means that one packet can be sent every 5 cycles (i.e., one packet can be sent every $1/\rho = 1/0.2 = 5$ cycles) and no variations over the required rate is permitted (as the σ value is 0). A (1, 0.2) regulator would allow a burst of one packet over the required packet rate. In the rest of this chapter, we assume that the σ value is chosen to be equal to 0, so that no variation is permitted over the required rate.

To ensure that each core sends data according to the regulator values, we need to add small hardware to each core (or to the Network Interface connecting the core to the network), which is shown in Figure 4.3. The additional hardware consists of a saturating credit counter and a comparator. The saturating counter is incremented at rate ρ and saturates when it reaches a count of $(1 + \sigma)$. A packet is transmitted only if the credit counter is non-zero and when a packet is transmitted the counter is decremented by 1. This counter ensures that the amount of traffic transmitted by the source matches the rate for which the links are designed to handle. For traffic streams to different destinations, different sets of (σ, ρ) values are used in the regulator. Note that power-area overhead of such a regulator is negligible as it is just a counter and a comparator. For supporting real-time constraints, we assume tight latency bounds for the real-time stream and during the mapping process we consider the criticality of the stream by using a weighted communication graph (called as *weighted core graph*), where the weights are a function of the criticality. In the next section, we explain this in more detail, where we present mathematical models for modeling the ρ value for the regulators.

4.2 Problem Formulation

The communication between the cores of the SoC is represented by the *weighted core graph*:

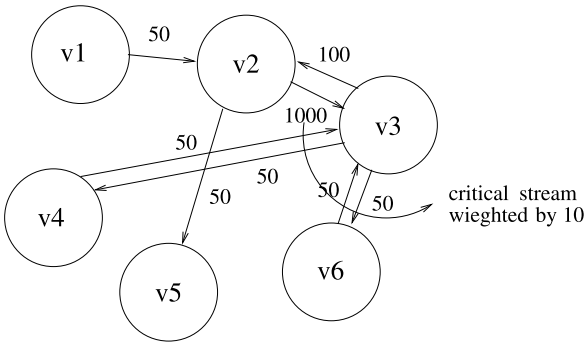


Fig. 4.4 Weighted core graph

Definition 5 The weighted core graph is a directed graph, $G(V, E)$ with each vertex $v_i \in V$ representing a core and the directed edge (v_i, v_j), denoted as $e_{i,j} \in E$, representing the communication between the cores v_i and v_j. The weight of the edge $e_{i,j}$, denoted by $comm_{i,j}$, represents the average bandwidth of the communication from v_i to v_j weighted by the criticality of the communication.

As an example, the weighted core graph of the *Filter* application is given in Figure 4.4. The edge weights are a function of the criticality of the stream (which depends on the application characteristics) and the amount of traffic communicated in the stream. The value of the weights depends on how critical are the streams and on the number of classes of streams. In this work, we assume two classes of streams: noncritical and critical and weigh the critical streams by a factor of 10 compared to noncritical streams. Other approaches such as weighing a stream based on the amount of slack permitted for the stream can also be used.

Definition 6 In $G(V, E)$, the traffic flow from each source v_i to each destination v_j, $\forall i, j \in V$ is represented by the set $T_{i,j}$. Each $T_{i,j}$ comprises of $M_{i,j}$ bursts, with each burst $b_{i,j,k}, \forall k \in M_{i,j}$, having a burst length of $blen_{i,j,k}$ cycles and a latency window of $blat_{i,j,k}$ cycles.

In the above example, the traffic flow between the *Filter* and the *ARM* (v_3 and v_2) is represented by the set $T_{3,2}$. The set $T_{3,2}$ consists of 1 burst (we assume such a small sampling window for illustrative purposes), with $M_{3,2}$ equal to 1, $blen_{3,2,1}$ equal to 100 cycles and $blat_{3,2,1}$ equal to 400 cycles. The latency window, $blat_{i,j,k}$ is the deadline (or slack) that is permissible for the burst, which is obtained from the initial simulation of the application and the application characteristics.

The $\rho_{i,j}$ values of the regulator for each source v_i to destination v_j is obtained by

$$\rho_{i,j} = \max_{\forall k \in M_{i,j}} \left(\frac{blen_{i,j,k}}{blen_{i,j,k} + blat_{i,j,k}} \right) \quad \forall i, j \quad \text{s.t. } e_{i,j} \in |E| \tag{4.1}$$

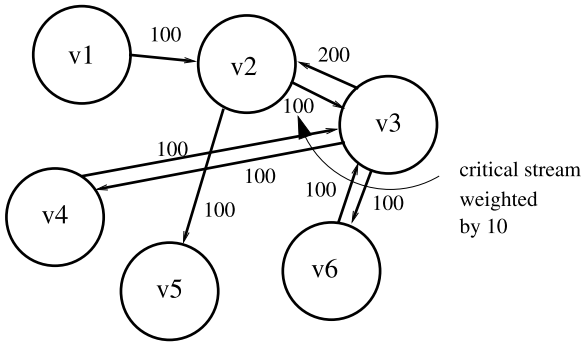


Fig. 4.5 Constraint Graph:BW in MB/S

Definition 7 The bandwidth constraint graph $CG(Q, R)$ is a directed graph where the vertex and edge sets are equal to the vertex and edge sets of $G(V, E)$ but with edge weights $r_{i,j}$ equal to $\rho_{i,j} \times PacketSize/Cycletime$, $\forall i, j \in$ s.t. $r_{i,j} \in |R|$.

The bandwidth constraint graph for the above example is given in Figure 4.5. The edge weights in the graph are $\rho \times packetsize/Cycletime$ values for the corresponding traffic flows. When calculating the ρ values, we neglect the network latency as it is of the order of tens of cycles, while burst lengths and latency windows are of the order of hundreds of cycles.

The NoC topology is defined by the adjacency information of nodes in the topology and by the capacity of the links. Formally, the NoC topology is defined as follows.

Definition 8 The NoC topology graph is a directed graph $P(U, F)$ with each vertex $u_i \in U$ representing a node in the topology and the directed edge (u_i, u_j), denoted as $f_{i,j} \in F$ representing a direct communication between the vertices u_i and u_j. The weight of the edge $f_{i,j}$, denoted by $bw_{i,j}$, represents the bandwidth available across the edge $f_{i,j}$.

The mapping of the application cores onto an NoC architecture is defined by the one-to-one mapping function:

$$map : V \rightarrow U, \quad \text{s.t. } map(v_i) = u_j, \quad \forall v_i \in V, \ \exists u_j \in U \tag{4.2}$$

Each link in the mapped NoC should satisfy the bandwidth constraints corresponding to the constraint graph $CG(Q, R)$. The design objective (area, power, or hop delay) of a mapping is obtained from the physical planning of the mapping. This ensures that the heterogeneity in the size of the cores and network components is taken into account for accurate estimation of the design objectives.

4.3 Mapping and Physical Planning Algorithm

In this section, we present the algorithm for mapping and physical planning. The general problem of embedding one graph into another is intractable and is a special case of the *Quadratic Assignment Problem* (*QAP*) [64]. *QAP* is well studied in the literature with many heuristic algorithms available [29]. In [29], *robust tabu search* is shown to be most effective for many classes of QAP and we use this to solve the mapping problem. The general structure of the mapping-physical planning algorithm is shown in Figure 4.6.

In the first step, an initial greedy mapping of the cores onto the topology is obtained. We also assume a greedy mapping of higher dimensional topologies (such as hypercube) onto the 2D plane. Then for each iteration of the *robust tabu search*, we perform the following computations:

- Compute the routes for the traffic flowing between the cores, based upon the routing function chosen from the library.
- Physical planning for this mapping. This includes computing the positions of the cores and the switches, sizes of the switches and soft cores, and automatic computation of switch buffers needed for the application. These steps are explained in detail in the next section.
- Check whether the mapping satisfies the delay/jitter and area constraints. For delay constraints, the links in the NoC should support the traffic through them, which is determined by the (σ, ρ) regulator values. We also check whether the real-time constraints for the critical streams are met by checking whether the hop delay for the streams are lower than the required value, which is obtained from the application characteristics.

```
Mapping_and_physicalplanning(G,CG,P)
{ obtain an initial greedy mapping of G onto P;
  obtain greedy mappings of higher dimensional
  topologies onto 2D plane;
  In each iteration of the robust tabu search  perform
  {  for all i and j in U of topology graph
     {  generate current move by swapping the cores
        in positions i and j ;
        if current move is not tabu
        {  compute routes  based on routing function;
           Simultaneously perform following steps
           in physical planning using MILP:
              Opimize design area, power or hop delay;
              Compute the core and switch sizes and positions;
              Compute number of buffers;
           Check whether QoS constraints are satisfied;
           Update tabu list and check whether aspiration
           condition is met;
           if current iteration's cost is better than best cost
           update the best cost and best solution;
        }
     }
  }
```

Fig. 4.6 Mapping and physical planning algorithm

In each step of the tabu search, we try to optimize the design objective (area, power, or hop delay) satisfying the QoS and criticality constraints. The area and power values are obtained from physical planning of that particular mapping. The parameters of the tabu search (such as the *size of tabu list*, *aspiration function computation*, etc.) are chosen as explained in [29]. This tabu search is applied to all topologies in the library. The library currently has *mesh, torus, hypercube, Clos, and butterfly* topologies, while other topologies can be easily added to the library. The best topology is selected and the switches and links are optimized to match the application characteristics. In this step, redundant switch ports and links (i.e., the links that do not carry any traffic and the corresponding switch ports) are eliminated. The links are sized (by changing the bit-width of the links or frequency of operation) according to the traffic flowing through them.

4.4 Physical Planning

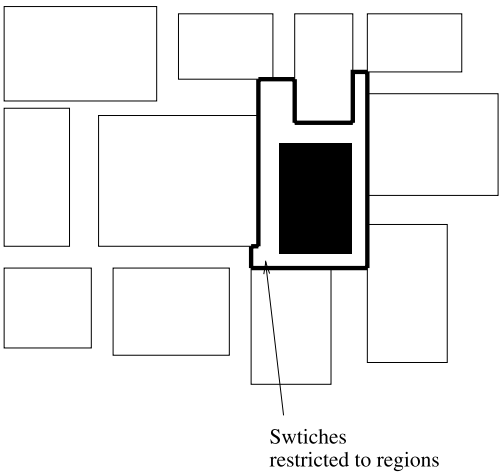
We use a *Mixed Integer Linear Program* (*MILP*) based physical planning algorithm. An MILP based physical planning for minimizing area, power of a design is presented in [63]. We modify this approach for NoCs by considering NoC specific features such as switch positioning, switch buffer calculation, etc.

As the cores are predesigned components, we assume the area and power values of the cores as an input. We also assume the type of the core (hard or soft) and aspect ratio constraints as an input. We use area, power libraries for various configuration of switches that are developed in [62].

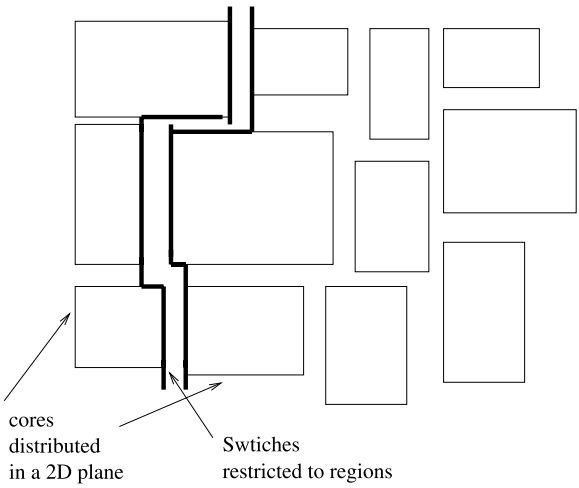
For a given mapping, the relative position of the cores with respect to each other is obtained from the tabu search, but the relative position of the switches is unknown. The switches in a direct topology (such as mesh, torus, hypercube) can be placed anywhere around the core to which it is connected. An important constraint to be considered in the MILP is that the switches and the cores should not overlap each other. If the switch positions are not restricted to a small region around the core, solving this overlap calculation as an MILP will be time consuming for large problem sizes (for >20 cores). To allow scaling of the algorithm, we restrict each switch to lie in a region of adjacent cores surrounding the core to which it is connected (refer Figure 4.7(a)). By restricting the switch positions to a small region, the overlap calculations are several orders of magnitude faster and are scalable for large problem sizes. The solution obtained in this scheme, for all the simulations performed, are within 1% from the solution obtained without restricting switch sizes as the switch position tends to be close to the core to which it's connected.

For the indirect topologies (such as the Clos and butterfly), we distribute the switches along the cores in a 2D plane, based on their connectivity to the cores and to other switches (refer Figure 4.7(b)). Here again, we restrict switch locations to lie within certain regions as shown in the figure. Then during each step of the tabu search, we compute the actual positions of the switches and cores.

During the physical planning, we also compute the buffering needed at each switch. We assume that the links are pipelined with the number of pipeline stages



(a) Direct topologies



(b) Indirect topologies

Fig. 4.7 Switch Position Restriction for direct and indirect topologies

depending upon the link length. For wormhole (or virtual channel) based switches with credit based flow control, for maximum throughput, the number of buffers in the switches should be equal to $2N + M$, where N is the number of pipeline stages in the link and M is the delay incurred for credit processing at the upstream and downstream switches [94]. As the switch size (power) depends on the number of buffers, we integrate this as a constraint in the MILP by breaking down the switch area (power) as a sum of buffer area (power) and crossbar (including logic) area (power). The buffer area (power) is a function of link length and is automatically calculated during physical planning.

4.5 Experiments and Case Studies

4.5.1 Effect of Physical Planning

In this subsection, we investigate the effect of combined mapping and physical planning applied to a variety of video applications. We consider four different video applications: *Video Object Plane Decoder* (VOPD-12 cores), *MPEG4 decoder* (mapped onto 12 cores), *Picture-In-Picture* application (PIP-8 cores), *Multi-Window Application* (MWA-14 cores). We assume that the design objective is to minimize design area subject to delay/jitter and criticality constraints. We consider two schemes: in the first scheme, the mapping and physical planning phases are done separately (as in past works), and in the second scheme we use the proposed integrated approach to mapping and physical planning.

The design area for the video applications as obtained for both the schemes are presented in Table 4.2. On an average, we have 1.4× area savings in the proposed approach.

4.5.2 Design for QoS Guarantees

In traditional design methodology, QoS can be guaranteed by designing the network to support the worst-case bandwidth needs of the application. Such a worst-case design approach, however, leads to an over-design of the network components. By using (σ, ρ) traffic regulation methodology for NoCs presented in this chapter, the network components are designed optimally to support the QoS constraints of the application.

As an example, for the *DSP Filter* application (Figure 4.2), the minimum bandwidth needed (assuming minimum-path routing) for the design methodology is 5× lower than a worst-case design approach. Moreover, in the proposed design methodology, the network is made to operate at very low contention, thereby reducing contention delay and power. Figure 4.8 shows the packet latency as obtained from the actual simulations of the *DSP Filter* application. In the first case, the links are designed to handle the average traffic through them. As the traffic is bursty in nature,

Table 4.2 Design area for video applications

Appln	Area-1 sqr mm	Area-2 sqr mm	Ratio
VOPD	20.25	18.01	1.12
MPEG	36.00	20.25	1.19
PIP	20.25	10.565	1.92
MWA	33.00	25.00	1.32
Avg.	–	–	1.39

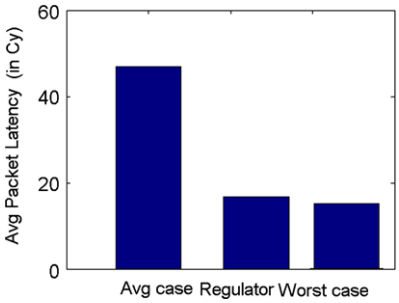


Fig. 4.8 Avg. latency for *DSP*

such a design approach leads to high network contention resulting in large packet latency. In the second case, the links are designed with the design methodology. The average latency is almost equal to the worst-case design approach (case 3) where the network components are over designed. As the design methodology for traffic regulators is based on initial simulation, it is static in nature and does not capture dynamic variations in the input data streams. But for many SoC applications, the traffic characteristics do not vary a lot with the input data [43]. Thus, the design methodology incurs only slight increase in latency (around 10%) due to dynamic changes in data when compared to the worst-case design approach.

4.5.3 VOPD Design

In this subsection, we explore VOPD mapping and physical-planning with QoS guarantees. We assume a conservative link bandwidth of 2 GB/S.

The bandwidth constraint graph for the VOPD application, based on the traffic characteristics and QoS needs of the application is presented in Figure 4.9. For minimum-path mapping, the minimum bandwidth needed to support the application is 2.4 GB/S and cannot be supported by any of the topologies. So, we apply split-traffic routing, spreading the traffic between the cores across multiple paths. As a butterfly network has no path diversity (only one path from any source to any destination) [94], it cannot support the traffic requirements of the application. All other topologies produce feasible mappings with split-traffic routing. We assume that the objective is to minimize power consumption of the design, satisfying QoS, and area constraints. Figure 4.10 shows the power consumption of the topologies. Mesh has the least power and is the best topology for VOPD for the chosen design objective.

4.5.4 Buffer Sizing and Network Optimization

During physical planning, the number of buffers needed for the switches is automatically computed based on the link lengths and this is integrated into the area (power) calculations of the physical planner. When the number of buffers is lower

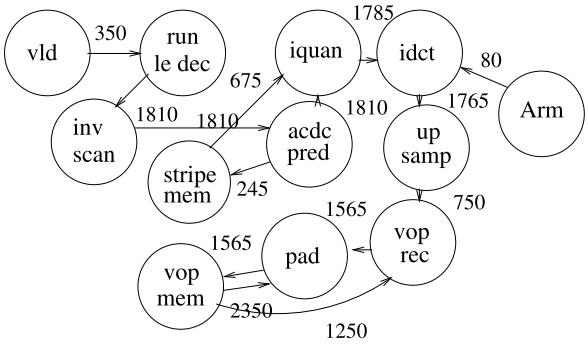


Fig. 4.9 Bandwidth constraint graph for VOPD with bandwidth in MB/S

Fig. 4.10 VOPD design

Topol.	Power (in mW)
Bfly	No mapping
Mesh	542
Torus	930
Hyp.	960
Clos	753

than the required number, throughput of the network is low. On the other hand, when the number of buffers is more than needed, the throughput remains the same, but switch area and power are increased. As an example, let us consider a homogeneous 16-node torus NoC in which each link has 4 pipeline stages. Let us assume that the credit processing delay (the M value) is 2 cycles, which is typical for most credit-based switches. Figure 4.11 shows the throughput dependence on the total number of buffers in the switches for the NoC. As seen, the relative throughput increases until the optimal count of 702 buffers, after which it remains constant. With the buffer computation methodology, the physical planner automatically computes this optimum number of buffers needed to support maximum throughput. Note that in a heterogeneous SoC, the number of buffers can be different for different switches and even different for different inputs of the same switch as the link lengths are nonuniform in nature. Even in this case, the physical planner automatically computes the optimum number of buffers needed at each input of the switch based on the corresponding link lengths. For the VOPD application, compared to an average-case design (where all the switches have the same number of buffers) we get 2.2× reduction in buffer count in this scheme.

After the topology selection phase, the network components (switches and links) are optimized based on the traffic flowing through them. The links and switch ports that do not carry any traffic are removed. Other links and switches are optimized to

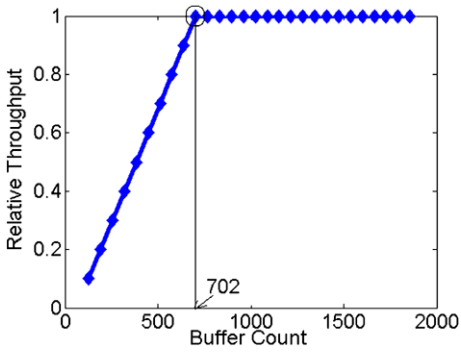


Fig. 4.11 Throughput vs. buffer count

Table 4.3 Network optimization

Component	Savings
Buffers	2.20×
Wire count	3.77×
Ports	1.60×

match the traffic rate through them by changing the bit-width of the links. The effect of network optimization on VOPD design is reported in Table 4.3.

For all the experiments, the mapping and physical planning phases are executed in few minutes on a 1 GHz SUN workstation and the algorithms are scalable for hundreds of cores.

4.6 Summary

A variety of applications require a regular interconnect structure. In such cases, selecting the most suitable topology for the application, mapping of cores onto that topology and generating the resulting network are important phases in designing the NoC. In this chapter, we have presented SUNMAP, a tool that automates all these steps, bridging an important design gap in building regular NoCs. It explores various design objectives such as minimizing average communication delay, area, power dissipation subject to bandwidth, and area constraints. The tool supports different routing functions (dimension ordered, minimum-path, traffic splitting) and uses floorplanning information early in the topology selection process to provide feasible mappings. Experiments on several realistic MPSoC applications show that it can also be used as a powerful design space exploration tool. In the next chapter, we present the SUNFLOOR tool that automates the design of custom (irregular) NoC topologies for applications.

Chapter 5
Designing Custom Topologies

In this chapter,[1] we present *SUNFLOOR*, a tool for synthesizing the best custom (irregular) topology that is tailor-made for a specific application and satisfies the communication constraints of the design. The tool automates the entire NoC front-end design process, including topology synthesis, routing, path computation, architectural parameter setting: thereby bridging an important gap in the design of the communication architecture for application-specific MPSoCs.

5.1 Objectives

The SUNFLOOR tool has several salient features:

1. The synthesis method is both performance and power consumption aware, which are two of the important design objectives in MPSoC design. It supports two objective functions: minimizing network power consumption and hop-count for data transfer. The designer can optimize for one of the two objectives or a linear combination of both. The topology design process supports constraints on several parameters such as the hop-count (when the objective is power minimization), network power consumption (when the objective is hop-count minimization), design area, and total wire-length.
2. SUNFLOOR incorporates mechanisms to guarantee the generation of networks that are free from deadlocks, which is critical for the deployment of custom NoC topologies in real designs. The deadlock-freedom is achieved without the use of special hardware mechanisms (refer to Section 5.1.2 for details).
3. The tool uses a floorplan-aware topology design method. It considers the wiring complexity of the design for accurate timing and power consumption estimation.
4. Accurate analytical models for the area, power, and timing information of the network components (switches and links) were built from layout level implementations, which are utilized during the synthesis process. The power values are obtained from layouts of the network components with back-annotated resistance and capacitance information (at 0.13 μm technology), based on the switching activity of the components. The area and power models are highly detailed: they even capture the impact of the frequency used for RTL synthesis on the area and power values of the components. This is further explained in detail in Section 5.2.

[1]We would like to acknowledge the contributions of Dr. Federico Angiolini, Paolo Meloni, Prof. David Atienza, Prof. Salvatore Carta, Prof. Luigi Raffo, Prof. Luca Benini, and Prof. Giovanni De Micheli.

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

5. To achieve design closure and fast time-to-market, the actual physical layer measures are considered during the high-level topology synthesis phase itself. The timing information of switches and links are accurately characterized from layouts. We model the maximum frequency that can be supported by a switch as a function of the switch size. During the synthesis process, we steer the algorithms to only synthesize those switches that would support the desired NoC frequency. From the floorplan of the NoC design, estimates of the length of the NoC wires are obtained, which are used to detect timing violations on the interconnects early in the design cycle.
6. The tool automatically tunes several important NoC architectural parameters (such as the frequency of operation, flit-width) during the synthesis process.
7. SUNFLOOR is seamlessly integrated with back-end tools. SUNFLOOR directly feeds the floorplan information of the NoC to standard industrial placement&routing tools. The entire layout of the NoC can then be obtained from the placement&routing tools.

The layout of a multi-media MPSoC with the NoC designed using the methodology is presented later in the chapter (in Section 5.4.2). It achieves a post-layout clock frequency close to 900 MHz. We could design the NoC architecture from input specifications to layout in 4 hours, a process that used to take weeks. A layout level comparison with a hand-designed architecture for this example is also presented, which shows that the automatic design methodology produces good results (in terms of power consumption and performance), matching those of carefully hand-crafted designs. Experiments on several MPSoC benchmarks show large power, performance, and wire-length improvements when compared to standard topologies. Despite the very large design space considered, due to the use of fast algorithms and tools, the design process completes in reasonable time for all the experiments (see Section 5.4.1).

5.1.1 Background on NoC Topology Synthesis

The standard topologies (mesh, torus, etc.) that have been used in macro-networks result in poor performance and have large power and area overhead when used for MPSoCs. A major motivation for the use of NoCs is the fact that the interconnect structure and wiring complexity can be well controlled. When the interconnect is structured, the number of timing violations that occur during the physical design (floorplanning and wire routing) phase is minimum. Such design predictability is critical for today's MPSoCs for achieving timing closure. It leads to faster design cycle, reduction in the number of design respins, and faster time-to-market. As the wire delay as a fraction of gate delay is increasing with each technological generation, having shorter wires is even more important for future MPSoCs. Early works on NoC topology design assumed that using regular topologies (such as mesh) would lead to regular and predictable layouts [43]. While this may be true for designs with homogeneous processing cores and memories, this is not true for most MPSoCs

Table 5.1 Topology comparisons

Parameter	Mesh	Application-specific
Power (mW)	301.78	79.64
Hop-count	2.58	1.67
Total wire-length (mm)	185.72	145.37
Design area (mm^2)	51.01	47.68

as they are typically composed of heterogeneous cores. This is due to the fact that the core sizes of the MPSoC are highly nonuniform and the floorplan of the design does not match the regular, tile-based floorplan of standard topologies [33]. An application-specific NoC with structured wiring, which satisfies the design objectives and constraints is required to have feasible NoC designs.

As a motivating example, the network power consumption (switch and link power consumption), hop-count, wire-length, and design area of two different NoC topologies for a video processor MPSoC with 42 cores is presented in Table 5.1. The first topology is a mesh, while the second is a custom topology generated using SUNFLOOR tool. The wire-lengths and design area are obtained from floorplanning of the NoC designs. The detailed explanation of the topologies and the floorplanning process is described later in this chapter (Section 5.3). The custom topology leads to a 3.8× reduction in network power consumption, a 1.55× reduction in average hop-count, and a 1.28× reduction in total length of wires when compared to the mesh.

5.1.2 Background on Deadlock-Free NoC Design

The deadlocks that can occur in NoCs can be broadly categorized into two classes: *routing-dependent* deadlocks and *message-dependent* deadlocks [84–87, 94]. Routing-dependent deadlocks occur when there is a cyclic dependency of resources created by the packets on the various paths in the network [94].

Message-dependent deadlocks occur when interactions and dependencies are created between different message types at network endpoints, when they share resources in the network. Even when the underlying network is designed to be free from routing-dependent deadlocks, the message-level deadlocks can block the network indefinitely, thereby affecting the proper system operation.

Example 2 An example of a situation where a message-dependent deadlock occurs is presented in Figure 5.1. In this case, two of the cores are masters and two other cores are slaves. In this system, we assume two types of messages: *request* and *response*. Consider the following situation: Master 1 sends a request to Slave 1 (*Req 1*), Slave 1 is replying to a previously issued request to Master 1 (*Resp 1*) and at the same time, Slave 2 sends a response to Master 2 (*Resp 2*). When requests and responses share the same links, *Resp 2* is waiting for link 1, which is used by *Req 1*,

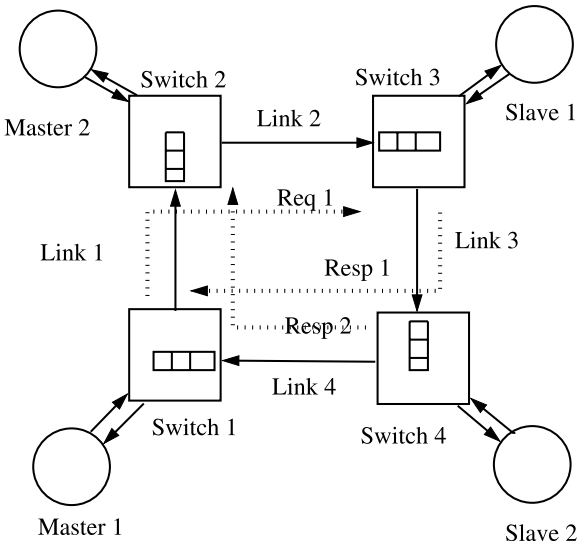


Fig. 5.1 Example of message-dependent deadlock

and *Resp 1* waits for link 4 used by *Resp 2*. Meanwhile, *Req 1* is waiting for Slave 1, the operation of which has been stalled as *Resp 1* could not complete. Thus, none of the messages can move ahead, leading to a deadlock situation. An interesting point to note here is that message-level deadlocks can be avoided if the receivers have infinitely large buffering or if they have perfectly ideal operation (consuming all received data instantly), which would avoid queuing of the packets in the network. Obviously, such a solution is not feasible in practice.

For proper system operation, it is critical to remove both routing and message-dependent deadlocks in the network. It is also important to achieve deadlock freedom with minimum NoC area and power overhead. In the topology synthesis process, we integrate methods to find paths that are free from both routing and message-dependent deadlocks. This is explained in detail in Section 5.3.

5.2 Input Models

5.2.1 Area, Power Models

We have built accurate analytical models for calculating the power consumption, area, and delay of the ×pipes network components [89]. To get an accurate estimate of these parameters, the place&route of the components is performed using SoC Encounter and accurate wire capacitances and resistances are obtained, as back-annotated information from the layout, with a 0.13 µm technology library. The switching activity in the network components is varied by injecting functional traffic. The capacitance, resistance, and the switching activity report are combined to estimate power consumption using Synopsys PrimePower [98].

A large number of implementation runs were performed, varying several parameters, such as the number of input, output ports, link-width, and the amount of switching activity at the layout level for the NoC switches. When the size of a NoC switch increases, the size of the arbiter and the crossbar matrix inside the switch also increases, thereby increasing the critical path of the switch. To have accurate delay estimates of the switches, we model the maximum frequency that can be supported by the switches, as a function of the switch size, presented in Figure 5.2.

We used linear regression to build analytical models for the area and power consumption of the components as a function of these parameters. Due to the intrinsic modularity and symmetry of NoC components, the models built are very accurate (with maximum and mean error of less than 7% and 5%, respectively) when compared to the actual values. Power consumption on the wires is also obtained at the layout level. As in the ×pipes architecture, each core is connected to a separate NI [34], we consider the power consumption of the NI to be part of the power consumption of the core.

The impact of the targeted frequency of operation on the area and energy consumption of an example 5 × 5 switch obtained from layout-level estimates is presented in Figure 5.3. Note that we plot the energy values (in power/MHz) instead of

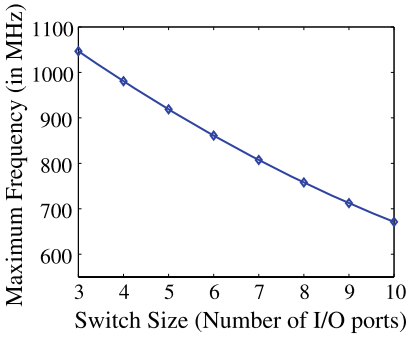
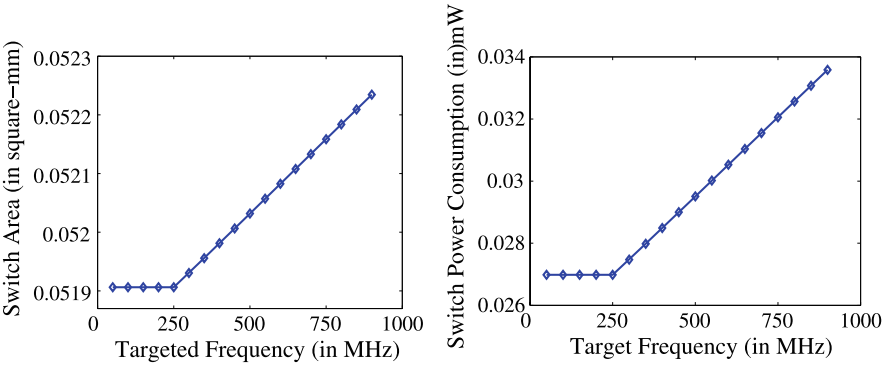


Fig. 5.2 Maximum frequency variation with switch size



(a) Impact on switch area (b) Impact on switch energy

Fig. 5.3 Impact of frequency on the area and energy of a 5 × 5 switch, for 0.13 µm technology

the total power, so that the inherent increase in power consumption due to increase in frequency is not observed in the plot. When the targeted frequency of operation is below a certain frequency, referred to as the nominal operating frequency (around 250 MHz in the plots), the area and energy values for the switch remains the same. However, as the targeted frequency increases beyond the nominal frequency, the area and energy values start increasing linearly with frequency. This is because the synthesis tool (such as Synopsys DC [98]) tries to match the desired high operating frequency by utilizing faster components that have large area and energy overhead. When performing the area, power estimates, we also model this impact of desired operating frequency on the switch area, power consumption.

5.2.2 Traffic Models

The traffic characteristics of the application are represented by a graph, as presented in the previous chapter, defined here again for convenience.

Definition 9 The core graph is a directed graph, $G(V, E)$ with each vertex $v_i \in V$ representing a core and the directed edge (v_i, v_j), denoted as $e_{i,j} \in E$, representing the communication between the cores v_i and v_j. The weight of the edge $e_{i,j}$, denoted by $comm_{i,j}$, represents the sustained rate of traffic flow from v_i to v_j weighted by the criticality of the communication. The set F represents the set of all traffic flows, with value of each flow, f_k, $\forall k \in 1, \ldots, |F|$, representing the sustained rate of flow between the source (s_k) and destination (d_k) vertices of the flow.

The edges of the core graph are annotated with the sustained rate of traffic flow, multiplied by the criticality level of the flow, as done in the previous chapter.

Definition 10 The message type for each flow f_k, $\forall k \in 1, \ldots, |F|$, is represented by $mtype_k$.

As an example, when a system has request and response message types, the $mtype_k$ value can be set to 0 for request messages and 1 for response messages.

5.3 Design Algorithms

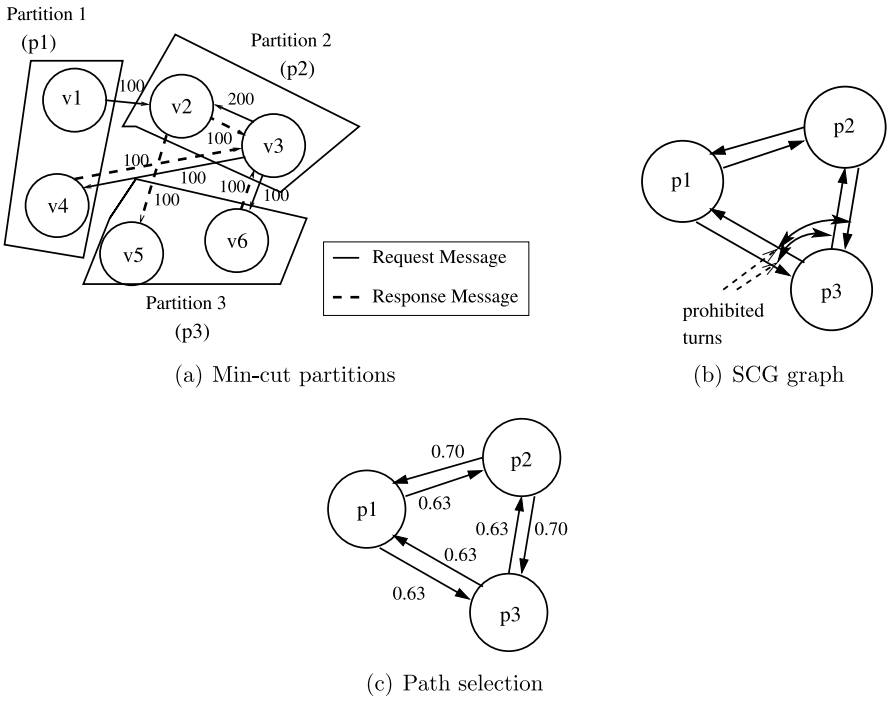
The algorithms for the topology design process are explained in this section. In the first step of Algorithm 2, a design point θ is chosen from the set of available or interesting design points ϕ for the NoC architectural parameters. In the current implementation, the synthesis engine automatically tunes two critical NoC parameters: operating frequency ($freq_\theta$) and link-width (lw_θ). As both frequency and link-width parameters can take a large set of values, considering all possible combinations of values would be infeasible to explore. The system designer has to trim down the

Algorithm 2 Topology Design Algorithm

1: Choose design point θ from ϕ: $freq_\theta$, lw_θ
2: **for** $i = 1$ to $|V|$ **do**
3: Find i min-cut partitions of the core graph
4: Establish a switch with N_j inputs and outputs for each partition, $\forall j \in 1, \ldots, i$. N_j is the number of vertices (cores) in partition i. Check for bandwidth constraint violations
5: Build *Switch Cost Graph* (*SCG*) with edge weights set to 0
6: Build *Prohibited Turn Set* (*PTS*) for *SCG* to avoid deadlocks
7: Set ρ to 0
8: Find paths for flows across the switches using function *PATH_COMPUTE (i, SCG, ρ, PTS, θ)*
9: Evaluate the switch power consumption and average hop-count based on the selected paths
10: Repeat steps 8 and 9 by increasing ρ value in steps, until the hop-count constraints are satisfied or until ρ reaches ρ_{thresh}
11: If ρ_{thresh} reached and hop-count not satisfied, go to step 2
12: Perform floorplan and obtain area, wire-lengths. Check for timing violations and evaluate power consumption on wires
13: If target frequency matches or exceeds $freq_\theta$, and satisfies all constraints, note the design point
14: **end for**
15: Repeat steps 2–14 for each design point available in θ
16: For the best topology and design point, generate information for ×pipesCompiler and Cadence SoC Encounter

exploration space and give the interesting design points for the parameters. The designer usually has knowledge of the range of these parameters. As an example, the designer can choose the set of possible frequencies from minimum to a maximum value, with allowed frequency step sizes. Similarly, the link data widths can be set to multiples of 2, within a range (say from 16 bits to 128 bits). Thus, we get a discrete set of design points for ϕ, as done in [53]. In all the experiments, 8 frequency steps and 4 link-width steps are used, providing 32 discrete design points in the set ϕ. The rest of the topology design process (steps 3–15 in Algorithm 2) is repeated for each design point in ϕ.

As the topology synthesis and mapping problem is NP-hard [48], we present efficient heuristics to synthesize the best topology for the design. For each design point θ, the algorithm synthesizes topologies with different numbers of switches, starting from a design where all the cores are connected through one big switch until the design point where each core is connected to only one switch. The reason for synthesizing these many topologies is that it cannot be predicted beforehand whether a design with few bigger switches would be more power efficient than a design with more smaller switches. A larger switch has more power consumption than a smaller switch to support the same traffic, due to its bigger crossbar and



(a) Min-cut partitions

(b) SCG graph

(c) Path selection

Fig. 5.4 Algorithm examples

arbiter. On the other hand, in a design with many smaller switches, the packets may need to travel more hops to reach the destination. Thus, the total switching activity would be higher than a design with fewer hops, which can lead to higher power consumption.

For the chosen switch count i, the input core graph is partitioned into i min-cut partitions (step 3). The partitioning is done in such a way that the edges of the graph that are cut between the partitions have lower weights than the edges that are within a partition (refer to Figure 5.4(a)) and the number of vertices assigned to each partition is almost the same. Thus, those traffic flows with large bandwidth requirements or higher criticality level are assigned to the same partition, and hence use the same switch for communication. Hence, the power consumption and the hop-count for such flows will be smaller than for the other flows that cross the partitions. For partitioning, we use Chaco, a hierarchical graph partitioning tool [93].

At this point, the communication traffic flows within a partition have been resolved. In steps 5–9, the connections between the switches are established to support the traffic flows across the partitions. In step 5, the *Switch Cost Graph* (*SCG*) is generated.

Definition 11 The *SCG* is a fully connected graph with i vertices, where i is the number of partitions (or switches) in the current topology.

Please note that the *SCG* does not imply the actual physical connectivity between the different switches. The actual physical connectivity between the switches is established using the *SCG* in the *PATH_COMPUTE* procedure, which is explained in the following paragraphs.

In NoCs, wormhole flow control [94] is usually employed to reduce switch buffering requirements and to provide low-latency communication [30, 33]. With wormhole flow control, deadlocks can happen during routing of packets due to cyclic dependencies of resources (such as buffers) [94]. We preprocess the *SCG* and prohibit certain turns to break such cyclic dependencies. This guarantees that deadlocks will not occur when routing packets. For finding the set of turns that need to be prohibited to break cycles, we use the turn prohibition algorithm presented in [6, 87]. The algorithm has polynomial time complexity (very fast in practice; see Section 7.6) and guarantees that at most $1/3$ of the total number of turns would be prohibited to remove cycles. The algorithm also guarantees connectivity between all nodes in the *SCG* after prohibiting the turns. From the algorithm, we build the *Prohibited Turn Set* (*PTS*) for the *SCG*, which represents the set of turns that are prohibited in the graph. To provide guaranteed deadlock freedom, any path for routing packets should not take these prohibited turns. These concepts are illustrated in the following example.

Example 3 The min-cut partitions of the core graph of the filter example (from Figure 4.2(a)) for 3 partitions is shown in Figure 5.4(a). The *SCG* for the 3 partitions is shown in Figure 5.4(b). After applying the turn prohibition algorithm from [87], the set of prohibited turns is identified. In Figure 5.4(b), the prohibited turns are indicated by circular arcs in the *SCG*. For this example, both the turns around the vertex P3 are prohibited to break cycles. So, no path that uses the switch P3 as an intermediate hop can be used for routing packets.

The actual physical connections between the switches are established in step 8 of Algorithm 2 using the *PATH_COMPUTE* procedure. The objective of the procedure is to establish physical links between the switches and to find paths for the traffic flows across the switches. Here, we only present the procedure where the user's design objective is to minimize power consumption. The procedure for the other two cases (with hop-count as the objective and with linear combination of power and hop-count as objective) follow the same algorithm structure, but with different cost metrics.

An example illustrating the working of the *PATH_COMPUTE* procedure is presented in Example 4. In the procedure, the flows are ordered in decreasing rate requirements, so that bigger flows are assigned first. The heuristic of assigning bigger flows first has been shown to provide better results (such as lower power consumption and more easily satisfying bandwidth constraints) in several earlier works [6, 46]. For each flow in order, we evaluate the amount of power that will be dissipated across each of the switches, if the traffic for the flow used that switch. This power dissipation value on each switch depends on the size of the switch, the amount of traffic already routed on the switch and the architectural parameter point

(θ) used. It also depends on how the switch is reached (from what other switch) and whether an already existing physical channel will be used to reach the switch or a new physical channel will have to be opened. This information is needed, because opening a new physical channel increases the switch size, and hence the power consumption of this flow and of the others that are routed through the switch. These marginal power consumption values are assigned as weights on each of the edges reaching the vertex representing that switch in the *SCG*. This is performed in steps 8 and 11 of the procedure.

When opening a new physical link, we also check whether the switch size is small enough to satisfy the particular frequency of operation. As the switch size increases, the maximum frequency of operation it can support reduces (as noted earlier in Section 5.2). This information is obtained from the placement&routing of the switches, taken as an input to the algorithms. The message type that is supported by a link between any two switches i and j is represented by $MType(i, j)$. Whenever a path is established for a flow, the links that are newly instantiated in the path are assigned the same message type as the flow. When choosing a path for a flow, we check whether the existing links in the path support the same message type as the flow (step 7 of Algorithm 3). Thus, flows with different message types are mapped onto different physical links in the NoC, thereby removing the chances of a message-level deadlock.

Once the weights are assigned, choosing a path for the traffic flow is equivalent to finding the least cost path in the *SCG*. This is done by applying Dijkstra's shortest path algorithm [95] in step 15 of the procedure. When choosing the path, only those paths that do not use the turns prohibited by *PTS* are considered. The size of the switches and the bandwidth values across the links in the chosen path are updated and the process is repeated for other flows.

Example 4 Let us consider the example from Figure 5.4(a). The input core graph has been partitioned into 4 partitions. We assume 2 different message types: *request* and *response* for the various traffic flows. Each partition p_i corresponds to the cores attached to the same switch. Let us consider routing the flow with a bandwidth value of 100 MB/S between the vertices $v1$ and $v2$, across the partitions $p1$ and $p2$. The traffic flow is of the message type *request*. Initially no physical paths have been established across any of the switches. If we have to route the flow across a link between any two switches, we have to first establish the link. The cost of routing the flow across any pair of switches is obtained. The edges between the switches are annotated by the cost (marginal increase in power consumption) of sending the traffic flow through the switches (Figure 5.4(c)). The cost on the edges from $p2$ are different from the others due to the difference in initial traffic rates within $p2$ when compared to the other switches. This is because the switch $p2$ has to support flows between the vertices $v2$ and $v3$ within the partition. The least cost path for the flow, which is across switches $p1$ and $p2$ is chosen. Now we have actually established a physical path and a link between these switches. We associate the message type *request* for this particular link. This is considered when routing the other flows and only those traffic flows that are of request type can use this particular physical link.

Algorithm 3 *PATH_COMPUTE*$(i, SCG, \rho, PTS, \theta)$

1: Initialize the set $PHY(i1, j1)$ to false and $Bw\_avail(i1, j1)$ to $freq_\theta \times lw\theta$, $\forall i1, j1 \in 1, \ldots, i$
2: Initialize $switch\_size\_in(j)$ and $switch\_size\_out(j)$ to N_j, $\forall j \in 1, \ldots, i$. Find $switching\_activity(j)$ for each switch, based on the traffic flow within the partition
3: **for** each flow $f_k, k \in 1, \ldots, |F|$ in decreasing order of f_c **do**
4: **for** $i1$ from 1 to i and $j1$ from 1 to i **do**
5: {Find the marginal cost of using link $i1, j1$}
6: {If physical link exists, can support the flow and is of the same message type}
7: **if** $PHY(i1, j1)$ and $Bw\_avail(i1, j1) \geq f_c$ and $(MType(i1, j1) = mtype_k)$ **then**
8: Find $cost(i1, j1)$, the marginal power consumption to reuse the existing link
9: **else**
10: {We have to open new physical link between $i1, j1$}
11: Find $cost(i1, j1)$, the marginal power consumption for opening and using the link. Evaluate whether switch frequency constraints are satisfied
12: **end if**
13: **end for**
14: Assign $cost(i1, j1)$ to the edge $W(i1, j1)$ in *SCG*
15: Find the least cost path between the partitions in which source (s_k) and destination (d_k) of the flow are present in the *SCG*. Choose only those paths that have turns not prohibited by *PTS*
16: Update *PHY*, *Bw_avail*, *switch_size_in*, *switch_size_out*, *switching_activity*, *MType* for chosen path
17: **end for**
18: Return the chosen paths, switch sizes, connectivity

We also note the size and switching activity of these switches that have changed due to the routing of the current flow.

The *PATH_COMPUTE* procedure returns the sizes of the switches, connectivity between the switches, and the paths for the traffic flows. The objective function for establishing the paths is initially set to minimizing power consumption in the switches. Once the paths are established, if hop-count constraints are not satisfied, the algorithm gradually modifies the objective function to minimize the hop-count as well, using the parameter ρ (in steps 7, 10, and 11 of Algorithm 2). The upper bound for ρ, denoted by ρ_{thresh}, is set to the value of power consumption of the flow with maximum rate, when it crosses the maximum size switch in the *SCG*. At this value of ρ, for all traffic flows, it is beneficial to take the path with least number of switches, rather than the most power efficient path. The ρ value is varied in several steps until the hop-count constraints are satisfied or until it reaches ρ_{thresh}.

In the next step (step 12, Algorithm 1), the algorithm invokes the floorplanner to compute the design area and wire-lengths. The floorplanner minimizes a dual-objective function of area and wire-length, with equal weights assigned to both. The floorplanner used [92] also supports soft cores, fixed pin/pad locations, and aspect ratio constraints for the generated design. From the obtained wire-lengths, the power consumption across the wires is calculated. Also, the length of the wires is evaluated to check any timing violations that may occur at the particular frequency ($freq_\theta$). In the end, the tool chooses the best topology (based on the user's objectives) that satisfies all the design constraints. At the last step, for the synthesized topology, the algorithm automatically generates the information required for the ×pipesCompiler tool for network instantiation and the SoC Encounter tool to perform placement&routing.

The presented NoC synthesis process scales polynomially with the number of cores in the design. The number of topologies evaluated by the methodology also depends linearly on the number of cores. Thus, the algorithms are highly scalable to a large number of cores and communication flows. The synthesis time for several different MPSoC benchmarks is presented in Section 5.4.1.

5.4 Experiments and Case Studies

5.4.1 Experiments on MPSoC Benchmarks

We have applied the topology design procedure to six different MPSoC benchmarks: *video processor* (*VPROC-42 cores*), *MPEG4 decoder* (*12 cores*), *Video Object Plane Decoder* (*VOPD-12 cores*), *Multi-Window Display application* (*MWD-12 cores*), *Picture-in-Picture application* (*PIP-8 cores*), *and IMage Processing application* (*IMP-23 cores*).

For comparison, we have also generated mesh topologies for the benchmarks by modifying the design procedure to synthesize NoCs based on mesh structure. To obtain mesh topologies, we generate a design with each core connected to a single switch and restrict the switch sizes to have 5 input/output ports. We also generated a variant of the basic mesh topology: *optimized mesh* (*opt-mesh*), where those ports and links that are unused by the traffic flows are removed.

The core graph and the floorplan for the custom topology synthesized by the tool for one of the benchmarks (VOPD) are shown in Figure 5.5. The network power consumption (power consumption across the switches and links), average hop-count and design area results for the different benchmarks are presented in Table 5.2. Note that the average hop-count is the same for mesh and opt-mesh, as in the opt-mesh only the unused ports and links of the mesh have been removed and the rest of the connections are maintained. The custom topology results in an average of 2.78× improvement in power consumption and 1.59× improvement in hop-count when compared to the standard mesh topologies. The area of the designs with the different topologies is similar, thanks to efficient floorplanning of the designs. It can be seen

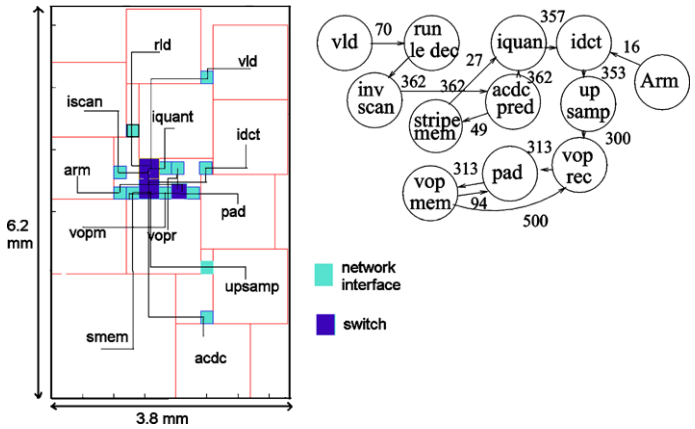


Fig. 5.5 VOPD custom topology floorplan and core graph

Table 5.2 Comparisons with standard topologies

Appl.	Topol.	Power (mW)	Avg. hops	Area mm^2	Time (mins)
VPROC	Custom	79.64	1.67	47.68	68.45
	Mesh	301.82	2.58	51.01	
	Opt-mesh	136.11	2.58	50.51	
MPEG4	Custom	27.24	1.52	13.49	4.04
	Mesh	96.82	2.17	15.00	
	Opt-mesh	60.97	2.17	15.01	
VOPD	Custom	30.03	1.33	23.56	4.47
	Mesh	95.94	2.03	23.85	
	Opt-mesh	46.48	2.03	23.79	
MWD	Custom	20.53	1.15	15.00	3.21
	Mesh	90.17	2.02	13.62	
	Opt-mesh	38.60	2.02	13.81	
PIP	Custom	11.71	1	8.95	2.07
	Mesh	59.87	2.02	9.61	
	Opt-mesh	24.53	2.03	9.34	
IMP	Custom	52.13	1.44	29.66	31.52
	Mesh	198.92	2.11	29.41	
	Opt-mesh	80.15	2.11	29.41	

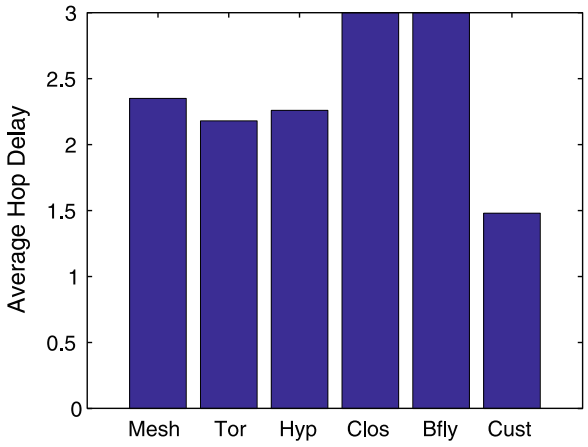


Fig. 5.6 Performance comparisons

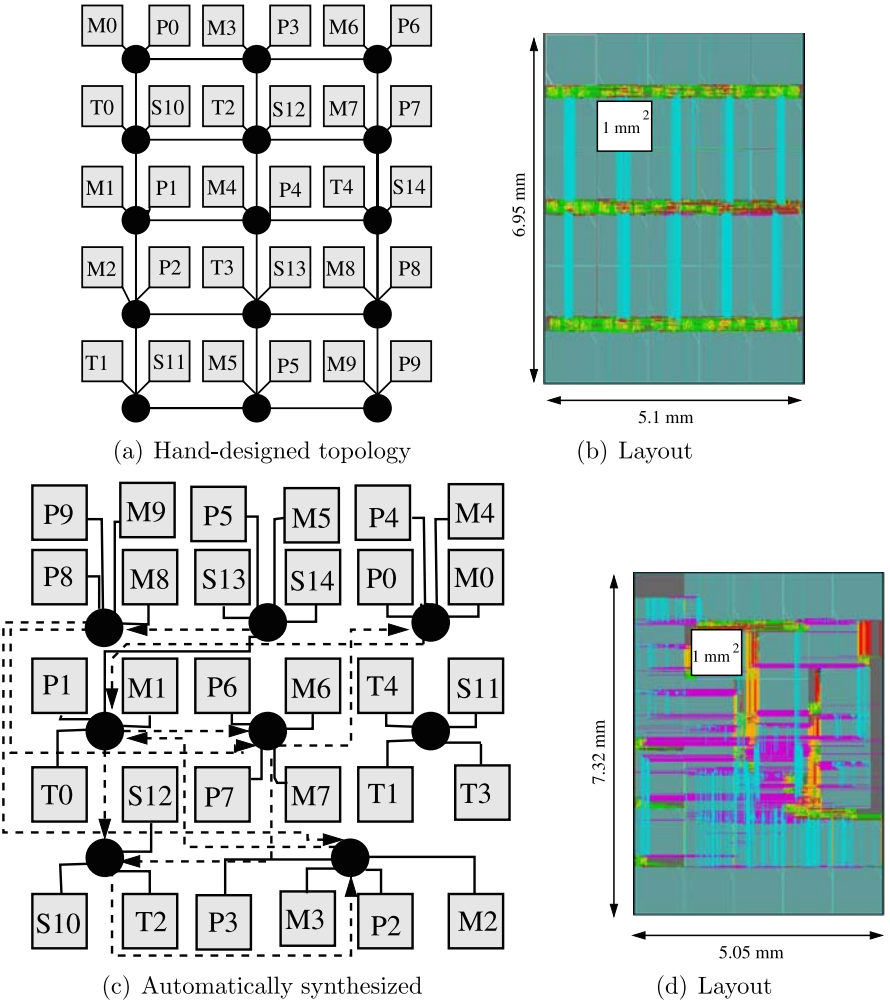
from Figure 5.5 that only very little slack area is left in the floorplan. This is because we consider the area of the network elements during the floorplanning process, and not after the floorplanning of blocks. The total run-time of the topology synthesis and architectural parameter setting process for the different benchmarks is presented in Table 5.2. Given the large problem sizes and very large solution space that is explored (8 different frequency steps, 4 different link-widths, 42 cores for VPROC and several calls to the floorplanner) and the fact that the NoC parameter setting and topology synthesis are important phases, the run-time of the engine is not large. This is mainly due to the use of hierarchical tools for partitioning and floorplanning and the development of fast heuristics to synthesize the topology.

We also performed comparisons of synthesized topology against several other standard topologies. For mapping the cores onto the standard topologies, we use the SUNMAP tool, presented in the previous chapter. We optimized the topologies for performance, subject to the design constraints. The comparisons against 5 standard topologies (mesh, torus, hypercube, Clos, and butterfly) for an image processing benchmark with 25 cores is presented in Figure 5.6. The custom topology synthesized by the method shows large performance improvements (an average of 1.73×) over the standard topologies.

As an interesting observation, we found that prohibiting certain turns to avoid deadlocks during routing had a negligible impact on the power and performance results for all of the benchmarks. This was because even if some turns were avoided, the path computation procedure could easily find other paths with low cost, as several alternative low cost paths exist between each source and destination in the *SCG* (refer to Section 5.3).

5.4.2 Layout-Level Comparisons

We had earlier manually developed a NoC design for a MPSoC that runs multimedia benchmarks [88]. The design consists of 30 cores: 10 ARM7 processors with



(a) Hand-designed topology (b) Layout

(c) Automatically synthesized (d) Layout

Fig. 5.7 (**a**), (**b**) Hand-designed topology and layout. M: ARM7 processors, T: traffic generators, P, S: private and shared slaves (**c**), (**d**) Automatically synthesized topology and layout. In (**c**), bidirectional links are *solid* and unidirectional links are *dotted*

caches, 10 private memories (a separate memory for each processor), 5 custom traffic generators, 5 shared memories and devices to support interprocessor communication. The hand-designed NoC has 15 switches connected in a 5×3 quasi-mesh network (2 cores connected to each switch), shown in Figure 5.7(a). The design is highly optimized, with the private memories being connected to the processors across a single switch and the shared memories distributed around the switches. The layout of the design (presented in Figure 5.7(b)) was performed using SoC Encounter and the mesh structure was maintained in the layout. Each of the cores has an area of 1 mm^2 [88] in the design. The entire process, from topology specification

to layout generation took weeks. The post-layout NoC could support a maximum frequency of operation of 885 MHz, which is determined by the critical path in the switch pipeline. The power consumption of the topology for functional traffic has been evaluated to be 368 mW.

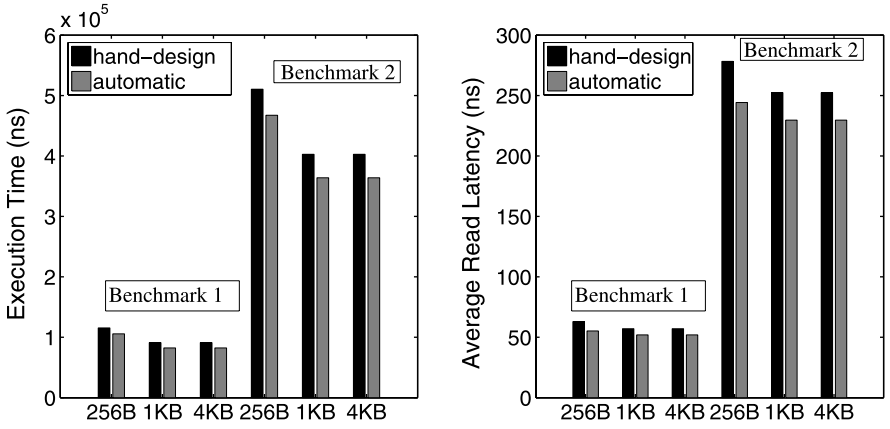
We apply the topology synthesis process with the objective of minimizing power consumption to automatically synthesize the NoC for this application. We set the design constraints and the required frequency of operation to be the same (885 MHz) as that of the hand-designed topology. The synthesized NoC topology and the layout obtained using SoC Encounter are presented in Figures 5.7(c) and (d). The synthesized topology has fewer switches (8 switches) than the hand-designed topology. It can support the same maximum frequency of operation (885 MHz), without any timing violations on the wires. As we considered the wire-lengths during the synthesis process to estimate the frequency that could be supported, we could synthesize the most power efficient topology that would still meet the target frequency. To reach such a design point manually would require several iterations of topology design and place&route phases, which is a very time consuming process.

Layout level power consumption calculations on functional traffic show that the synthesized topology has 277 mW power consumption, which is 1.33× lower than the hand-designed topology. Given the fact that the hand-designed topology is highly optimized, with much of the communicating traffic (which is between the ARM cores and their private memories) traversing only one switch, these savings are achieved entirely from efficiently spreading the shared memories around the different switches. The layout of the hand-designed NoC was manually optimized to a large extent (by moving switches, network interfaces) to reduce the area of the design. The layout of the synthesized topology is obtained completely automatically, and still the area of the design is close to that of the manual design (only a marginal 4.3% increase in area).

We perform cycle-accurate simulations of the hand-designed and the synthesized NoCs for two multimedia benchmarks. The total application time for the benchmarks (including computation time) and the average packet latencies for read transactions for the topologies are presented in Figures 5.8(a) and (b). The custom topology not only matches the performance of the hand-designed topology, but provides an average of 10% reduction in total execution time and of 11.3% in packet latency.

5.4.3 Impact of Frequency Constraints

The maximum frequency of operation that can be supported by the NoC switches depends on the number of switch I/O ports, as indicated earlier in Figure 5.3(b). In this subsection, we study the impact of the required NoC frequency on the topology synthesis process. We consider the multi-media MPSoC considered in Section 5.4.2 and apply the SUNFLOOR tool to synthesize the most power-efficient topology for different operating frequency constraints. The number of switches and maximum switch sizes (maximum over the number of input and output ports of all the



(a) Execution time (b) Average read latency

Fig. 5.8 Run-time and latency for different cache sizes

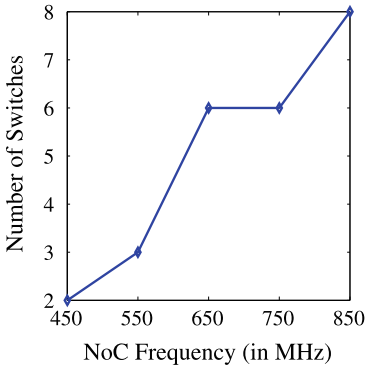


Fig. 5.9 Topology size variations with NoC frequency

switches) used in the synthesized topologies for different NoC frequencies are presented in Figures 5.9 and 5.10. From these plots, we can infer that at low operating frequencies, a topology with few but large switches results in the most power optimal design. This is due to the fact that the increase in power consumption is mostly linear with the increase in switch size [89]. Thus, in a design with fewer switches, the traffic flows traverse shorter paths, thereby leading to more power optimal designs. But as the required NoC operating frequency increases, the timing delay constraints cannot be met by large switches, thereby the optimal design point moves to a topology with smaller, but more switches. As the tool flow automatically considers the frequency constraint of the switches as well, we am able to prune the infeasible design points (that violate the timing constraints) early in the design process.

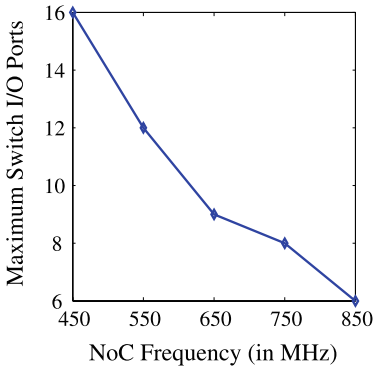


Fig. 5.10 Switch size variations with NoC frequency

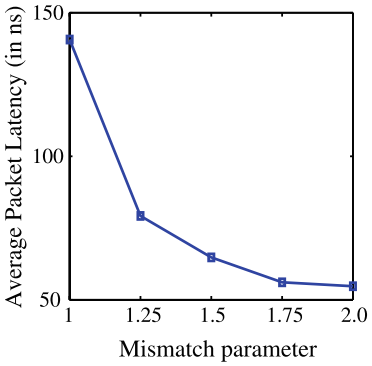


Fig. 5.11 Dynamic effects

5.4.4 Handling Dynamic Effects

When the designed NoC is simulated, there can be some mismatch between the observed traffic patterns and the initial traffic estimates. This may be either because of inaccurate traffic models or because of dynamic effects, such as congestion. Note that it will be too time consuming to simulate each topology during the synthesis process. To bridge the gap between topology synthesis and simulation, we use the *mismatch* parameter; the input traffic rates are multiplied by the value of this parameter. The parameter is fed as an input to the synthesis engine. It is initially set to 1 and the user can manually tune the parameter and redesign the NoC, until the simulations satisfy the required performance level. The effect of increasing the parameter on performance for the MPEG4 NoC is presented in Figure 5.11.

5.5 Summary

Having a power and latency efficient NoC architecture that closely matches the application traffic characteristics is key to have an efficient MPSoC implementation. Synthesizing such NoC architecture is nontrivial, given the large design space that

needs to be explored. In this chapter, we have presented SUNFLOOR, a tool that automates the process, generating efficient NoCs that satisfy the design constraints of the application. To have fewer design respins and faster time-to-market, we have integrated the architectural synthesis models with back-end physical design models, thereby bridging a big design gap in NoC synthesis. The synthesis process also considers the wiring complexity of the NoC to accurately estimate interconnect delay and power consumption and produces networks that are free from deadlocks. We have shown a layout-level implementation of the NoC for a multi-media MPSoC, validating the Netchip design flow.

Chapter 6
Supporting Multiple Applications

In the previous three chapters, we presented approaches to design the NoC to match the traffic requirements of an application. As technology advances, it becomes cost-effective to integrate several different applications or use-cases onto a single SoC chip. As an example, the *PNX8550* (*Viper2*) set-top box SoC based on the Philips Nexperia platform has multiple resolution video processing capabilities (like high definition, standard definition), multiple picture modes (like split-screen, picture-in-picture), video recording features, high speed internet access, file transfer services, etc.

Current state-of-the-art SoCs also allow several of the use-cases to run in parallel. As an example, in a set-top box SoC, video display, and recording applications can run in parallel, where the recorder could potentially record a different program than what is being displayed on the screen. We refer to such use-cases that run in parallel as *compound modes* (Figure 6.1). The transition between the single use-case mode to compound mode needs to be smooth. As an example, when we start a new function such as video-recording in a set-top box, the video display that is currently going on should be unaffected. However, when there is a switching between compound modes, there can be a configuration time overhead to load the new set of use-cases, as shown in Figure 6.1. As the different use-cases have different functionalities, the communication characteristics can be very different across the use-cases. As an example, in Figure 6.2, a small fragment of the communication constraints for two different use-cases for the *Viper2* set-top box SoC is presented, where the bandwidth requirements for some of the traffic streams for the use-cases are different.

In this chapter,[1] we extend the synthesis approach to design NoCs that support multiple applications. To show the generality of the methods presented in this and preceding chapters, we apply the synthesis procedure to a different NoC design: the Ætheral architecture [30]. We integrate the synthesis tool with the Æthereal design flow [96], similar to the integration into the Netchip flow presented in last chapter.

The proposed synthesis process performs mapping, path selection, and resource reservation in the NoC that satisfies the communication constraints of multiple use-cases of the SoC. We consider compound modes, where two or more use-cases run in parallel, and automatically compute the communication constraints for such modes from the constituent use-cases. When there is switching between the use-cases that are run, there is a possibility of changing the paths and resource reservations in the NoC across the use-cases. The dynamic network reconfiguration can be applied when the use-case switching times are large and it helps in reducing the operating

[1] We would like to acknowledge the contributions of Dr. Andrei Radulescu, Martijn Coenen and Dr. Kees Goossens.

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

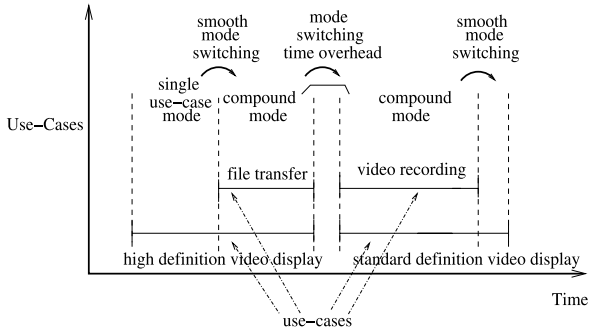


Fig. 6.1 Use-cases and compound modes

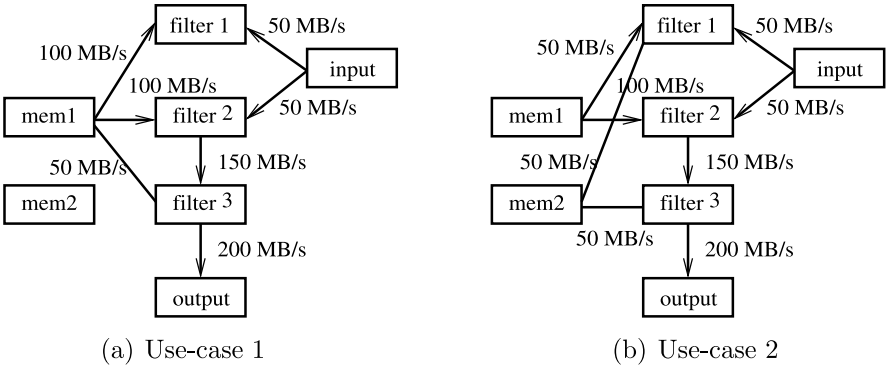


Fig. 6.2 Example use-cases

frequency and power consumption of the NoC. In the methodology, we preprocess the use-cases and identify the set of use-cases that need to share the same NoC configuration and use-case switching where the NoC configuration can be changed. We also explore the effect of dynamic voltage and frequency scaling (DVS/DFS) techniques for reducing the power consumption of the network across the different use-cases. We apply the methods to several SoC designs (set-top box, TV processor SoCs) and synthetic benchmarks to validate the design methodology. The methods are scalable to a large number of use-cases and are applicable even when the use-cases have very different communication characteristics.

6.1 The Æthereal NoC Architecture

In this section, we present the architecture of the Æthereal NoC, which provides support for predictable communication behavior and the mechanism for dynamic NoC configuration.

6.1.1 Switch/NI Architecture

The Æthereal NoC architecture interconnects IP blocks by connecting them to Network Interfaces (NIs), which convert the IP-block communication view (protocols such as DTL, AXI, and OCP) to the network communication view, which is packet-based. The NIs are then interconnected by a switch network, which may consist of multiple switches in different topologies. Æthereal supports a request-response transaction scheme, in which a master IP block may send a request (such as a read or a write) to a slave IP, and the slave IP might return a response (in case of a read, for example). This transaction scheme is implemented in Æthereal by a *connection*, which consists of a request channel and a response channel.

Many applications run on SoCs have real-time requirements, such as audio and video streaming applications. In order to make sure these applications can meet their requirements from the communication point-of-view, Æthereal offers so-called GT connections which provide bandwidth and latency guarantees on that connection. In order to provide bandwidth and latency guarantees, the Æthereal NoC uses a Time Division Multiple Access (TDMA) mechanism to divide time in multiple time slots, and then assigns each GT connection a number of slots. The result is a slot-table in each NI, stating which GT connection is allowed to enter the network at which time-slot. Once a GT connection is granted a slot, it is guaranteed a contention-free path to its destination NI. This is achieved by computing the slot table allocations for each NI such that two GT packets never use the same output link at any time on any switch. This slot allocation is computed at design time by a heuristic algorithm [6], and must be programmed in the NIs at run-time. For example, when a connection is given half of the slots, it will be granted half of the total bandwidth.

For traffic that has no real-time requirements, Æthereal implements Best-Effort (BE) connections. BE connections are allowed to enter the network whenever a slot is not used by a GT connection or when the slot is not allocated. Besides the allocation of slots for GT connections, both GT and BE connections need to have a path through the switch network. Æthereal uses a static path routing scheme, in which paths are inserted at the source NI of the traffic flow. These paths are computed at design-time, and must be programmed into the NIs at run-time. NoC configuration thus consists of programming both the slot-allocation and path for each connection.

6.1.2 Dynamic NoC Reconfiguration

When multiple applications (use-cases) must be supported by the NoC and they have different requirements, the paths and slot-tables allocations for these use-cases will vary. The tables therefore need to be stored in memory and loaded whenever a use-case switch is executed. As the on-chip memory available is mostly limited and as the number of use-cases is usually large, the off-chip memory can be used to store the paths and slot-tables for the different use-cases.

We investigated the overhead for the reconfiguration mechanism for the set-top box SoC. The amount of data required to store the path and slot-table information for each use-case is around 560 bytes. With 400 use-cases, the memory requirement for the reconfiguration mechanism is 224 KB. The time required to load the data from the memory and spread it around the NoC for an use-case is of the order of micro-seconds and the energy dissipation is of the order of micro-Joules. Using traditional mechanisms to scale the frequency and voltage of the system may require few milliseconds for configuration.

6.2 Design Methodology

The communication characteristics and constraints of the various use-cases of a SoC are the input to the design methodology (U1 $\cdots$ Un in Figure 6.3). The communication design constraints for each use-case includes the required bandwidth for various connections between the cores in the use-case, the maximum latency allowed for the connection, the QoS level required for the connection (like GT or BE), etc. An example fragment of the input file is presented in Figure 6.4. The user specifies the set of use-cases that can run in parallel (*PUC* in the figure). In the first phase of the design process, new use-cases are generated automatically to represent such parallel modes of operation.

For a multiple use-case SoC, when the system switches between use-cases, some timing overhead is incurred in loading the new use-case. This delay is mainly due to the fact that the new use-case's data and code need to be loaded, control signals need to be distributed to different parts of the design, and the already running use-case need to be gracefully shut down. This switching time varies with different use-cases

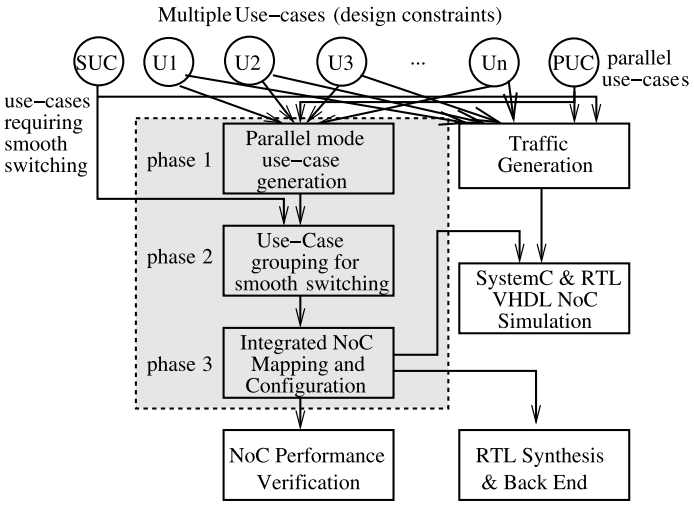
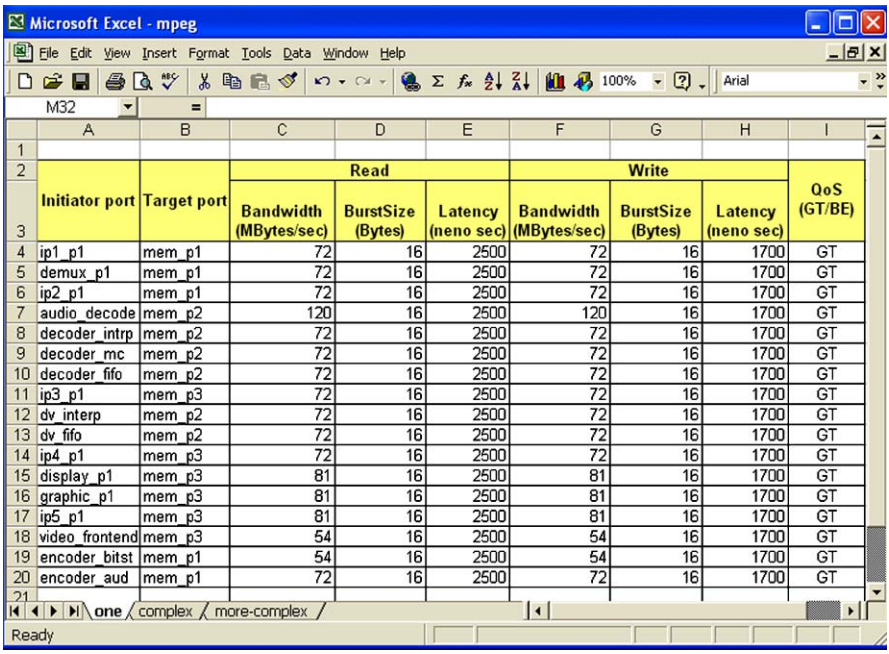


Fig. 6.3 Multi use-case NoC design methodology



Microsoft Excel - mpeg

File Edit View Insert Format Tools Data Window Help

M32

	Initiator port	Target port	Read			Write			QoS (GT/BE)
			Bandwidth (MBytes/sec)	BurstSize (Bytes)	Latency (neno sec)	Bandwidth (MBytes/sec)	BurstSize (Bytes)	Latency (neno sec)	
4	ip1_p1	mem_p1	72	16	2500	72	16	1700	GT
5	demux_p1	mem_p1	72	16	2500	72	16	1700	GT
6	ip2_p1	mem_p1	72	16	2500	72	16	1700	GT
7	audio_decode	mem_p2	120	16	2500	120	16	1700	GT
8	decoder_intrp	mem_p2	72	16	2500	72	16	1700	GT
9	decoder_mc	mem_p2	72	16	2500	72	16	1700	GT
10	decoder_fifo	mem_p2	72	16	2500	72	16	1700	GT
11	ip3_p1	mem_p3	72	16	2500	72	16	1700	GT
12	dv_interp	mem_p2	72	16	2500	72	16	1700	GT
13	dv_fifo	mem_p2	72	16	2500	72	16	1700	GT
14	ip4_p1	mem_p3	72	16	2500	72	16	1700	GT
15	display_p1	mem_p3	81	16	2500	81	16	1700	GT
16	graphic_p1	mem_p3	81	16	2500	81	16	1700	GT
17	ip5_p1	mem_p3	81	16	2500	81	16	1700	GT
18	video_frontend	mem_p3	54	16	2500	54	16	1700	GT
19	encoder_bitst	mem_p1	54	16	2500	54	16	1700	GT
20	encoder_aud	mem_p1	72	16	2500	72	16	1700	GT

one complex more-complex

Ready

Fig. 6.4 Example input file with design constraints for an MPEG application

and depends on the underlying computational architecture. Some use-cases represent control sequences that are critical and are loaded and run quickly. For many other use-cases, the switching time is of the order of hundreds of microseconds to milliseconds. In this time, we can reconfigure the paths and TDMA slot-tables in the NoC to match the communication characteristics of the use-cases. When the switching times are in the order of few milliseconds, we can also scale the supply voltage and NoC frequency to match the use-case characteristics, which can lead to a reduction in the NoC power consumption.

To accommodate NoC reconfiguration and to apply DVS/DFS, we perform two different groupings of the various use-cases. If two use-cases have a switching time of less than few microseconds, then they belong to the same *configuration group*. The use-cases within the same configuration group share the same path and slot-table reservations, as there would not be sufficient time to change the paths and TDMA time-slots across the use-cases. Similarly, if two use-cases that have switching time of less than few milliseconds, they belong to the same *DVS/DFS* group, that is, they use the same NoC operating frequency and voltage.

When some use-cases can run in parallel, we require a smooth transition between the single use-case mode to the parallel use-case mode, and thus they would belong to the same configuration and DVS/DFS group. Other use-cases that are in the same configuration or DVS/DFS group are given as an input to the design flow (*SUC* in Figure 6.3). In the second phase of the design, we preprocess the use-cases identifying the set of use-cases that can have reconfiguration and those that should

share the same NoC configuration. We also find the set of use-cases across which the *DVS*/*DFS* schemes can be applied. The detailed description of this phase is presented in Section 6.5.

In the third phase of the design, we perform mapping and NoC configuration. The objective of the mapping process is to design the smallest size NoC (in terms of the number of the switches used) that satisfies the design constraints of all the use-cases. We assume that all of the use-cases utilize the same mapping of cores onto the NoC components and only the paths and TDMA slot-tables can be potentially reconfigured across the different use-cases. This is because, if each individual use-case has a different mapping, then each core potentially needs to be connected to several different NIs, which may not be feasible because of physical layout restrictions and wiring complexity. The methods presented in this chapter can be easily extended to support even limited reconfiguration of the mapping across the different use-cases.

In the last phase of the design, the SystemC/VHDL code for the NoC design is generated and simulations of the design are performed. The NoC performance for the GT connections is also verified analytically in this step.

6.3 Use-Case Preprocessing

The set of use-cases that can run in parallel is specified by the user as an input. As the number of combinations of the use-cases can be large, it is a tedious process for the user to manually create use-cases to represent the parallel modes. In the first phase of the methodology, we automatically compute the bandwidth, latency requirements for such parallel modes from the individual use-cases. The bandwidth of a flow between two cores in such a compound mode is obtained by summing the bandwidth of the flows between the two cores across the use-cases that comprise the mode and the latency requirement of the flow is taken to be the minimum of the requirements of the flows across the different use-cases in the mode. Such compound modes are then taken as separate use-cases in the design flow.

To find the set of all use-cases that belong to the same configuration group, and hence need to have the same NoC configuration, we construct a *Switching Configuration Graph* (*SCG*).

Definition 12 The $SCG(SV, SE)$ is an undirected graph, with each vertex $sv_i \in SV$ representing an use-case and the undirected edge (sv_i, sv_j) (or (sv_j, sv_i)), representing the fact that the use-cases sv_i and sv_j belong to the same configuration group.

As an example, in Figure 6.5, a *SCG* graph for 10 use-cases is presented. The use-cases U_123, U_45 are automatically generated by the first phase of the design flow to represent the compound modes of operation where use-cases 1, 2, 3 and 4, 5, respectively, run in parallel. We require a smooth switching between use-cases 6

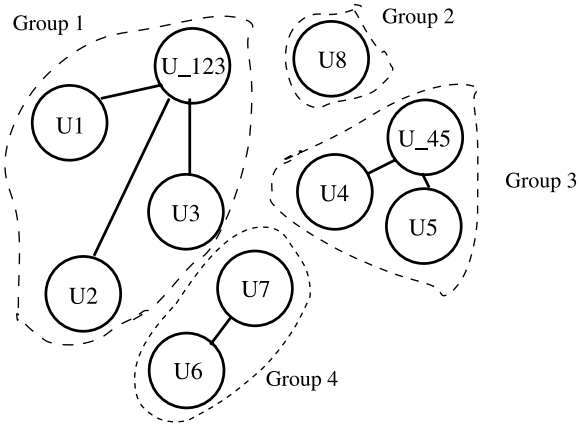


Fig. 6.5 Example *SCG*

Algorithm 4 Use-Case Grouping

1. Initialize $sv_i \in SV, \forall i \in 1, \ldots, |SV|$, unvisited.
2. Choose unvisited vertex $v \in SV$ and mark it visited.
3. Perform depth first search from v on *SCG*. Group all vertices traversed in the search and mark them visited.
4. Remove visited vertices and their edges from *SG*.
5. Repeat steps 2–4 until all vertices in *SCG* are visited.

and 7, as use-case 7 is considered to be critical. The set of use-cases that need to have the same NoC configuration have an edge between them in the *SCG* graph.

To find the set of all use-cases that need to have the same NoC configuration, we use the algorithm presented in Algorithm 4. In the algorithm, the *SCG* graph is traversed and those vertices that are reachable from each other are grouped. The vertices in the same group represent those use-cases that need to have the same NoC configuration. This is obtained by performing depth-first search of *SCG*, possibly multiple times, until all vertices are traversed. The set of vertices traversed in a single search are grouped together, as they are reachable from each other. During the mapping process, the set of use-cases that are in the same group utilize the same NoC configuration.

A similar approach is used for finding the set of all use-cases that are in the same DVS/DFS group and need to have the same operating voltage and frequency.

6.4 Unified Mapping–NoC Configuration

In the next steps, we must map the IP cores to NIs and generate configurations for the NIs which support the various use-cases. To perform the mapping, we formulate the following definitions.

Definition 13 Let the set of use-cases be U. The communication between set of all pairs of cores in an use-case $i, \forall i \in 1, \ldots, |U|$, is represented by the set F_i. Each flow in the use-case i, $flow_{i,j}, \forall j \in 1, \ldots, |F_i|$, is associated with a *bandwidth* requirement, $comm_{i,j}$ and a *latency constraint*, $lat_{i,j}$. Let the source core of the traffic flow $flow_{i,j}$ be $source(flow_{i,j})$ and the destination core of the flow be $dest(flow_{i,j})$.

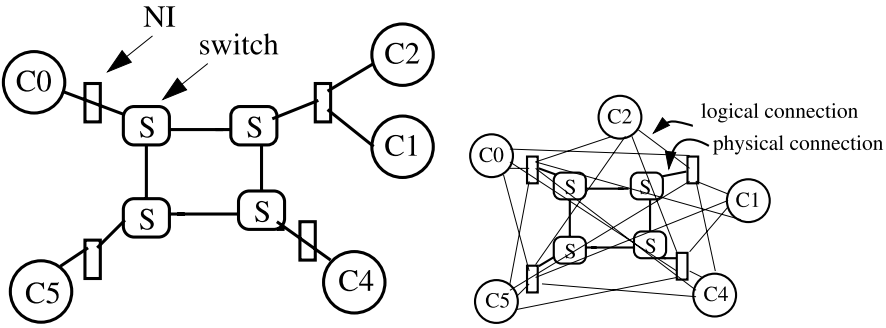
The bandwidth of the flow represents the maximum rate of traffic communicated in the flow and the latency of the flow represents the maximum delay by which a transaction of the flow should reach the destination.

We define the NoC topology by the *topology graph*.

Definition 14 The NoC topology graph is a directed graph $P(V, L)$, with each vertex $v_i \in V$ representing a core or core-cluster,[2] NI or a switch in the design and the directed edge (v_i, v_j), denoted as $l_{i,j} \in L$, representing a link between vertices v_i and v_j.

As an example, a NoC topology graph with 5 cores, 4 NIs and 4 switches is shown in Figure 6.6(a). As the connectivity of the cores with the switches and NIs, which defines a mapping of the cores onto the network components, is an output of the design methodology; the complete connectivity information of the topology graph is only obtained after the application of the algorithms.

To facilitate the mapping process, taking into account the possibly different paths and TDMA slot reservations to be used by the different use-cases, we define a *virtual topology graph* for each use-case. The *virtual topology graph* keeps track of the current path and slot-allocation per use-case while the algorithm is running.



(a) Topology graph (b) *VTG*

Fig. 6.6 (**a**), (**b**): Example NoC topology graph and *VTG* for a use-case. In the *VTG*, the connections between the switches and the NIs represent actual physical connections, while between the cores and the NIs represent logical connectivity, signifying that any core can be connected to any NI. The actual physical connections between the cores and the NIs are obtained as outputs of the proposed mapping procedure

[2]The procedure can also consider a cluster of cores that are to be mapped to the same NI.

Definition 15 The *virtual topology graph* for use-case $i, \forall i \in 1, \ldots, |U|$ (represented as $VTG_i(W, M)$) is a directed graph with the same number of vertices as the topology graph $P(V, L)$. Each edge, $m_{i,i1,j1}$, $\forall i1, \ j1 \in 1, \ldots, |M|$, in the graph represents the possibility of an actual physical link. Each vertex representing a core or a core-cluster is connected to all the vertices that represents NIs, which means initially each core could potentially be connected to any NI. The connectivity between the switches and the NIs are determined by the physical switch network architecture, which is an input to the design methodology. Each edge $m \in M$ is associated with the residual bandwidth capacity value, $comm_m$ (the amount of bandwidth that is not yet reserved) and a TDMA time-slot table t_m. The source vertex of the edge m is represented by $s(m)$ and the destination vertex of the edge is represented by $d(m)$.

As an example, the *VTG* for a use-case, for the example presented in Figure 6.6(a), is shown in Figure 6.6(b). Please note that an edge between the vertex representing a core and the vertex representing a NI in the *VTG* (as an example, the edge denoted as a *logical connection* in the figure) does not imply an actual physical link. The actual physical connections are established by the mapping procedure. In this work, we assume the connectivity between the switches to be based on a mesh topology, and the procedure can be easily extended to accommodate other topologies as well. The connectivity between the NIs and the switches is determined by the amount of NI to switch clustering that is permitted, which is obtained as an user input. In the example from Figure 6.6(b), two NIs are connected to each switch and a 2×2 mesh topology is used to interconnect the 4 switches.

Definition 16 A path $\pi_{i,i1,j1}$ for a traffic flow from a source vertex $w_{i,i1}$ to destination vertex $w_{i,j1}$ in the graph VTG_i is a nonempty sequence of edges $\langle m_1, m_2, \ldots, m_k \rangle$, such that

- $d(m_{k1}) = s(m_{k1+1}), \forall k1 \in 1, \ldots, k-1$.
- $s(m_1) = w_{i,i1}$ and $d(m_k) = w_{i,j1}$.

The cost of traversing an edge of the path is determined by the contention on the edge (based on the residual bandwidth and slot-table availability) and the total cost of a path is determined by the weighted sum of contention on the edges of the path and hop-count.

In the mapping procedure, we select a path with a low contention (high probability of successful allocation) and at the same time try to keep the path length short, so that we do not consume unnecessarily many resources. As the cost function of a path depends on two factors (contention and hop-count), we use a weighted linear combination of the two measures, with the weights for the measures set up experimentally.

The mapping algorithm for multiple use-cases is presented in Algorithm 5. In the first step, the maximum NoC operating frequency, the data-width of the links, the maximum TDMA slot-table size and the number of NIs connected to each switch are obtained as user inputs. From the data-width and the maximum frequency design point, the maximum bandwidth available on each link of the NoC is obtained

Algorithm 5 Unified Mapping and NoC Configuration

1: Obtain the NoC operating frequency, link data-width, maximum TDMA slot-table size, and number of NIs connected to each switch.
2: Set the maximum bandwidth available across each NoC link (*max_bw*) as a product of the NoC frequency and link data-width.
3: First, increase the number of slots until the predetermined maximum with a valid mapping is obtained. If the maximum slot-table size is reached without finding a valid mapping, increase the number of switches in the design until a valid mapping is obtained, or until an user-defined threshold is reached.
4: Generate, $VTG_i(W, M)$ for each use-case i, $\forall i \in 1, \ldots, |U|$.
5: Assign the residual bandwidth value of all edges, bw_m $\forall m \in M$, to *max_bw*.
6: Sort the flows $f_{i,j}$, $\forall i \in 1, \ldots, |U|$, $j \in 1, \ldots, |F_i|$, in nonincreasing order of the bandwidth values, $comm_{i,j}$.
7: Choose the flow in order of the bandwidth value, preferring flows that have source/destination vertices already mapped. Let $f_{u1,n}$ be the flow chosen.
8: Let $w_{u1,i1}$ be the vertex representing the core $source(f_{u1,n})$ and $w_{u1,j1}$ be the vertex representing the core $dest(f_{u1,n})$.
9: Find constrained least cost path $\pi_{i,i1,j1}$ from $source(f_{u1,n})$ to $dest(f_{u1,n})$ such that
10: (i) $\text{Min}_{\forall m_{k1} \in \pi_{i,i1,j1}} bw_{m_k1} \geq comm_{u1,n}$ (each link has enough bandwidth to accomodate the flow)
11: (ii) TDMA slots are available in m_{k1}, $\forall m_{k1} \in \pi_{i,i1,j1}$.
12: (iii) The latency (hop-delay) constraint of the flow is satisfied.
13: (iv) And the cost of the path is minimum.
14: Reduce the residual bandwidth available (bw_{m_k1}) on m_{k1}, $\forall m_{k1} \in \pi_{i,i1,j1}$ by $comm_{u1,n}$ and re-compute available TDMA time-slots.
15: Remove edges other than the one present in the chosen path from source/destination vertices to NIs. Apply this process to all the use-cases.
16: For all other use-cases i, $\forall i \in 1, \ldots, |U|$, $i \notin u1$, choose the flow f, that has the same source and destination vertices as $f_{u1,n}$, if such a flow exists.
17: Choose a least cost path in each use-case that satisfies the constraints and reserve resources. For use-cases in same group, choose path for that use-case in the group that has the maximum bandwidth value for the flow and reserve resources across the path in each use-case.
18: Remove mapped flows and repeat steps 7–17 until all flows are mapped.
19: If no path that satisfies the constraints is available for any flow of any of the use-cases, go to step 3.
20: Once a valid mapping is obtained, obtain the required frequency of operation and supply voltage for all the use-cases in each DVS/DFS cluster.
21: Store the paths, slot-table allocations, supply voltage, and operating frequency for the use-cases.

(step 2). In the next step (step 3), the number of slots and switches in the design are increased until a valid mapping is obtained in the subsequent steps. The objective of

the algorithm is to find the smallest NoC design (in terms of the number of switches and NIs utilized) that satisfies the bandwidth and latency constraints of all the use-cases. The algorithm declares the mapping as infeasible when the switch/NI count reaches an user-defined threshold.

For a chosen switch/NI count, the *VTG* graphs for all the use-cases are constructed (step 4). Then the residual bandwidth on all the edges of the *VTG* graphs are assigned to the maximum bandwidth value. In the next step (step 6), the traffic flows are sorted in a non-increasing order of their bandwidth values for all the use-cases in the design. Then (step 7) the flow with the maximum bandwidth value is chosen across all the use-cases. The intuition behind choosing the flow that has the largest bandwidth value first is that it reduces bandwidth fragmentation and larger flows get to use shorter paths, which is desirable as it leads to lower power consumption [46]. While choosing a flow, we prefer to choose a flow from the already mapped nodes before other flows, as it further helps in satisfying the bandwidth constraints.

For the chosen flow, the least cost path that satisfies the bandwidth, TDMA slot-table and latency constraints is obtained (steps 8–13). For obtaining the least-cost path, we use Dijkstra's algorithm that is modified to accommodate the constraints, applied onto the *VTG* representing the use-case that has the chosen traffic flow. Once the least cost path is computed, the residual bandwidth and the TDMA time slots available on the edges of the path are recomputed (step 14). In the next step, all edges from the vertex representing the source/destination cores to the vertices representing the NIs, other than the one in the chosen path, are removed. This is because, in the Æthereal architecture, each core is connected to only one NI (while a single NI can connect multiple cores) due to physical layout restrictions. Now, the edge of the chosen path has established an actual physical connection between the source/destination cores and the NI.

As all the use-cases use the same mapping of cores onto the NIs, we fix the physical connection between the source/destination cores and the NIs in all the other use-cases as well (step 15). Then (step 16) for all the other use-cases, the flows that have the same source and destination cores as the one that is mapped are chosen and allocated in a similar manner (as done in steps 8–13). For the flows belonging to the use-cases from the same configuration group, the flow with the maximum bandwidth value is chosen first and the path and slot-table reservations for the flow are obtained. Then for all the other use-cases in the group, the same path, and slot-table reservations are utilized for the corresponding flows.

The process is repeated until all flows in all the use-cases are mapped in the NoC (step 18). At any stage, if a flow cannot be mapped (as a path that satisfies the constraints of the flow is not available), the entire mapping process is repeated using a bigger NoC design.

Once a valid mapping is obtained, the required operating frequency for the use-cases in each DVS/DFS group is obtained (step 20). The required operating frequency is computed from the maximum bandwidth requirements of any link across all the use-cases in a DVS/DFS group, as all the use-cases in a group use the same operating frequency and voltage. From the operating frequency, the required supply voltage is also obtained. The paths, slot-table allocations, operating frequency,

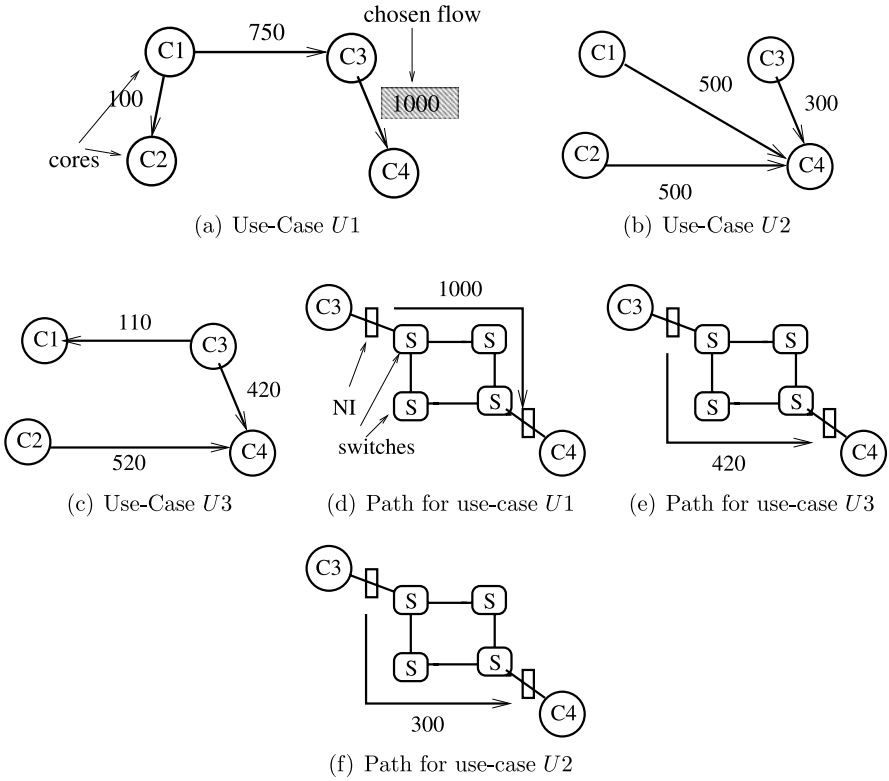


Fig. 6.7 (**a**), (**b**), (**c**): Example use-cases with traffic flows annotated with bandwidth values (in MB/s). (**d**), (**e**), (**f**): The paths chosen for the three use-cases for a flow between cores C3 and C4

and supply voltage for the use-cases, along with the designed NoC are given as the outputs of the procedure.

Example 5 Let us consider a small example of the procedure for 3 use-cases shown in Figure 6.7. Let us assume that use-cases U2 and U3 are in the same configuration and DVS/DFS group, and hence should utilize the same path and slot-table allocations. Let us also assume that the use-case U1 is in a different configuration and DVS/DFS group. The largest flow across the 3 use-cases is the flow between the cores C3 and C4 in U1. A mapping of the cores C3, C4 onto the NoC topology, along with unified path selection and TDMA slot table reservation for the first use-case is performed (Figure 6.7(d)). The flow between C3 and C4 in the other two use-cases are selected next. As U2 and U3 are assumed to be in the same configuration group, the flow should use the same path and slot-table reservations in both the use-cases. As the flow from C3 to C4 in U3 is larger than the one in U2, the paths and slot-table reservations for the flow are obtained in U3 (refer to Figure 6.7(e)). Then the same allocations are used for the flow in U2 (refer to Figure 6.7(f)). Note that all the use-

cases use the same mapping of the cores onto the topology, but can use a different path if NoC reconfiguration is possible when the two use-cases switch. The residual capacity and time slots on the NoC links are updated separately for the use-cases. The process is repeated for all the remaining flows in the use-cases. Finally, after routing all the flows, the required NoC frequency and supply voltage are obtained for each DVS/DFS group. As an example, let us assume that the NoC link data-width is 32-bits. Then after routing the flow from C3 and C4, the minimum NoC frequency required for use-case U1 is computed to be 250 MHz. This is because the bandwidth of a link is the product of frequency (250 MHz) and link data-width (4 bytes), which would match the maximum traffic rate for the mapped flow in U1 (1000 MB/s). For the DVS/DFS group consisting of use-cases U2 and U3, the NoC frequency requirement for the currently mapped flows is 105 MHz (the maximum requirement across the 2 use-cases).

6.5 Simulation Results

6.5.1 Experimental Benchmarks

To validate the performance of the multiple use-case mapping methodology, we perform experiments on existing SoC designs and synthetic benchmarks. We consider four simplified versions of real SoC designs: a set-top box SoC with 4 use-cases (D1), set-top box SoC with 20 use-cases (D2), a video processing SoC used in TVs with 8-use-cases (D3), and video processing SoC with 20-use-cases (D4). The designs D2 and D4 are based on scaled versions of the designs D1 and D3 for supporting more use-cases. Each use-case has a large number of (50 to 150) communicating pairs of components. The set-top box SoC and the TV processor have different functionalities and communication patterns. The set-top box design uses an external memory for storing and retrieving data and the amount of data communicated to the memory is very large when compared to the rest of the design. The video processor design uses a streaming architecture with local memories on the chip, thereby distributing the communication load across several components. We apply the proposed design method to these SoCs with different architectures to validate the generality of the method.

We also generated synthetic benchmarks for testing the method with more number and variety of use-cases. The benchmarks are structured to follow the application patterns of real SoCs. We identify two classes of such benchmarks: (i) Spread communication benchmarks (*Sp*), where each core communicates to few other cores. These benchmarks represent designs such as the TV processor that has many small local memories with communication spread evenly in the design. (ii) Bottleneck communication benchmarks (*Bot*), where there are one or more bottleneck vertices to which most of the communication takes place. These benchmarks characterize designs using shared memory/external devices such as the set-top box example. We vary the bandwidth and latency constraints across the different traffic flows of the

use-cases. Most of the video processing architectures have traffic flows that have bandwidth/latency values that fall in to few (around 3–4) clusters. As an example, the HD video streams have traffic flows with bandwidth requirements of few hundred MB/s, the SD video streams have few MB/S bandwidth needs, the audio streams have low bandwidth needs and the control streams have low bandwidth needs, but are latency critical. We capture such effects in the synthetic benchmarks generated, with the traffic parameters taking a cluster of values, with small deviations in the values within each cluster.

6.5.2 Effect of Mapping for SoC Benchmarks

In order to compare the quality of mappings produced by the design approach presented in this work with the worst-case design method (*WC* method) presented in [54], we fix the operating frequency and link sizes of the NoC to be the same (500 MHz, 32 bits) for the methods. We apply the design methods and find the smallest size network that satisfies the constraints of the use-cases. We fix the number of cores to be same (equal to 20 with 60–100 connections between cores) for all the synthetic benchmarks and vary the number of use-cases across the benchmarks (from 2 to 40 use-cases) to evaluate the quality of the mappings. In Figure 6.8, the number of switches used in the mesh NoC for the current design methodology normalized with respect to the number of switches used in the *WC* method for the various benchmarks is presented. For the designs D1, D2 and for the synthetic benchmarks with small number of use-cases, the *WC* method performs reasonably when compared to the method presented in this work. However, as the number of use-cases increase, the *WC* method starts to perform poorly, as the worst-case use-case becomes heavily over-specified and the resulting NoC design becomes big. The method presented here, on the other hand, performs well even for large number of use-cases and is scalable. As an example, for the D3 design, the current methodology produced a successful mapping of the application onto a 2×2 mesh, while the *WC* method required a 11×11 mesh for the design. For the synthetic benchmarks (both *Sp* and *Bot*) with 40 use-cases, the current methodology resulted in a 2×2 mesh, while the *WC* method failed to produce a valid mapping even onto a 20×20 mesh topology (thus they did not plot in Figures 6.8(b) and (c)). Compared to the *Bot* benchmarks, for the *Sp* benchmarks, the current method performs much better than the *WC*. This is attributed to the fact that the *Sp* benchmarks have more variations in the communication patterns across the different use-cases and the *WC* method is unable to adapt to such variations, while the current method does. For all the benchmarks, both the methods produced the results in less than a few minutes when run on a Linux workstation.

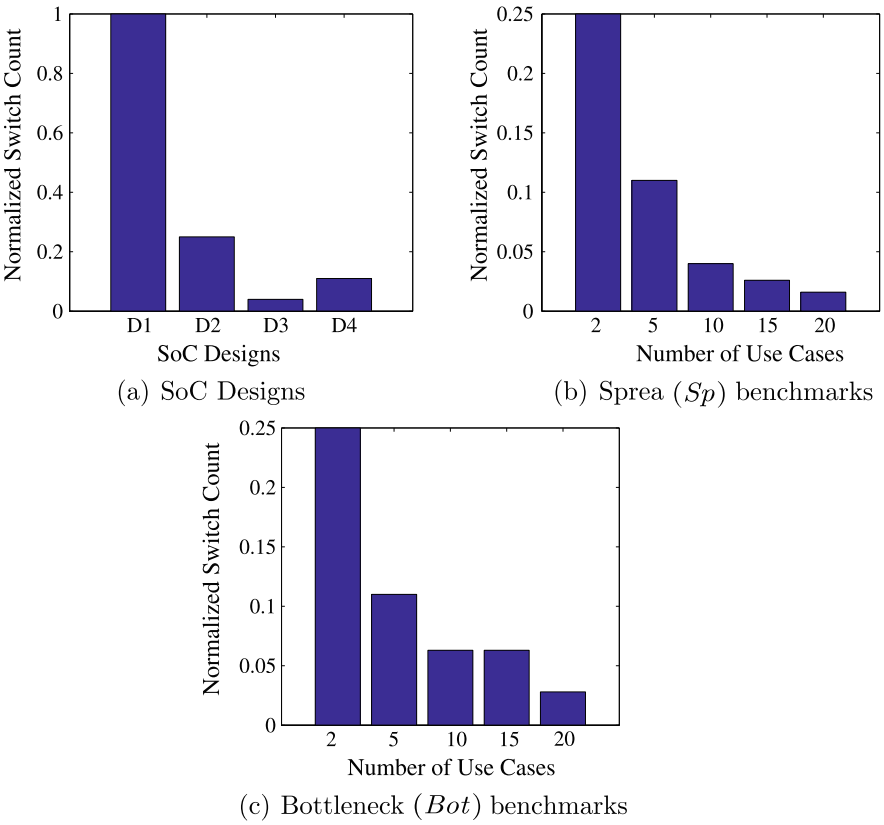


Fig. 6.8 The number of switches used for the current method normalized with respect to the *WC* method

6.5.3 Frequency-Area Trade-offs

We can perform area-frequency trade-offs using the method presented in this work. When the NoC frequency is higher, the bandwidth and resources available across the NoC is higher and a smaller network can satisfy the constraints of the design. On the other hand, higher frequency of operation implies a higher power consumption in the network. In Figure 6.9(a), we present the Pareto curve for the area-frequency trade-off for the D1 design. The area of the switches is obtained from layouts with back-annotated worst-case timing in 0.13 μm technology. At low operating frequencies ($\leq$350 MHz), the area of the NoC (which is taken to be the sum of the area of all the switches)[3] is large as an increased number of switches are needed to satisfy the design constraints. At very high-frequencies ($\geq$1.5 GHz), the area of the NoC is

[3]Here, we assume that the NI area is taken to be part of the core area.

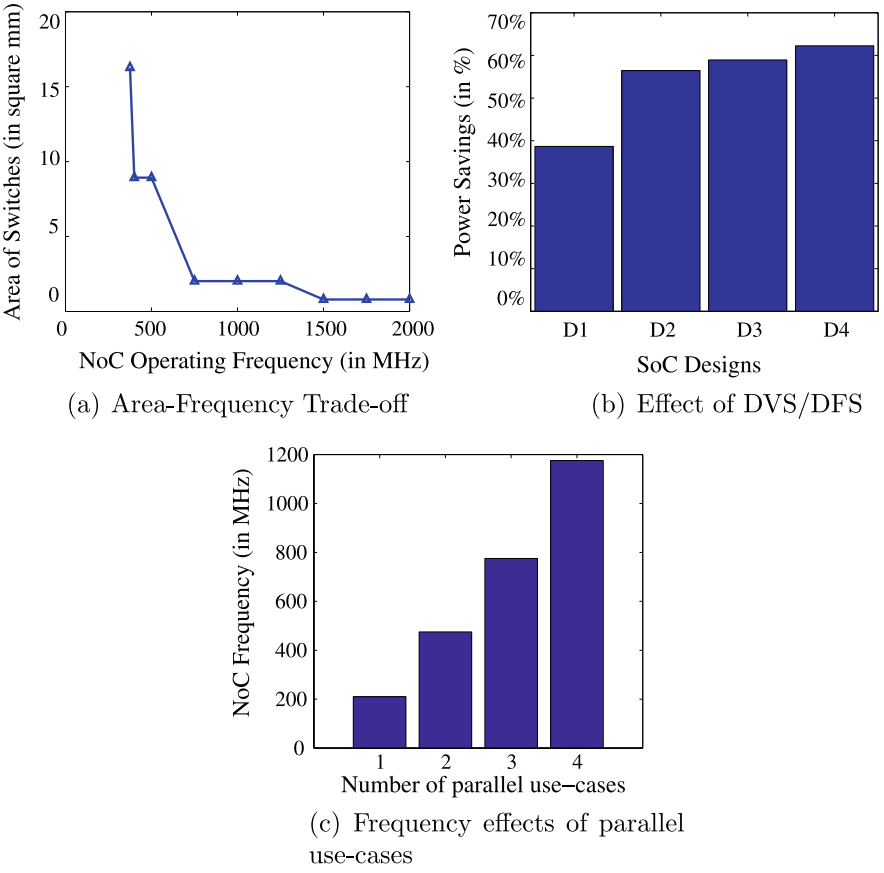


Fig. 6.9 (**a**) Area-Frequency trade-offs, (**b**) The power savings achieved using DVS/DFS, and (**c**) The impact of running use-cases in parallel

very small. The optimum design point can be chosen based on the objectives of the designer from such a curve.

6.5.4 Dynamic Configuration

The switching time between most use-cases in a SoC is of the order of a few milliseconds. When the use-cases are expected to run for a long time, the frequency of operation of the NoC can be varied during this switching time to match the communication characteristics of the use-cases, thereby resulting in large power savings for the system. When the different use-cases require different NoC frequencies, the voltage of the NoC can also be dynamically changed to match the requirements of the use-cases. We use a conservative model for voltage scaling, where we assume

that the square of the voltage scales linearly with the frequency [97]. The dynamic voltage and frequency scaling technique (DVS/DFS) results in an average of 54% reduction in power consumption for the different SoC designs when compared to the design where no DVS/DFS scheme is used (Figure 6.9(b)).

6.5.5 Parallel Use-Cases

As the number of use-cases that can run in parallel increases, the NoC size or frequency also increases. The methodology can be applied by the designer to quickly perform trade-offs involving the number of use-cases that run in parallel with the size/frequency required for the NoC to support the parallel use-cases. As an example for a 20-core, 10 use-case *Sp* benchmark, the required NoC frequency as the number of use-cases run in parallel is varied is presented in Figure 6.9(c). Such a plot helps the designer in evaluating the trade-offs involved in the NoC for supporting multiple parallel use-cases.

6.6 Summary

As the number of applications or use-cases integrated onto a single SoC increases, the designer is faced with the challenge of building an interconnect structure that supports the design constraints of all the use-cases. In this chapter, we motivated the importance of the problem and presented extensions of the design methods presented in the previous chapters, to handle the multiuse case scenario. We also presented a way to dynamically configure the interconnect to support multiple use-cases and integrated Dynamic Voltage and Frequency (DVS/DFS) techniques with the reconfiguration mechanism.

Chapter 7
Supporting Dynamic Application Patterns

To efficiently utilize the large number of transistors that are available on the chip with manageable design complexity and wiring requirements, *Chip Multiprocessors* (*CMPs*) have been recently proposed [100–103]. In CMPs, the chip area is divided into a number of regular and identical tiles, where each tile represents a processor/memory core. The use of a simpler architecture for the processor in a single tile, coupled together with the reuse of the tile across the chip, results in a reduced design complexity, when compared to conventional single-core processor systems.

7.1 NoC Design Challenges for CMPs

The systems that utilize NoCs can be broadly classified into two types: *Application-Specific Systems-on-Chip* (*ASSoCs*) and CMPs. In ASSoCs, single or a fixed set of applications are statically mapped onto the different processor and hardware cores in the design. The communication between the various cores is known and the interconnect architecture can be tailor-made to suit the application traffic characteristics. In all the preceding chapters, we targeted the design of such ASSoCs. On the other hand, in CMPs, general-purpose processor cores are used to run software tasks of different applications (an example shown in Figure 7.1). In such systems, the communication between the cores cannot be precharacterized, as the different application processes can be mapped differently to the cores, typically with the support of the compiler [100]. As the total system performance of CMPs is increasingly dominated by the interconnect performance [100], designing an interconnect architecture with predictable performance is critical.

In NoC-based systems for CMPs, to provide predictable performance and optimal network throughput, the bandwidth capacity of the different links of the NoC should be sufficient to support the peak rate of traffic on the links. If the network links cannot support the peak traffic that can be routed on them, then the network might experience traffic congestion. In a congested network, the latency for the traffic streams, and hence the interconnect performance will become unpredictable, which needs to be avoided for dependable system operation.

In traditional multiprocessor interconnection networks (the chip-to-chip networks), the bandwidth on the network links is limited by the number of pins that are available on the chip and all the links of the network have the same bandwidth capacity [107]. For most interconnect topologies and routing patterns, the load on the different links of the network is nonuniform. Thus, in traditional multiprocessor networks, the interconnect throughput is limited by the bottleneck links of the network [107].

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

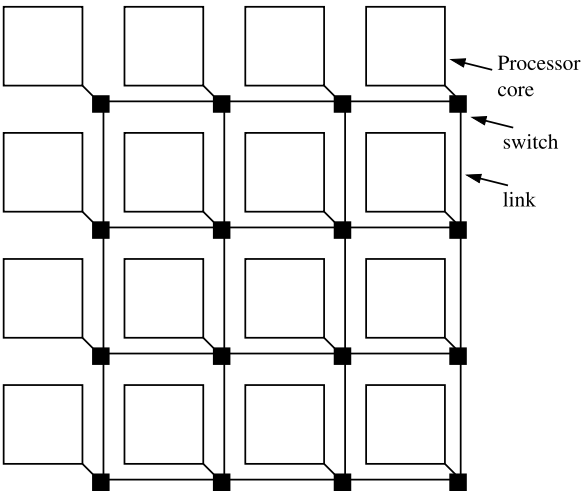


Fig. 7.1 Example tile-based CMP architecture

On the other hand, CMPs have enormous wiring resources at their disposal [22]. The links in different parts of the network can be sized differently, so that the network throughput is no longer limited by few bottleneck links. As chip designs are increasingly power consumption limited, when sizing the links, it is important to achieve a NoC design with the least power consumption.

However, in order to design such a network, there are several challenges that have to be addressed:

- The first challenge is that the exact traffic pattern of the CMP cannot be precharacterized. Usually, to evaluate the quality of the interconnection network in multiprocessors, the network is simulated with different traffic patterns, such as uniform, nearest neighbor, hot-spot, etc. If such a template of traffic patterns is used to size the links of the NoC, there is a huge drawback that the methodology is ad hoc and does not guarantee network throughput for other traffic patterns that can occur when real applications are executed.
- The second challenge is to efficiently utilize the link bandwidth (which is a product of link width and frequency) available. Traditionally, links with different bandwidth capacities are obtained by varying either their frequency of operation or their width. However, both schemes require complex frequency and width converters for potentially every input of every switch in the design. This drastically increases the design complexity and NoC area. Moreover, such designs incur significant serialization and parallelization delay at every switch, which results in high packet latencies. Thus, a way to efficiently utilize the link bandwidth is needed.
- Finally, the interconnect has to maintain a regular structure, so that a predictable and modular architecture is obtained.

In this chapter, we address the important problem of synthesizing the most power efficient NoC for CMPs that have dynamic traffic patterns, providing theoretically

guaranteed optimum throughput and predictable performance for any application to be executed on the CMP.

We achieve a predictable interconnect design in two ways: First, the architecture is designed to provide predictable performance under all application traffic conditions. Second, the synthesis approach considers accurate information of the physical layer measures (such as wire-lengths, wire delays, network component delays), thereby bridging the gap between the synthesis models and the actual physical layout implementation. Thus, the design process becomes more predictable, leading to quicker design convergence.

7.2 Basics of the Synthesis Approach

To efficiently utilize the large on-chip wiring resources that are available, we use multiple physical channels for each link, namely, a link is segmented into different physical channels that can be utilized by different traffic flows in parallel. As an example, a 2×3 mesh topology is presented in Figure 7.2. Each vertex in the figure represents a switch (and the core that is connected to the switch) and a link between two vertices has one or more physical channels. For example, the link from vertex v_1 to vertex v_3 has two physical channels, while the link from vertex v_0 to vertex v_1 has one physical channel. In the synthesis process, we size the different links with different number of physical channels, such that each channel supports the load due to any traffic pattern of the NoC.

When multiple physical channels are used between two switches, if different channels are dynamically assigned to incoming packets, it may lead to out-of-delivery of packets. In this case, reorder buffers are required for ordering the packets at each receiver. Such buffers have large power and area overhead and deterministically sizing them is infeasible in practice [43]. To avoid such out-of-order delivery, for the traffic flow from each source to destination, we statically assign a single channel in every link that is used by the flow. We integrate this mapping of traffic flows to the different channels in the synthesis procedure.

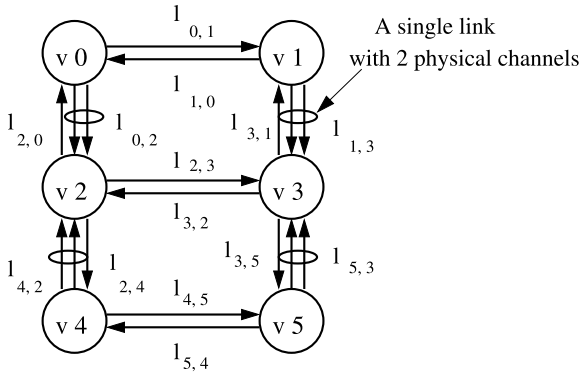


Fig. 7.2 Example 2×3 mesh topology

We also tune the setting up of NoC operating frequency during the synthesis process. To evaluate the quality of the different NoC designs, we use accurate analytical models for power consumption of the network components. The power consumption values are obtained from layouts with back-annotated resistance and capacitance information at 0.13 µm technology using standard industrial tools.

During the synthesis of the NoC, we consider the physical layer measures as well: the delay encountered on the wires in the NoC and the target frequency that can be supported by the designed network components. The synthesis approach utilizes the floorplan knowledge of the NoC to detect timing violations on the NoC links early in the design cycle. This results in a faster design cycle that leads to a reduction in the number of design re-spins and faster time-to-market, which are critical for today's complex chips. We validate the design flow predictability of the proposed approach by performing a layout of the NoC synthesized for a 25-core CMP. The approach maintains the regular and predictable structure of the NoC and is applicable in practice to existing NoC architectures.

7.3 Design Flow

In this section, we present the synthesis flow used to design the NoC (see Figure 7.3). The network topology, utilized routing function, operating frequency of the core, core data width and network link width are inputs from the user. In the

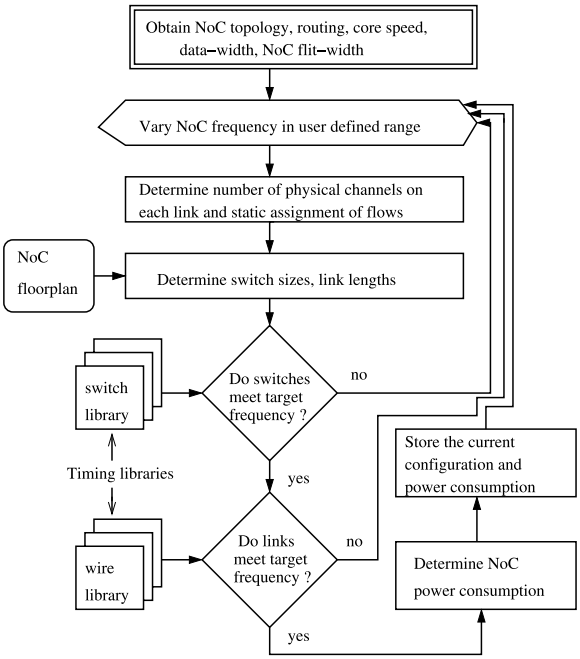


Fig. 7.3 NoC synthesis design flow

outer loop of the synthesis process, the operating frequency of the NoC is varied in a user-defined range. For each frequency point, the number of physical channels on each link and an assignment of traffic flows to the different physical channels are computed by the synthesis process.

From the number of physical channels instantiated between the switches, the different switch sizes are obtained. Then we evaluate whether every switch of the NoC can support the corresponding frequency point (chosen in the outer loop). As the switch size increases, the maximum frequency of operation it can support reduces (as the critical path inside the switch gets longer) [34]. This information is obtained from the layout of the switches for different sizes, which is taken as an input library for the synthesis method. Then all the links in the NoC are checked for timing delay violations. For evaluating the wiring delays, we include the floorplan of the NoC as an input to the synthesis flow. Usually, standard topologies, such as mesh, are used for CMPs because the floorplan of the NoC is regular and known at design time. Based on the link lengths and wire models from [58], the delay values on the NoC links are calculated. Any timing violations on the NoC links are then evaluated by the method. If the design satisfies the timing constraints on NoC switches and links, then the power consumption of the NoC is computed, based on the layout-level power models. From the set of all feasible NoC designs, the design with the least power consumption is finally chosen by the synthesis process.

7.4 Problem Formulation

The topology of the network that defines the connectivity between the switches and the cores is taken as input. The number of physical channels used for each link is to be determined by the synthesis procedure. Formally, the NoC topology is defined by the *topology graph*:

Definition 17 The NoC topology graph is a directed graph $P(V, L)$, with each vertex $v_i \in V$ representing a core (and the switch to which it is connected) and the directed edge (v_i, v_j), denoted as $l_{i,j} \in L$, representing a link between vertices v_i and v_j. The set of physical channels that are instantiated for each link $l_{i,j}$, is represented by the set $CH_{i,j}$.

An example topology graph was presented earlier in Figure 7.2, which represents a 2×3 mesh network. The graph has 6 vertices (v_0 through v_5) and 14 links ($l_{0,1}, \ldots, l_{5,4}$). The number of physical channels used in each link varies. For example, link $l_{0,2}$ has 2 physical channels. Please note that this number is an output of the synthesis process. To begin with (when the inputs are fed), all the links are initialized to have no physical channels. Then the communication among NoC nodes can be defined as follows.

Definition 18 The communication between each pair of cores is treated as a flow of single commodity, represented as d_k, $k = 1, 2, \ldots, |V| \times |V|$, with the source of the commodity represented as $source(d_k)$ and the destination represented as $dest(d_k)$.[1]

We assume that a deterministic routing function is utilized for routing packets, as most existing NoC architectures support only a deterministic routing function [30, 33]. This is because the area-power overhead involved in adaptive routing is quite high. Moreover, adaptive routing presents several problems such as out-of-order packet delivery, which are hard to tackle in on-chip networks that need to have low power overhead. The routing function defines the set of links used by each commodity as follows.

Definition 19 The routing function R maps the traffic flows of commodities onto the links of the network, i.e., $R : d_k \rightarrow L$, $\forall k$. The set of links utilized by the commodity k for the routing function is represented by the set L_k.

In Figure 7.2, links $l_{1,0}$ and $l_{0,2}$ are used by the traffic flow that has vertex v_1 and source and vertex v_2 as destination, for the dimension-ordered (with x first, y next) routing scheme.

The maximum rate at which each core injects traffic into the network is also taken as an input to the synthesis engine. It is defined formally as follows.

Definition 20 The rate of traffic injection of each core, v_i, $\forall i$, is represented by r_i. The rate of each commodity d_k, represented as $rate(d_k)$, is equal to the rate of traffic injection of the source core of that commodity, i.e., $r_{source(d_k)}$.

Practically, for most CMPs, each core can inject one data word into the network every clock cycle. Thus, the injection rate is the product of the operating frequency of the core and its data width. For instance, if a core has a data width of 32 bits and operates at 100 MHz, its injection rate is 400 MB/s (i.e., 4 B $\times$ 100 MHz).

We also obtain as inputs the set of interesting operating frequencies to explore for the NoC design, and the data width of the channels (which is usually set to match the data width of the cores).

Then the *Problem Statement* is the following:

The synthesis procedure has to determine the number of channels ($|CH_{i,j}|$) required for each link ($l_{i,j}$) and a static mapping of each commodity (d_k) onto a single channel ($ch \in CH_{i,j}$) of each link $l_{i,j} \in L_k$. The mapping has to satisfy the constraint that every channel should support the traffic rates of all the commodities mapped onto that channel for any traffic pattern. The synthesis process should also determine the NoC operating frequency that results in the most power efficient NoC design.

[1] In the rest of this chapter, we follow the convention that variables i, j are defined for $1, \ldots, |V|$ and the variable k is defined for $1, \ldots, |V| \times |V|$.

An optimum (100%) throughput can be achieved if each channel supports its worst-case load, i.e., the channel bandwidth matches or exceeds the channel load. Here, we would like to point out that to practically achieve the full throughput value, the NoC architecture should have a predictable communication behavior, as in [30, 104, 105].

7.5 Synthesis Algorithm

The detailed synthesis algorithm to solve the defined problem is presented in Algorithm 6. In step 1, the NoC frequency of operation is varied in user-defined steps.

Algorithm 6 Synthesis Algorithm

1: **for** Each NoC frequency (*freq*) design point in user defined range **do**
2: **for** each link $l_{i,j} \in L$ **do**
3: Build the Link Loading Graph ($LLG_{i,j}$) for the link $l_{i,j}$
4: Build the Vertex Conflict Graph ($VCG_{i,j}$) for the link $l_{i,j}$
5: Initialize number of channels to zero, $m = 0$
6: Increment m by 1 and instantiate new physical channel ch_m
7: Find m *max-cut* partitions of *VCG*
8: Assign *bw_satisfied* to true
9: **for** each max-cut partition **do**
10: Build Partition Loading Graph (*PLG*)
11: max_load = maximum_weight_matching(*PLG*)
12: If ($max\_load > freq \times width$), $bw\_satisfied$ = false
13: **end for**
14: **if** *bw_satisfied* **then**
15: Assign those commodities k such that $source(d_k)$ is in partition $m1$ to channel $m1$, $\forall m1 \in 1, \ldots, m$
16: Set $CH_{i,j} = \bigcup_{\forall m1 \in m} ch_{m1}$
17: **else**
18: Go to step 6
19: **end if**
20: **end for**
21: From computed $CH_{i,j}$, $\forall i$, and j, compute the switch sizes
22: Evaluate whether the switch size implementations can match the target frequency (*freq*)
23: Evaluate whether all the links in the NoC can meet the target frequency (*freq*). Utilize the NoC floorplan information to estimate the link lengths
24: If target frequency met, obtain the power consumption for the synthesized NoC
25: **end for**
26: From the set of synthesized NoCs, choose the design with least power consumption

Then in step 2, we consider each link individually to size the different links with different channels.

7.5.1 NoC Link Sizing

For each link $l_{i,j}$, we first build a *Link Loading Graph* (*LLG*) (in step 3), defined as follows.

Definition 21 The $LLG_{i,j}(LV, LL)$ is a bipartite graph with $|LV| = 2 \times |V|$ (i.e. with $2 \times |V|$ vertices). An edge exists between vertices lv_x and lv_y, $\forall x \in 1, \ldots, |V|$, $\forall y \in |V| + 1 \cdots 2 \times |V|$, if $\exists k$ such that $source(d_k) = lv_x$ and $dest(d_k) = lv_{y-|V|}$ and $l_{i,j} \in L_k$. The weight of the edge is the rate of traffic flow of the commodity, i.e., equal to $rate(d_k)$.

The edges of the *LLG* represent the set of all traffic flows that utilize the link, depending on whether the link is part of the route for the different traffic flows. The weights of the edges represent the rate of the traffic flows.

Example 6 The *LLG* for the link $l_{0,2}$ (i.e., $LLG_{0,2}$) of the 2×3 mesh example is presented in Figure 7.4. With x–y routing, the link $l_{0,2}$ is used by the traffic flows that originate from vertex v_0 to v_2 and v_4, and by the traffic flows that originate from vertex v_1 to v_2 and v_4. The maximum rate of all these traffic flows is 400 MB/s. In the *LLG* bipartite-graph, each vertex on both the left and the right columns represents a single core. Those vertices with traffic flows that utilize this particular link have edges between them. In this example, there are edges between those vertices that represent *core*_0 and *core*_1 with *core*_2 and *core*_4, with the edge weights being the rate of the flows.

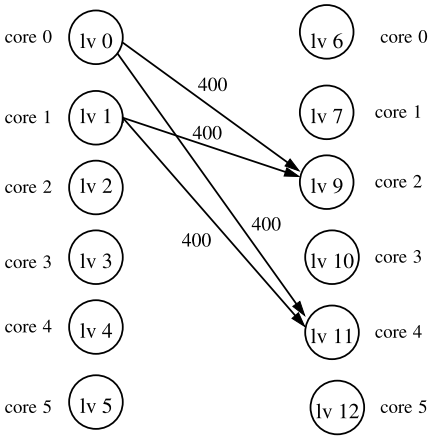


Fig. 7.4 Link loading graph. The edges are annotated with weights in MB/s

The load on a link is equal to the sum of the loads caused by each source-destination pair using that link. The worst-case link load can be obtained by considering all possible permutation traffic patterns. In [108] and [109], the authors show that the worst-case load can be obtained by representing all permutations as matchings within the *LLG* bipartite graph. A maximum-weight matching on the graph yields the exact worst-case permutation for a particular link and the worst-case (maximum) load on that link. We utilize this basic approach to evaluate the worst-case load on the different channels.

In the next step of the algorithm (step 4), we build the Vertex Conflict Graph (*VCG*), defined as follows.

Definition 22 The $VCG_{i,j}(VV, VL))$ is an undirected graph with $|VV| = |V|$ (i.e. with $|V|$ vertices). An edge $vl_{i,j}$ exists between two vertices vv_i and vv_j if $degree(lv_i) + degree(lv_j) > 0$. The weight of the edge is the value of the maximum weight bipartite matching of modified *LLG*, where the edges from all vertices other than lv_i and lv_j are removed.

The edge-weight assignment in *VCG* is such that if the traffic flows from a pair of cores (representing two vertices connected by an edge in *VCG*) are mapped onto the same physical channel, then they would together cause a maximum load on the channel that is given by the edge-weight.

Example 7 The *VCG* for link $l_{0,2}$ is presented in Figure 7.5. In $LLG_{0,2}$, as the two cores *core*_0 and *core*_1 have traffic flows originating from them, the edges from vertices vv_0 and vv_1 to all other vertices exist. Let us consider the edge between vv_0 and vv_5. The value of the maximum weight matching obtained on the modified *LLG* when only edges of vertex lv_0 and lv_5 are maintained is 400. Thus, the weight of the edge between vertices vv_0 and vv_5 is 400, as seen in Figure 7.5.

Then in steps 5–20, physical channels are instantiated for the link and the commodities are mapped onto the channels. The number of channels is increased from

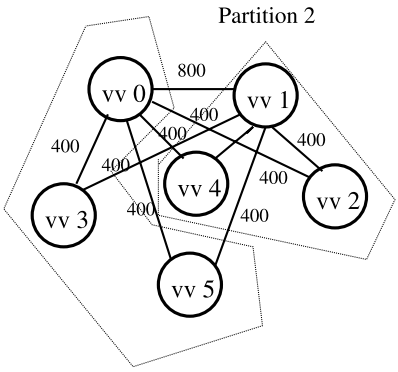


Fig. 7.5 Vertex Conflict Graph (*VCG*) and example partitions

1 until the load on each channel can be satisfied by the channel. Note that the maximum number of instantiated physical channels would be $|V|$. Thus, the traffic flows from every source that utilizes the link would be assigned to a separate channel.

For a certain number of physical channels, the $VCG_{i,j}$ is divided into that many number of partitions (step 7 of the algorithm). The partitioning is such that the sum of the edge weights cut across the partitions is maximized and the total number of vertices within each partition is almost the same. For partitioning, we use *Chaco*, an efficient hierarchical graph partitioning tool [93]. The intuition behind such partitioning is that the traffic flows that would cause higher channel loads are assigned to different channels, and channels are loaded uniformly.

Example 8 The 2 max-cut partitions of the *VCG* graph for link $l_{0,2}$ are shown in Figure 7.5. Note that vertices vv_0 and vv_1 are in different partitions.

To evaluate the load on each physical channel, we build the *Partition Loading Graph* (*PLG*) for each partition. This bipartite graph is obtained from a modified *LLG*, where the edges from all vertices other than those of the partition are removed. By finding the maximum weight matching of the *PLG*, the load caused by the partition on a channel is obtained. Then (in step 12), we check whether the load on each channel is less than or equal to the bandwidth capacity of the channel. For the channel bandwidth calculation, the data width of all the channels (*width* in Algorithm 6) is taken as an user input.

Example 9 The two *PLG* graphs for the two partitions for the mesh example are shown in Figure 7.6. The load on the two physical channels, onto which the flows from the vertices of the two partitions are mapped is 400 MB/s (obtained from the value of the maximum weight matching of each of the *PLG* graphs). If the vertices vv_0 and vv_1 had been assigned to the same partition, then the load on the channel supporting the traffic flows from the vertices of the partition would be 800 MB/s

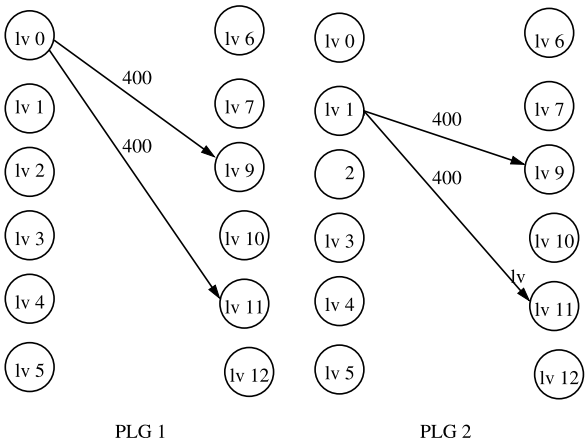


Fig. 7.6 Example physical channel loading graphs for the two partitions

(with the load on the other physical channel being 0). Thus, the partitioning process is steered to uniformly load the different channels of the link.

7.5.2 Timing Feasibility Check

In step 21 of the algorithm, the sizes of the different switches are obtained, which are based on the number of physical channels instantiated for each link. In the next step, we evaluate whether all the switches can meet the particular NoC operating frequency design point. This check is needed because, when switch size increases, the maximum supported frequency of operation reduces (as the critical path inside the switch gets longer) [34]. This information is obtained from the Place&Route of the switches, which is an input to the synthesis algorithm. Based on the frequency design point and the size of the switches, the power consumption values of the switches are obtained. For power consumption estimations, the switching activities of NoC components are obtained from several functional traffic traces. In the next step (step 26), the different links of the NoC are checked for timing violations. The length of the links are obtained from the NoC floorplan, which is taken as an input to the synthesis engine. The timing models for the interconnect wires are obtained from [58], for 0.13 μm technology.

For each frequency design point (steps 1–25, outer loop), the best NoC topology is synthesized. Finally, the most power efficient design across all these points is chosen in step 26.

7.5.3 Algorithm Run-Time

The run-time complexity of the algorithm is dominated by the maximum weight matching calculations carried out for each channel (as fast heuristics are used for partitioning, it has a low impact on the algorithm run-time). The maximum weight matching for a *PLG* graph can be computed in $O(|V|^3)$ time complexity [108] and the total number of times the matchings are performed (for each frequency design point) is at most $O(|L||V|^2)$. This is because each link can have at most $|V|$ channels and we need to perform at most $O(|V|^2)$ matchings for each link. Overall, the algorithm finds the best solution for even large CMP designs in few tens of minutes, running on a 3.2 GHz workstation.

7.6 Experimental Results

In this section, we present the experimental results obtained after applying the proposed synthesis algorithm on NoC designs with different parameter values. First, we present the application of the method to a 5×5 mesh topology. Then we study

the impact of varying the data injection rates and the number of processing cores in the design. Then we perform experiments to show the effect of link lengths on the solutions produced. The generality of the method (applicability to any CMP NoC topology and deterministic routing function) is shown next, by applying it on a torus topology with two different routing functions. Finally, the design flow predictability is validated by performing a complete layout of the synthesized NoC architecture.

7.6.1 Experiments on a Mesh Topology

In this experiment, we consider a 5×5 mesh topology. We assume the operating frequency of each core is 200 MHz, the data width of the cores and NoC channels are 32-bits, and dimension-ordered (x-first, y-next) routing is utilized. We assume that the length of each NoC link to be 1 mm. We assume these as the default values and in the subsequent subsections we study the impact of varying some of these parameters.

We vary the NoC operating frequency from 200 MHz to 1 GHz and synthesize the efficient NoC for each frequency point using the proposed synthesis procedure. The total power consumption values for the synthesized NoCs (sum of switch and link power consumption) for the different frequency points are plotted in Figure 7.7. At operating frequencies lower than 400 MHz, a large number of physical channels were needed for each link, which resulted in switches with a large number of inputs and outputs. Hence, the designed switches could not support the required NoC operating frequencies. Similarly, at the 1 GHz frequency point, the designed switches

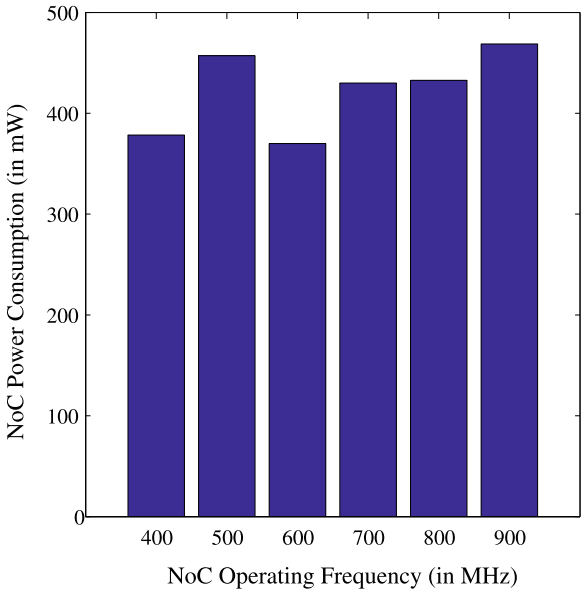


Fig. 7.7 Power consumption of 5×5 mesh topology

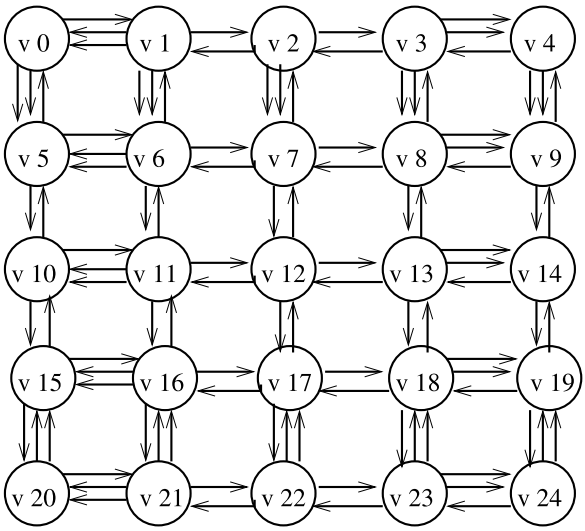


Fig. 7.8 Synthesized 5 × 5 mesh

could not support the frequency point. Thus, no feasible NoC design is obtained below 400 MHz and at 1 GHz. NoCs synthesized at lower operating frequencies (e.g., 400 MHz) require larger switches, which leads to higher power consumption. At higher operating frequencies, such as 900 MHz, the switch hardware complexity is higher (as more logic is needed to achieve faster clock speeds during physical design) and the clock-net power consumption is also higher. In fact, clock nets account for approximately 15% of NoC power consumption. Note that for the power consumption estimations of the NoC components, we run several functional traffic traces and obtain the average values. Thus, we do account for the fact that the switch input/output ports and the links of a NoC running at a higher frequency have lower switching activities than for a NoC design operating at a lower frequency. The most power optimal frequency point for the 5 × 5 mesh is 600 MHz, and the synthesized NoC at this frequency is presented in Figure 7.8.

As no previous work has directly addressed NoC design for CMPs, for comparison purposes, we evaluate how a direct extension of the approach from [108] would perform (we call this the *Reference* approach). When the procedure from [108] is applied to the 5 × 5 mesh topology, the maximum load on a link is computed to be 4× the traffic rate of each core. Thus, the NoC operating frequency required would be 800 MHz. As seen from Figure 7.7, the NoC designed using the *Reference* approach would consume 1.17× more power than the optimal NoC designed using the proposed approach.

7.6.2 Effect of Core Injection Rates

When the processor operating frequency increases, the rate of traffic injected on the NoC links also increases significantly. The actual operating frequency of the

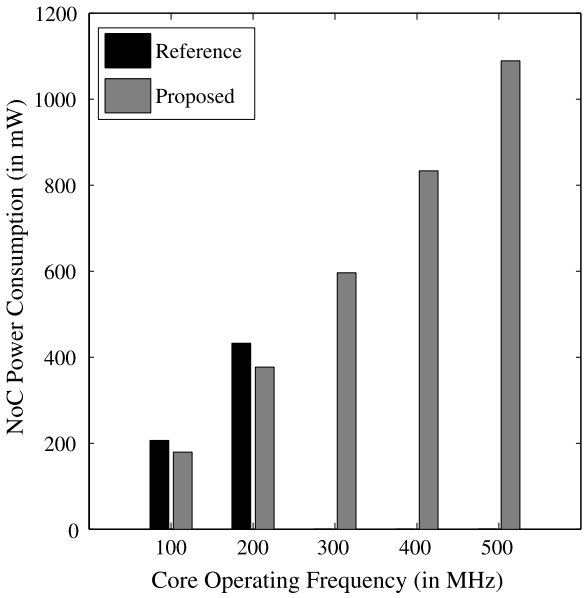


Fig. 7.9 Effect of increasing injection rates

cores varies widely across the different CMP architectures proposed in the literature. As an example, the RAW architecture has cores operating around few hundred MHz [110], while some of the commercial CMPs operate at much higher operating frequencies [103].

The power consumption requirements for different operating frequencies of the cores for the *Reference* and *proposed* approaches are depicted in Figure 7.9. This figure shows that the *Reference* approach does not produce valid NoC designs when the operating speed of the cores exceeds 200 MHz. This is because the designed NoCs needed very high operating frequency, which could not be supported by the switches. As an example, a 4×4 switch of the $\times$pipes architecture can only operate at a maximum frequency of 1 GHz approximately. While these values strongly depend on the underlying NoC architecture, the basic fact is that the *Reference* approach typically requires the NoC to be several times faster than the cores (4 times for the 5×5 mesh and higher for larger topologies). In systems where the cores themselves operate at high frequencies, it would not be feasible in practice to clock the network at such excessively high frequencies. Thus, the *Reference* approach cannot produce a valid design. On the contrary, the proposed approach supports a larger range of core operating speeds and produces more power-efficient designs as well.

7.6.3 Effect of Different NoC Sizes

The different CMP architectures available today have different number of tiles on the chip, and thus require NoCs of different sizes. As an example, for exploiting fine

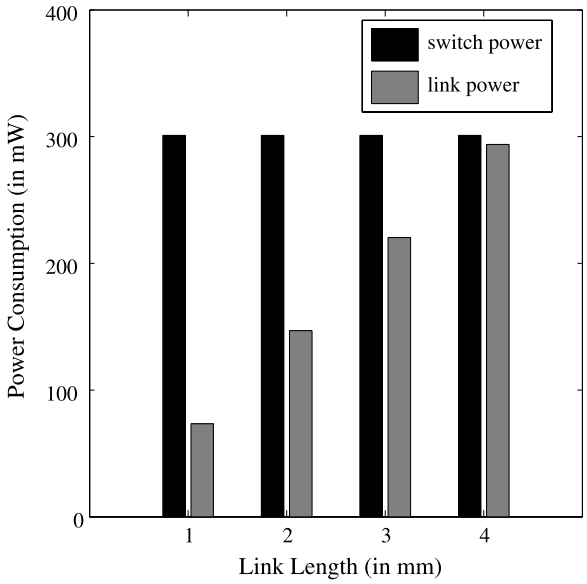


Fig. 7.10 Effect of different link lengths

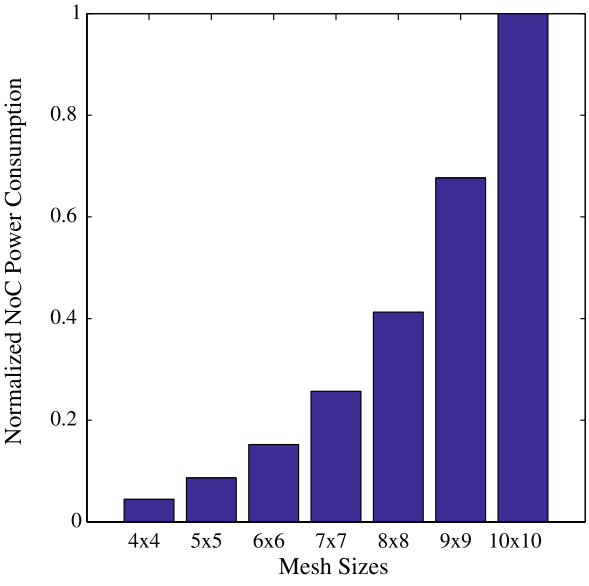


Fig. 7.11 Effect of mesh sizes

grained parallelism, CMP architectures with 50–100 tiles can be utilized, while to exploit coarse-grained parallelism, architectures with few tens of tiles are utilized [102]. In this experiment, we study the impact of different mesh sizes on the quality of the synthesized NoCs.

The NoC power consumption for different mesh sizes for the proposed synthesis approach is presented in Figure 7.11. The power numbers are normalized with

respect to the power consumed by the 10×10 mesh design. As expected, when the mesh size increases, NoC power consumption rapidly grows as well. Even for the largest 10×10 mesh design, the method completed in few tens of minutes on a 3.2 GHz workstation. This shows that due to the use of fast heuristics and exact polynomial algorithms, the proposed synthesis method is highly scalable to large problem instances.

7.6.4 Effect of Link Length

To see the importance of considering wire power consumption during the synthesis process, we have varied the length of the NoC links in the design. For this experiment, we fixed the NoC topology to be a 5×5 mesh. Thus, the designs with different link lengths represent designs with different total chip area. For example, when the link length is 1 mm, the dimensions of the mesh NoC are 5×5 mm, but when the link length is 4 mm, the dimensions are 2×20 mm.

The motivation for considering different link lengths is that different CMP architectures have wires of different lengths. As an example, in the *Smart Memories* architecture [101], the link lengths of the global network are around 4 mm [111], while the link lengths in a smaller NoC design are from 1 mm to 2 mm [88].

The NoC switch and link power consumption values for different link lengths are presented in Figure 7.10. As the link length starts to increase, the link power consumption largely augments. This shows that the wire power consumption must be considered during the NoC synthesis phase, as it is done in this approach. Note that the power numbers are for 130 nm technology. With more advanced process technologies (especially at 90 nm and below), the impact of wire power consumption on the total NoC power consumption is expected to increase considerably [106]. Thus, the exploration of such technology dependent effects is a necessary direction for future work in the design of efficient on-chip interconnects.

7.6.5 Application to Torus Topology

The proposed approach is applicable to any NoC topology and deterministic routing function. We have applied it to a 5×5 torus topology and studied the impact of 2 different routing functions: One routing function in which the wrap-around links of the torus are not used (*Routing 1*), and another one where the wrap-around channels are utilized (*Routing 2*). The NoC power consumed by the synthesized designs for the two routing functions are shown in Figure 7.12. It shows that the use of the wrap-around links in the torus topology is beneficial, not only as generally believed for latency, but also for power. This is because when the wrap-around links are utilized, the traffic is spread more evenly in the network. Thanks to the proposed synthesis approach, this type of architectural test can be easily performed, showing its effectiveness for NoC design exploration purposes. Usually, standard NoC topologies,

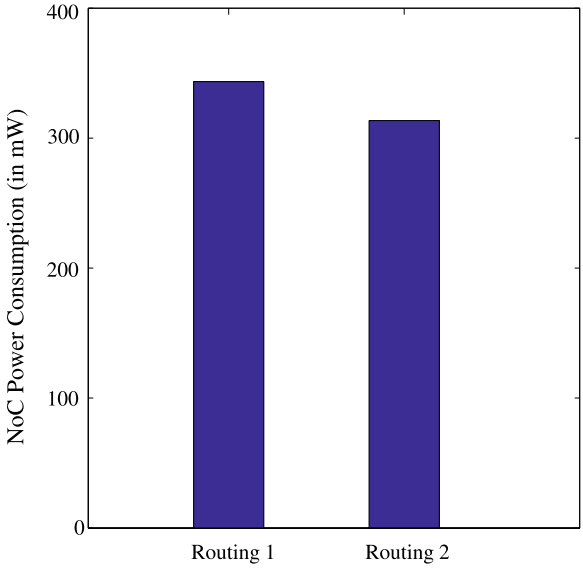


Fig. 7.12 Results for torus topology

such as mesh and torus, are used for CMPs, as the NoC floorplan for such topologies is predictable [100, 101]. This is the reason for choosing these topologies for the experiments. However, the synthesis approach is general and applicable to any NoC topology.

7.6.6 Validating Design Flow Predictability

Usually, a design gap exists between the architectural level model and the actual physical layout implementation. Bridging this design gap is key to decrease the number of design iterations and to achieve quicker design convergence and faster time-to-market. In this work, we achieve a predictable design flow by bridging this design gap between the architectural and physical models. This is achieved due to two factors. First, we consider the physical layer measures, such as wire delays and accurate NoC component delays, during the synthesis process. Second, the use of regular NoC topologies results in easily predictable NoC floorplan and link lengths, which help us to accurately model the wire delays. In fact, achieving a predictable design flow is one of the most important reasons for utilizing NoC-based interconnects [88]. To validate the predictability of the design flow, we implemented the layout of the optimal 5×5 mesh topology synthesized by the procedure at 600 MHz. The CMP consists of 25 cores, and the area of each core is 1×1 mm.

To obtain the layout, we have first generated the RTL code of the designed NoC components using a custom built tool, $\times$ipesCompiler [62]. Then we have synthesized the RTL design using Synopsys Design Compiler [98]. After this, we have performed the place&route phase of the synthesized design using Cadence SoC Encounter [99]. The resulting layout of the design is presented in Figure 7.13. For the

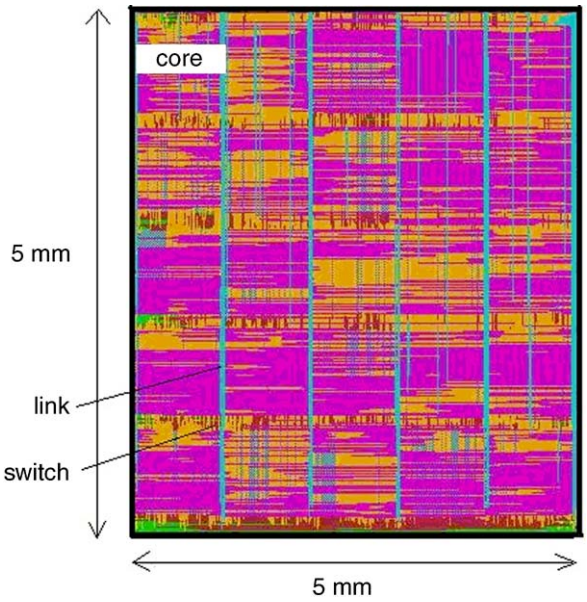


Fig. 7.13 Layout of a 5 × 5 mesh topology

layout, a 0.13 µm process technology with 8 metal layers are used for wire routing. Among these, 5 metal layers are used for intracell routing inside the cores and the remaining 3 metal layers are used for over-the-cell routing of NoC links.

We have performed post-layout timing checks on the different switches and links of the NoC. We could achieve a fully functional design at the target frequency of 600 MHz, without any timing violations. We could design the NoC till layout level quickly, thanks to the predictability of the design flow.

Finally, we studied the impact of adding multiple physical channels on NoC area. For the 5×5 mesh topology, the use of multiple physical channels increased the total switch area from 0.94 mm^2 (when only a single physical channel is used for all the links) to 1.18 mm^2, which is negligible when compared to the total chip area of the CMPs. From the layouts, we also found that sufficient routing area was available for the multiple physical channels that were instantiated. This is in accordance with several earlier studies [22, 88], which have shown that sufficient routing area is available between the switches of regular topologies to route a large number of wires.

7.7 Summary

Having a predictable interconnect architecture is critical to manage the increasing interconnect complexity of current *Chip Multiprocessors* (*CMPs*). The CMPs differ from *Application Specific SoCs* (*ASSOCs*) in the fact that their traffic characteristics cannot be predetermined. Thus, the NoC predictability for CMPs needs to be tackled at several design levels. On the one hand, from the architectural viewpoint, the

interconnect has to provide predictable performance under different operating conditions. On the other hand, from the design flow viewpoint, the design gap between the architectural model and the physical implementation should be minimized, so that a quicker design convergence is obtained. Designing an efficient NoC architecture that provides predictable performance for any application running on a CMP is a challenging task.

In this chapter, we have presented a synthesis method that addresses this important design issue of synthesizing the most power efficient NoC interconnect for CMPs, providing guaranteed optimum throughput and predictable performance for any application to be executed on the CMP. We achieve a predictable interconnect design in two ways: first, the architecture is designed to provide predictable performance under all application traffic conditions. Second, the synthesis approach considers accurate information of the physical layer measures, such as wire-lengths, wire delays, and network component delays, thereby bridging the gap between the synthesis models and the actual physical layout implementation. This leads to a faster design cycle and quicker design convergence across the high level synthesis approach and physical implementation of the design. We have validated the design flow predictability of the proposed approach by performing a layout of the NoC synthesized for a 25-core CMP. The proposed synthesis approach can also be used as a design space exploration tool to evaluate the efficiency of different NoC topologies and routing functions. Finally, the presented approach maintains the predictable layout of regular NoC architectures; thus, it can be applied to existing NoC architectures.

In the preceding chapters, we have seen methods to design NoC architectures under various operating conditions. Now, it is time to proceed to make their operation reliable. This will be focus of the next part of the book.

Part II
NoC Reliability Mechanisms

Chapter 8
Timing-Error Tolerant NoC Design

With technology scaling, the device characteristics fluctuate to a large extent due to process variations and can cause significant variations in wire delay [122]. Wire delay is also affected by other forms of interference such as supply bounce, transmission line effects, etc. [123, 124]. As such delay variations can affect multiple bits simultaneously, special mechanisms are needed to handle timing errors. In this chapter, we present *T-error*, a timing-error tolerant mechanism to make the interconnect resilient against timing errors arising due to such delay variations on wires.[1]

Current NoC design methodologies are based on a worst-case design approach that considers all the delay variations that can possibly occur due to the various noise sources and environmental effects and targets a *safe* operation of the system under all conditions. The system state is considered *safe* if there are no timing violations for all operating conditions and in the presence of the various noise sources. Such a conservative design approach targets timing error free operation of the system. In *Razor* [113, 114], an aggressive, better than worst-case design approach was presented for processor pipelines. In such a design, the voltage margins that traditional methodologies require are eliminated and the system is designed to dynamically detect and correct circuit timing errors that may occur when the worst-case noise variations occur. *Dynamic Voltage Scaling* (*DVS*) is used along with the aggressive design methodology, allowing the system to operate robustly with minimum power consumption.

The proposed *T-error* methods are used to aggressively design the NoC components (switches, links, and NIs) to support higher operating frequencies than designs based on conservative approaches. Aggressive design of the communication architecture has several implications when compared to the design of processor pipelines. First, the hardware overhead required to recover from timing errors can be minimized by smart utilization of the buffering resources available in the NoC. Second, the error recovery penalty can be mostly hidden under the network operation, so that large performance benefits can be obtained. Finally, the switches, NIs should be redesigned to handle errors, as they may receive a wrong piece of data before the right one.

In many SoCs, *Dynamic Frequency Scaling* (*DFS*) and *Dynamic Power Management* (*DPM*) policies are used to reduce the operating power of the SoC [55]. In such systems, at the application level, the voltage and frequency of the components are selected to match the performance level of the application. The NoC can also be dynamically tuned at runtime. When a communication-intensive application

[1] We would to acknowledge the contributions of Rutuparna Tamhankar, Stergios Stergiou, Antonio Pullini, Dr. Federico Angiolini, Prof. Luca Benini, and Prof. Giovanni De Micheli.

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

requires fast execution, the NoC can be over-clocked to higher operating frequencies. When an application does not require a fast NoC, the frequency of the NoC can be lowered to reduce the power consumption of the system. Unlike many of the earlier works [113], where the system's error rate is constantly monitored to tune the voltage or frequency, we envision that the *T-error* based NoCs to be utilized in systems with application-level DFS/DPM policies. Thus, complex network error rate monitoring controllers are not needed in the design. Moreover, the large delay incurred to change the frequency/voltage to reduce errors is avoided. The required voltage and frequency parameters of the network for the different applications can be stored in programmable registers or memories and can be accessed by the operating system upon task switches among the applications that are running on the SoC.

In this context, we distinguish two possible operating modes for the NoC: *normal mode* and *over-clocked mode*. In *normal mode*, the NoC operates at frequencies less than or equal to the frequency of a conservative design. Under *over-clocked* mode, the frequency of operation can be higher than that of the traditional design. The NoC under the *over-clocked* mode incurs some penalty for error resiliency, even when there are no errors in the system (this is explained in detail in Section 8.4.2). Under *normal mode*, the NoC does not need to encounter the additional error resiliency penalty, as it operates at a *safe* operating frequency. To remove any additional overheads when in *normal mode*, we present a way to dynamically configure the NoC between the *normal* and *over-clocked* modes of operation at the application level.

The *T-error* scheme for a NoC link is presented in [137]. In this work, we present two robust link design methods. In the first scheme, link buffers are efficiently utilized, so that error resiliency is achieved without much additional hardware overhead. In the second scheme, more hardware resources are used to achieve higher performance. The two link schemes have the same timing relation and logic interpretation of control signals from/to the switch. The two schemes can be used in a *plug-and-play* fashion by the designer to suit the application and NoC architecture characteristics. We integrate the link designs with NoC flow control and present *T-error* schemes for switches/NIs.

We developed cycle-accurate SystemC models of the *T-error* based switches, links, and NIs and integrated them onto the $\times$pipes NoC architecture. Functional SystemC simulations on several benchmark applications have been carried out. Detailed case studies of the *T-error* design and comparisons with the traditional mechanisms are presented. Experiments show large performance improvements (up to 33% reduction in communication delay) for the benchmark applications for the aggressive NoC design methodologies, when compared to traditional design methodologies. The application of DVS/DFS techniques result in 57% reduction in the NoC power consumption when compared to traditional design approaches.

8.1 The Double Sampling Technique

In most NoC realizations, when errors are detected, corrupted packets are retransmitted. Unfortunately, retransmissions incur significant performance penal-

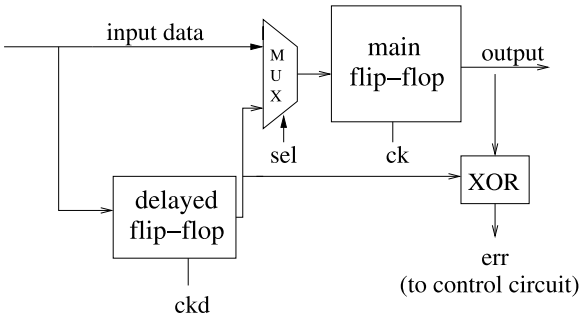


Fig. 8.1 Double data sampling technique

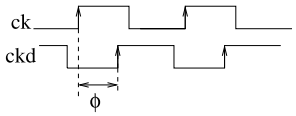


Fig. 8.2 Phase shift between clocks

ties [130]. Moreover, timing delay variations occurring due to higher operating frequencies can potentially affect multiple data bits in a packet, requiring complex multibit error detecting/correcting codes that may be impractical to use [130].

To recover from timing errors in a digital system, *double data sampling techniques* have been proposed and used by several researchers [113–120]. In such double sampling schemes, each pipeline flip-flop in the design (called *main flip-flop*) is augmented with an additional latch/flip-flop (called *delayed flip-flop*), as shown in Figure 8.1. Both the *main* and the *delayed flip-flops* have the same frequency of operation. However, the clock to the *delayed flip-flop* has a phase shift from the clock to the *main flip-flop* and it samples data at delayed clock edge, as shown in Figure 8.2.

Thus, data sampled by the *delayed flip-flop* has more time to settle, compared to the *main flip-flop*. The delayed clock is usually generated locally at the pipeline stage from the main clock using an inverter chain (delay element). After that the *delayed flip-flop* has sampled data, the values of the two flip-flops are compared through an *EXOR* gate; if there is any difference, data from the *delayed flip-flop* is assumed to be correct and is resent through the *main flip-flop* in the next clock cycle. The control circuitry also sends flow control signals to the pipeline stages before and after the stage where the error occurred, so that they can recover from the error.

Let us consider a bit-line of a NoC link with one pipeline stage, where the pipeline flip-flop (*main flip-flop*) is augmented with a *delayed flip-flop*. Let the maximum *safe* operating frequency of the link for the original design (without using any double-sampling technique) be 1 GHz. If the double sampling technique is used, we can have a higher frequency of operation, as the link no longer needs to have *safe* operation at the main flip-flop. As an example, if the delay or phase shift between the clocks to the main and delayed flip-flops (ϕ/(clock period) in Figure 8.2) is 50%, the delayed flip-flop will sample the right data even when the link operates at 1.5 GHz. Even though the *main flip-flop* may incur timing errors, we can recover the right data from the *delayed flip-flop*.

Note that higher operating frequency can also be achieved by having a deeper pipeline in the NoC components. However, there are several advantages in using the *T-error* based design than having a deeper pipeline:

1. When the NoC is operating in the *normal mode*, a deeper pipeline depth will result in a fixed increase in latency across the link, while in the *T-error* based scheme, this latency is avoided (in fact, *T-error* design can be viewed as a way to dynamically change the pipeline depth of the NoC components).
2. As the traditional design frequency is conservative, even in the *over-clocked* mode the errors introduced due to over-clocking may not be substantial. Thus, the *T-error* design can achieve the same frequency of a deeply pipelined design with a lower latency for the average case. This is because, in the *T-error* design, the pipeline depth changes dynamically according to the error rate, while the deeply pipelined design always incurs a high latency.
3. Significant redesign, verification, and timing validation of switches and NIs are needed to increase the pipeline depth, while the *T-error* design can be incorporated with lower design efforts. The normal FIFOs used in the links, switches and NIs need to be replaced by the *T-error* FIFOs, which can be designed and used as library elements.
4. *T-error* can always be used as an add-on to a deeply pipelined NoC system to improve the operating frequency of the system.

In this work, we present methods that address only the timing delay variations on the NoC that are introduced due to over-clocking. Coping with other kind of errors (such as soft errors, capacitive coupling based cross-talk, data upsets, etc.) is assumed to be done by means of existing techniques (such as [126–135]). By operating the NoC at higher frequencies, the effect of these errors on the system may vary and we assume that the techniques used to address them are designed to handle the maximum over-clocked frequency of operation.

8.2 Using Links as a Storage Medium

Flow control is needed in networks to support full throughput operation. Specifically, it is needed to ensure that enough buffering is available at each switch to store the incoming data and the available buffers are utilized efficiently. In traditional designs, queuing buffers are either located at the inputs (*input-queued* switches) or at the outputs (*output-queued* switches). In some switches, the buffers can be located at both the inputs and the outputs to improve the performance of the NoC [94]. A *credit-based* or *on/off flow control* mechanism is typically used to manage the input buffers of the switch. In such designs, for maximum network throughput, the number of queuing buffers needed at each input of the switch should be at least $2N + 1$ flits [94], where N is the number of cycles needed to cross the link between adjacent switches. This is because in credit-based flow control, it takes at least 1 cycle to generate a credit, N cycles for the credit to reach the preceding switch, and N cycles for a flit to reach the switch from the preceding switch [94]. To support

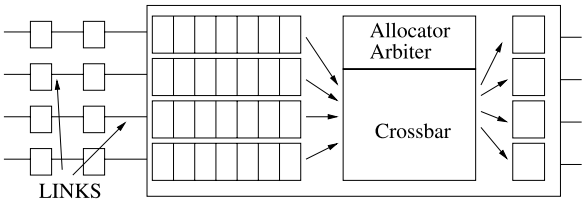


Fig. 8.3 Input queued switch

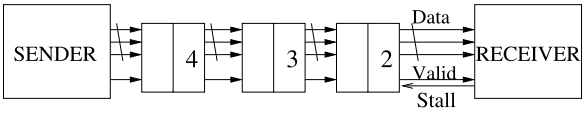


Fig. 8.4 Modified link design with 3 stages

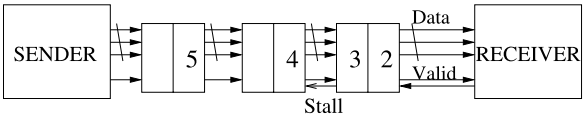


Fig. 8.5 Entry 3 buffered in *secondary flip-flop*

link pipelining, there need to be $N-1$ pipeline buffers on each bit-line of the link connecting the switches. Thus, effectively we need $3N$ flit-buffers for each input of the switch/link (Figure 8.3).

In [70], the use of relay stations and link-level flow control has been presented. In such a scheme, each pipeline flip-flop on the link is replaced by a 2-entry FIFO and a link-level flow control is used to ensure full throughput operation. We utilize such links for the NoC architecture. In the NoC architecture, the switch input buffers are also replaced by a 2-entry FIFO. Figure 8.4 shows a 3-stage link pipeline using 2-entry FIFO at each pipeline stage ($N=4$, as it takes 1 more cycle to reach the receiver from the last pipeline stage of the link). The scheme has two control signals (*stall* and *valid*) transmitted between sender, receiver, and the link pipeline stages. The *stall* signal is sent by the receiver and flows in the opposite direction to that of the data, while the *valid* signal is driven by the sender and it flows in the same direction as that of the data. The sender or receiver may be a switch or a network interface. The receiver generates a *stall* signal when its storage capacity is full or if it receives a stall request from the following stage. The *valid* signal informs that the data which was received in the previous cycle (at the previous rising edge of clock *ck*) is valid. During normal operation (i.e., when there is no stall request), only one of the flip-flops in the 2-entry FIFO is used, as shown in Figure 8.4. When a *stall* signal is received by the 2-entry FIFO (shown in Figure 8.5), the data on output of the *main flip-flop* is stalled and the new data is received by the *secondary flip-flop*. The *stall* signal is propagated to the previous stage, as shown in Figure 8.6. The schematic of the 2-entry FIFO is shown in Figure 8.7.

This flow control mechanism ensures full throughput operation with performance similar to that of *input-queued* switches with credit-based or on/off flow control. As previously shown, in traditional *input-queued* schemes (Figure 8.3), the total number of buffers needed for maximum throughput is $3N$, as compared to only $2N$

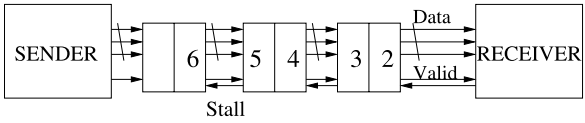


Fig. 8.6 The *stall* signal propagated to previous stage

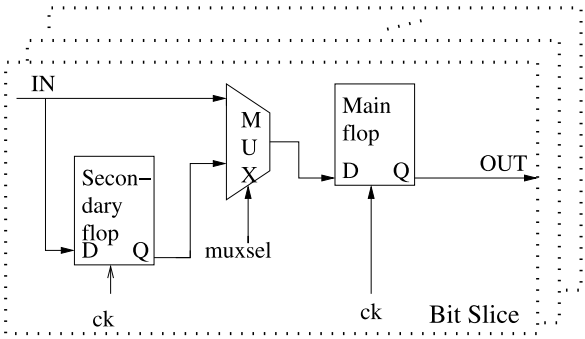


Fig. 8.7 A 2-entry FIFO. The control circuit is common for all the bit lines

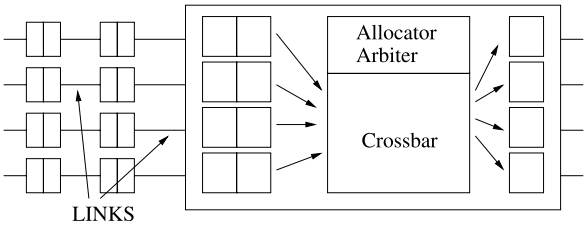


Fig. 8.8 Modified link and switch design

buffers ($2 \times (N - 1)$ along the link and 2 at the switch input) in this scheme (Figure 8.8). The traditional *input-queued* design has one flip-flop at each link pipeline stage. In the *stall/valid* protocol, it takes one clock cycle for the *stall* signal to reach the preceding pipeline stage. During this time, the data which is in transit from the preceding pipeline stage cannot be stored when it reaches the current pipeline stage. Thus, for full throughput operation in such a scheme, the link flip-flops are not used for queuing data, and instead data is queued at the input of the next switch. By augmenting the link pipeline stage with one more flip-flop, the full throughput operation is achieved. As we also utilize the pipeline flip-flops, the scheme leads to reduced buffering requirements. As the link buffering scheme can be viewed as merely spreading the FIFO buffers of the switch inputs onto the links, it maintains the same deadlock and livelock properties of a design with *input-queued* switches. Moreover, as all the inputs of a switch have same buffer count in the link-buffer scheme, the switch design becomes more modular, when compared to the traditional switch design. Note that the control circuit used at a link pipeline stage in this scheme is common for all the w data bits in a flit of the NoC, and thus the overall cost of the control circuit is negligible.

8.3 *T-error* Link Designs

In this section, we present two link designs to support timing error tolerant operation needed for over-clocking the links. The first design reuses the link FIFO for error recovery with very little hardware overhead (the overhead is only for the control circuitry). This scheme, in the worst case, can incur a 1-cycle penalty for each error occurrence at a pipeline stage. In the second link design scheme, the 2-entry FIFOs are augmented with an additional flip-flop. The resulting design is a high-performance link that incurs a 1-cycle penalty only for the first occurrence of an error for a continuous stream of data at each pipeline stage. The design is such that all subsequent errors are automatically resolved.

8.3.1 Scheme 1: Low overhead T-error Links

In the *T-error* scheme, the 2-entry FIFOs along the links are modified to support timing error tolerant operation. The modified FIFO structure is shown in Figure 8.9. The second flip-flop of the FIFO is clocked at a delayed clock (*ckd*) compared to the clock *ck* of the *main flip-flop*. *ckd* and *ck*, however, feature the same period. The phase shift among them is configured after proper delay analysis, as will be discussed later.

The incoming data is sampled twice, once by the *main flip-flop* (at time instant t_0 in Figure 8.11) and then by the *delayed flip-flop* (at time instant t_1). There are two modes of operation at each pipeline stage of the link: *main mode* and *delayed mode*. Initially all the pipeline FIFOs are set to the *main mode* and data transmission begins. In every cycle, at the clock edge *ck*, the *main flip-flop* captures and transmits the incoming data. At clock edge *ckd*, the *delayed flip-flop* captures the incoming data and the error detection control circuit checks whether there is any difference between the main and the *delayed flip-flop* values. As shown in Figure 8.9, an `EXOR` gate is connected to the outputs of the *main flip-flop* and *delayed flip-flop* to detect a timing error. The *err* signals of all w bits of the flit (vertically across the width of the link) at a pipeline stage are `ORed` and fed as an input to the control circuit.

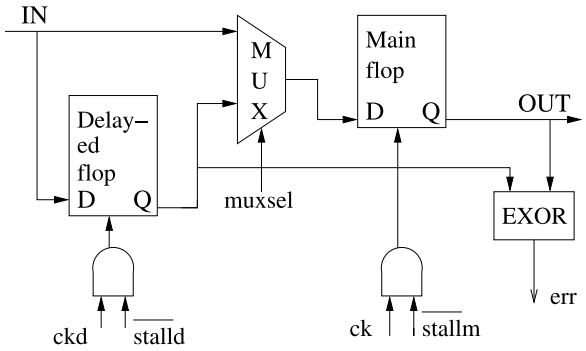


Fig. 8.9 Low overhead *T-error* buffer

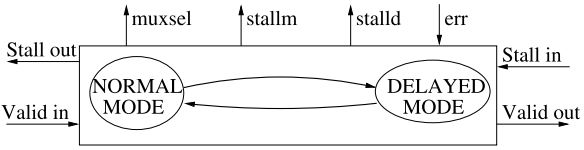


Fig. 8.10 Control circuit for scheme 1

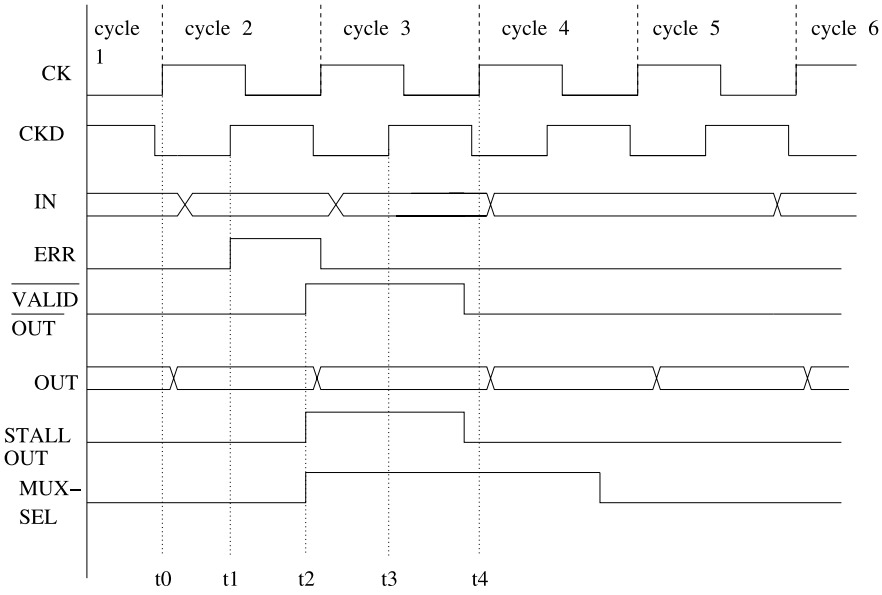


Fig. 8.11 Waveforms for scheme 1

Thus, a timing error in any bit of the flit causes the entire flit to be resampled at the pipeline stage. The control circuit at each pipeline stage, which is common for all the bit-lines of the link, is presented in Figure 8.10.

If there is an error in the data sampled by the *main flip-flop*, the data that was transmitted at clock edge *ck* is incorrect. The correct data from the *delayed flip-flop* is sent at the next clock edge (at time instant t_2). Whenever a timing error occurs (i.e., *err* signal is set to one), a *stall* signal is sent to the previous stage such that the previous stage is stalled for one cycle. Also, a $\overline{valid}$ signal is sent to the following stage, informing that the data sent in the previous cycle was nonvalid.

A FIFO at a pipeline stage of the link enters the *delayed mode* when a *stall* signal from the next stage causes queuing of data at the FIFO. The *stall* signal can be issued to handle regular congestion, that is as a flow control wire, or to let the downstream stage sort out an error condition. When a FIFO is in *delayed mode*, all timing errors are automatically avoided, as the incoming data is always sampled through the *delayed flip-flop*. Thus, in networks with severe congestion, most timing errors are automatically avoided. Examples of operation of the FIFOs for a network with no congestion and with congestion are presented in Figures 8.12 and 8.13. In the network with no congestion, at each pipeline stage, data is always directly

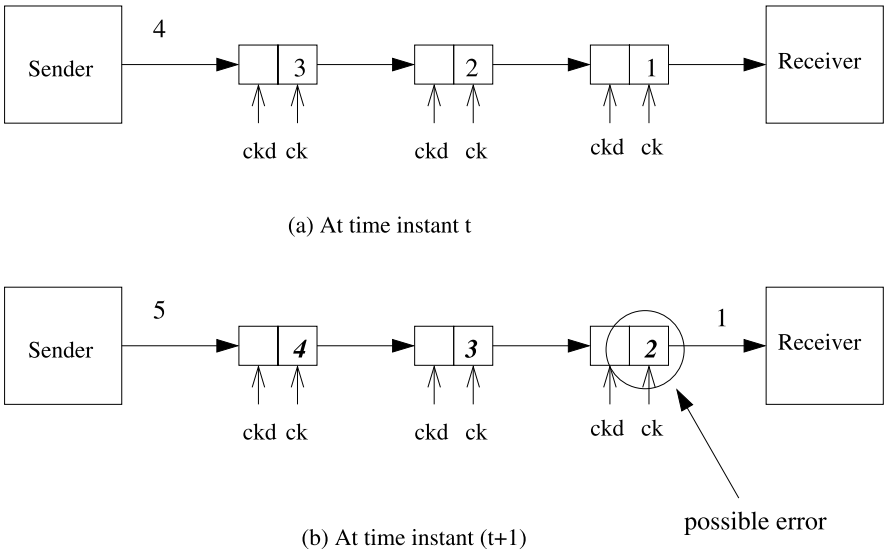


Fig. 8.12 Network operation without congestion. The data in the FIFOs at time instances t and $(t + 1)$ are presented in (**a**) and (**b**)

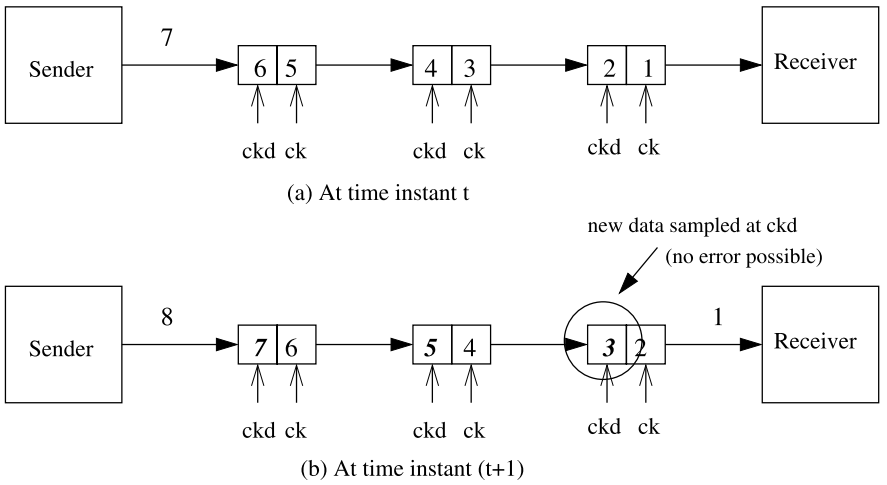


Fig. 8.13 Network operation under congestion. The data in the FIFOs at time instances t and $(t + 1)$ are presented in (**a**) and (**b**)

sampled by the *main flip-flop* and sent out by it. In the network with congestion, the data from the preceding pipeline stage is always captured by the *delayed flip-flop* at the current pipeline stage, and later sent out by the *main flip-flop*. Since data is always sent at *ck* from the preceding stage and sampled at *ckd* in the current stage,

the wire transitions have more than one clock period to settle, and thus timing errors are automatically avoided. In the worst case, if the FIFO always operates in the *main mode*, each timing error occurrence will incur one clock cycle penalty for recovery.

However, in the worst case, when there is no congestion and the FIFO always tries to operate in *main mode*, each timing error occurrence incurs 1 clock cycle penalty for recovery. The link stage switches from *main mode* to *delayed mode* and back for each faulty piece of data. Detailed performance analysis of this scheme and comparison with the next link design scheme for several benchmark applications is presented in Section 8.6.6.

The amount of timing delay that is tolerated by the *T-error* design depends on the phase shift between the clocks of the *main* and the *delayed flip-flops*. This shift should be as large as possible, so that the *delayed flip-flop* is guaranteed to sample the right data and to provide correct system operation. However, the maximum shift is constrained by internal repeater delays (the error detection logic must operate between a *ckd* edge and the following *ck* edge). Detailed timing analysis and SPICE simulations (for a link size of 32 bits) showed that clock *ckd* can be delayed by 53.3% of the clock period with respect to *ck*. In this work, we assume that a maximum delay of 50% of the clock is tolerable with a *T-error* enabled system. Thus, the delayed clock *ckd* is just the inverted value of the main clock, and delay chains are not needed to generate it. At the same time, the maximum delay which is tolerated on a wire is 150% of the clock period, providing ample margin for timing error correction. In the *T-error* scheme, metastability conditions may occur and are corrected using efficient transistor-level implementation of the FIFO circuit, which are presented in [137]. The control lines (*stall, valid*) that need to have error-free operation can be made robust using a variety of methods (such as using wider metal lines, shielding). We refer the interested reader to [137] for transistor-level implementation details, timing analysis, and SPICE simulation results of the *T-error* scheme.

8.3.2 Scheme 2: High-Performance T-error Links

The performance of the above link design can be improved by having an additional flip-flop to store incoming data whenever a stall is encountered. A 3-entry FIFO, instead of the 2-entry FIFO previously described, is used in this scheme (refer to Figures 8.14 and 8.15). The third flip-flop, called *auxiliary flip-flop*, is added in series to the *delayed* and *main flip-flops*; it also samples data on rising edges of the delayed clock *ckd*. The operation is similar to the above design, except that for a continuous stream of data, even if all incoming pieces of data were to be corrupted, only a single 1-cycle penalty would incurred to correct timing errors at a pipeline stage. This is because the FIFO enters the *delayed mode* upon the first error occurrence; once in this mode, all subsequent pieces of data are sampled through the *delayed* and *auxiliary flip-flops*, making them automatically error free. The presence of the *auxiliary flip-flop* lets the link stage continue operating even upon fault occurrences; the sender does not perceive any interruption in data flow.

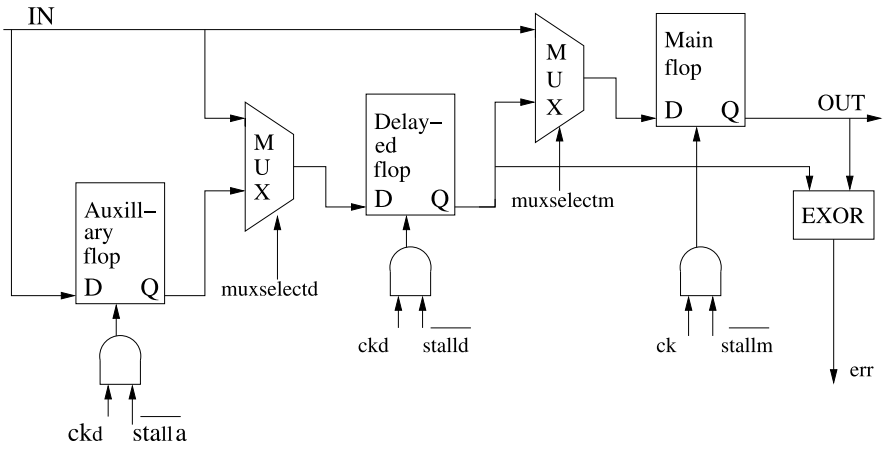


Fig. 8.14 Schematic for scheme 2

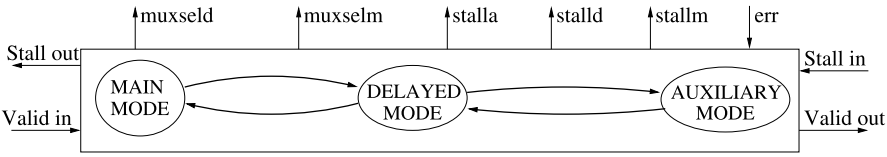
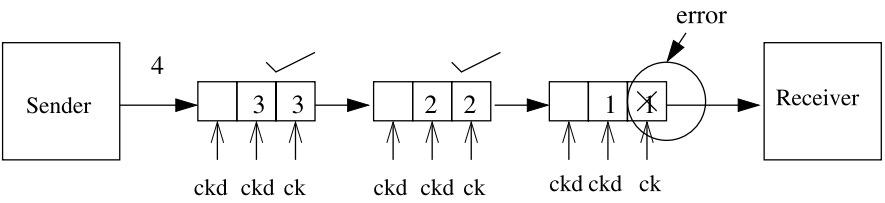
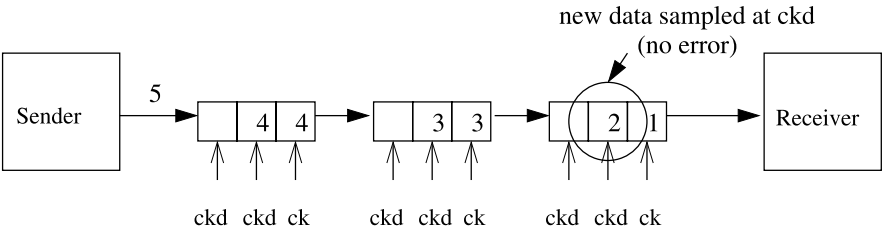


Fig. 8.15 Control circuit for scheme 2



(a) At time instant t, clock edge ckd



(b) At time instant (t+1), clock edge ckd

Fig. 8.16 Example of 3-entry FIFO operation where for a continuous stream of data, an error occurrence at a pipeline stage causes further errors to be automatically avoided at that stage

Only at the end of the whole data stream, the stage empties and switches back to *main mode*. An example is presented in Figure 8.16. Note that even in absence of timing errors, the *auxiliary flip-flops* can still improve general system performance, as they also behave as queuing buffers to minimize congestion-related penalties.

8.4 Aggressive Switch/NI Design

In this section, we describe the changes needed in the basic architecture to support the *over-clocked* mode of operation. The ×pipes NI is composed of two modules: a front-end interface with the cores and a back-end interface with the switches and links. The NI back-end is the only part that needs to support NoC over-clocking. Since its architecture is similar to that of the switches, we describe only the changes required in the switches.

There are two changes required in the switches to support NoC over-clocking. The first is that the switches should also be able to operate at higher frequencies to utilize the faster links. The other is that the switches should be able to handle the data from the links that may have timing errors. A NoC switch, as shown in Figure 8.8, consists of input buffers, allocator/arbiter, crossbar and output buffers. In the link based flow-control, there is a two entry FIFO at the input of the switch, which can be made timing-error tolerant, similar to the link FIFO *T-error* schemes presented in the previous section. The switch design changes will now be presented.

8.4.1 Output Buffer Changes

In an *input-queued* switch, normally a single register is used at each output to store data, before sending the data onto the links. Note that in some designs, the output buffer can be taken to be part of the link design, depending on the targeted operating frequency of the switch. In some other cases, more than one buffer may be used at each output, so that the performance of the NoC can be improved. In the ×pipes architecture, the number of buffers at the output is a parameter that can be configured by the user according to his or her application needs.

As a starting point, the architecture of a ×pipes switch with a single output buffer is shown in Figure 8.8. The ×pipes switch already supports distributed buffering along the links. In this architecture, the switch has a latency of 2 cycles for data transfers. There are two sets of flip-flops in the switch that may cause timing violations when over-clocked: output buffers and flip-flops that are used to maintain the allocator/arbiter states. From synthesis of the ×pipes architecture, we found the operating frequency of the original switch to be 1 GHz. The path from the input of the switch to the state flip-flops was 0.4 ns, while the critical path was from the input to the output (which also samples the arbiter/allocator states). With over-clocking, we target a 1.5× increase in frequency (i.e., 1.5 GHz operating frequency) of the

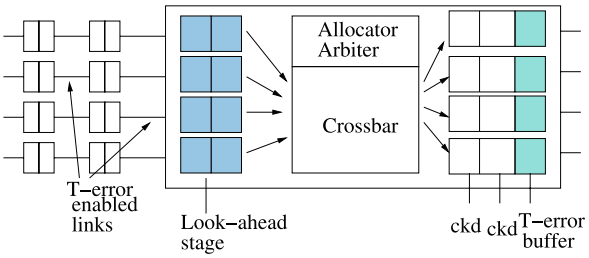


Fig. 8.17 Over-clocked switch design with output and input buffer changes

switches. Therefore, we found that the state flip-flops are *safe* even under over-clocking, since the available cycle time is 0.66 ns, and that only the output buffers need to be made timing error tolerant. Note that in other switch architectures, if the state flip-flops are not safe when over-clocked, they should be *T-error* enabled as well. Otherwise, the amount of over-clocking will be limited by them. Also, if the switch has more pipeline stages, the *T-error* principle needs to be applied to each pipeline stage.

In order to over-clock the switch, we apply the *T-error* design to the head flip-flop of the output FIFO and the other flip-flops in the output FIFO are made to sample data at *ckd*. Figure 8.17 shows the changes in the output buffer of the switch. Note that errors can occur only when the data is sampled through the head of the FIFO and when the NoC operates in the *over-clocked* mode.

8.4.2 Input Buffer Changes

When timing errors occur at a link pipeline stage, wrong data can reach the switch input before the correct data is received. If the switch samples wrong data, several complications can arise. As an example, timing errors on the routing fields of the header flit may result in misrouting a packet. In order for the switch to handle data errors, there are several cases to be considered and recovering the switch state from such cases require complex hardware and control circuits [94]. Another way to detect wrong data at the switch input is to use some error detecting code (such as cyclic redundancy check) for each flit of the packet. However, in the *over-clocked* mode, all the bits of the data could encounter timing errors and such schemes may be inefficient. Thus, to simplify the switch hardware, we use a look-ahead stage at the input of the switch that ensures that correct data is always fed to the internal switch logic (see again Figure 8.17). The look-ahead stage stores an incoming flit for one clock cycle, i.e., until the *valid* line indicates whether the received data was correct or not. In case of correct reception, data is fed to the switch arbiter/allocator. Otherwise, it is discarded by the look-ahead stage. Note that even when there are no errors occurring in the system, a latency penalty could arise from insertion of the look-ahead buffers, unless properly tackled, as explained in the next section.

8.5 Dynamic Configuration of the NoC

When the frequency of the NoC is varied based upon DFS/DPM techniques, the NoC may operate at frequencies lower than or equal to the conservative design frequency. In such a *normal operating mode*, the error resiliency penalty due to *T-error* needs to be completely hidden. The *T-error* mechanism at the link FIFO and the switch/NI output buffers incur error resilience penalty only when an error occurs. Thus, they dynamically adjust to the errors happening in the system. However, the look-ahead stage at the input of the switch incurs a 1-cycle penalty even under the *normal* operating mode. To avoid this 1-cycle penalty in the *normal mode*, we use a global *BOOST* signal that is issued at the application level by (one or more) processing cores. A value of $BOOST = 1$ indicates that the NoC is in *over-clocked* mode, while $BOOST = 0$ indicates *normal mode* of operation. The *BOOST* signal may take several clock cycles (tens of cycles) to spread to all the switches and NIs in the NoC. The actual transition between the *normal* and *over-clocked* modes occur after the *BOOST* signal is completely spread around the NoC.

The input buffer control logic is modified such that the *look-ahead stage* is used only when $BOOST = 1$, as shown in Figure 8.18. The transition from the *normal mode* to *over-clocked* mode is smooth in the design, as the look-ahead is started when the *BOOST* signal is spread. However, transition from the *over-clocked* mode to the *normal mode* requires special care, as there may be some residual errors in the NoC. To make a smooth transition dynamically (i.e., without flushing all the data in the network), we use the following design change. In the *T-error* NoC, all residual errors are maintained on the links between the switches, as the switches always receive the right data due to the look-ahead mechanism. When a transition to the *normal mode* occurs, the look-ahead stage is bypassed only when there is no incoming data from the link. Thus, any data from the output buffer of the switch or the link that may have residual errors goes through the look-ahead stage, which ensures that the right data is fed to the switch inputs. As the transitions between *normal* and *over-clocked* modes occur at the application level (which may occur every tens of thousands of cycles), the performance overhead incurred due to this dynamic configuration is negligible.

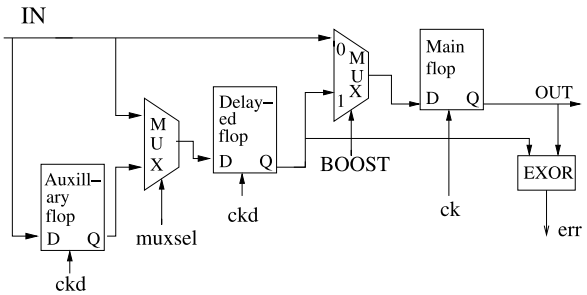


Fig. 8.18 The *look ahead stage* at the switch input

8.6 Experimental Results

In this section, we present the simulation case studies for the *T-error* designs.

8.6.1 Simulation Platform

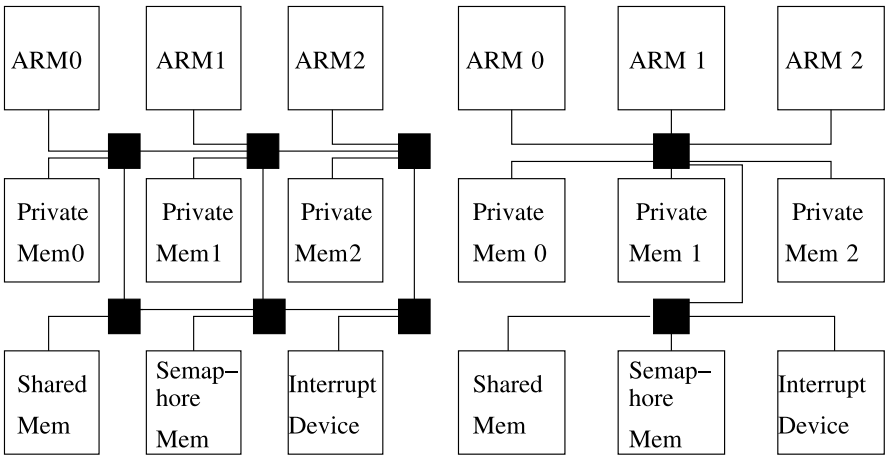
The simulation platform consists of cycle-accurate SystemC models of the *T-error* designs for the switches, links, and NIs, incorporated on the ×pipes architecture. Functional SystemC simulations were carried out on a variety of application benchmarks.

8.6.2 Experiments on a Multi-Media Benchmark

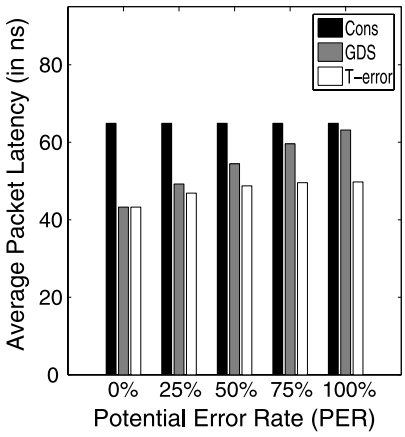
We plugged 3 ARM7 processors, 3 private memories (one for each processor), and 3 shared memories for interprocessor communication on the MPARM platform. We ran functional benchmarks modeling multimedia processing on the general purpose cores. The benchmarks include heavy synchronization activity through the shared memories, since they model producer/consumer pipelines of multimedia processing. The benchmarks create a large number of connections (around 30) between the various cores. We hand-mapped the application onto two topologies (Figures 8.19(a) and (b)): a 3×2 mesh topology, with the processors connected to their private memories using a single switch, and a custom topology with 2 switches. The mappings were performed such that the most demanding traffic flows traverse fewer switches in the NoC.

We assume the size of each predesigned processor and memory core to be 2×2 mm, typical of today's small processors and on-chip memories. From the approximate floorplans of the topologies, we conservatively assume that the links of the mesh topology have 1 pipeline stage, while those of the custom topology have 2 pipeline stages.

We perform experiments on 3 schemes: a traditional *CONServative* (*CONS*) design approach, a *General Double-Sampling* (*GDS*) scheme that is not integrated with the network flow control (such as presented in the earlier works [113] and the *T-error* scheme with 3-stage FIFO presented in this work. From synthesis of the original ×pipes architecture, the conservative NoC's maximum operating frequency is found to be 1 GHz. With 50% delay between the clocks to the *main* and *delayed flip-flops*, the *GDS* and *T-error* designs' maximum frequency (under *over-clocked mode*) is assumed to be 1.5 GHz. To evaluate the designs, we define a new metric: *Potential Error-Rate* (*PER*). The *PER* represents the percentage chance that a flit reaching a FIFO incurs one or more timing errors if sampled directly on a *ck* edge. Note that even if the *PER* is 100%, the actual errors happening at the *T-error* FIFO can be very few, as most of the errors after the first are automatically avoided by



(a) Mesh Topology (b) Custom Topology



(c) Mesh Topology Results

Fig. 8.19 Mesh and custom topology mappings and comparison of traditional schemes with *T-error*

the design. This is because in most scenarios, data is sampled first by the *delayed flip-flop* and only afterward sent out by the *main flip-flop*, avoiding all potential errors. For an *over-clocked* system, the *PER* value depends on how much the system is *over-clocked*, the actual operating conditions of the system (such as effect of process variations on the FIFO, operating temperature, other noise effects), actual data patterns on the link, etc. As an example, if bus encoding techniques are not used to reduce the effects of capacitive cross-talk, the conservative design is capable of operating with the worst-case data patterns on the links. In such a case, even at the highest frequency in the *over-clocked* mode, if the adversarial switching patterns do

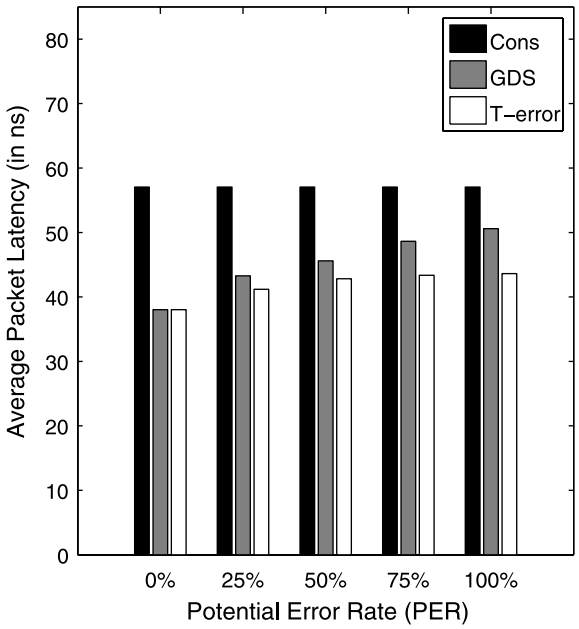


Fig. 8.20 Custom topology results

not occur on the link, the *PER* can be 0%. The *T-error* design dynamically adapts to all these effects and operates under the entire range of *PER* values. For simulations, we vary the *PER* values and we inject potential errors at each *T-error* FIFO randomly based on the chosen *PER* value.

The average packet latency for the mesh and custom topologies for the various schemes for different PER values are presented in Figures 8.19(c) and 8.20. As we *over-clock* only the communication architecture, we compare the schemes based on the average packet latency for communication, instead of comparing the total application run-time. When compared to the traditional conservative design (CONS), the *T-error* design results in significant performance improvements. Latency is reduced by 33.33% in the best case (0% *PER*) and by 23.42% in the worst case (100% *PER*). When compared to the general double sampling scheme (GDS), the *T-error* scheme still shows up to 21.2% reduction in latency, as much of the error recover penalty is hidden under the network operation. When compared to the GDS technique applied to input-queued switches, the *T-error* scheme (with 3-stage FIFOs at the links) also results in 30% reduction in the number of queuing buffers used. In fact, the 3-entry *T-error* FIFO scheme utilizes $3 \times (N-1)$ buffers on each link (where N is the number of cycles needed to traverse the link) and 2 buffers at the switch input, while the input queued switches with the general double sampling technique needs $2N+1$ buffers at the input of the switch and $2 \times (N-1)$ buffers on the links (refer to Section 8.2, where results for 2-entry FIFOs are presented).

To see the impact of the length of the links on the *T-error* scheme, we simulated the design mapped onto the custom topology with varying number of pipeline stages on the links. As seen from Figure 8.21, even on significantly long links, the

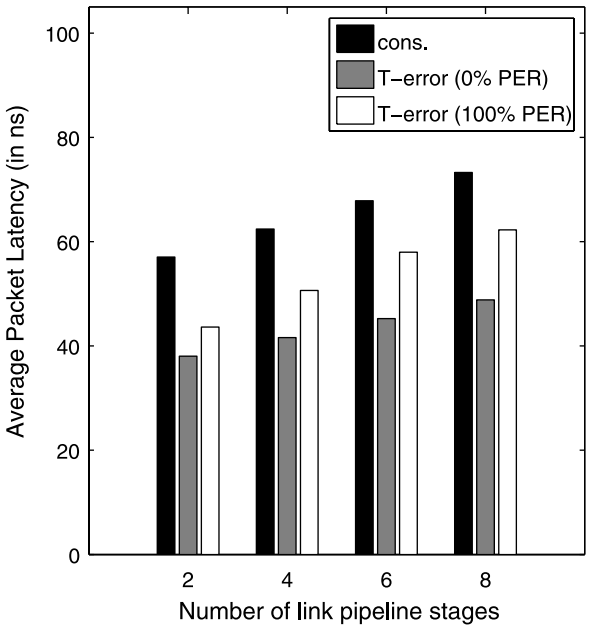


Fig. 8.21 Effect of pipeline depth

T-error scheme gives a large improvement in performance when compared to the conservative design approach.

8.6.3 Effect of Application-Level Power Management

We conducted experiments on the multimedia benchmarks to show the usefulness of the application-level DPM policies. We model 4 different application scenarios in the platform: *Standard Definition* video decoding and display (*SD*), *High Definition* video decoding and display (*HD*), *Picture-in-Picture Standard Definition* (*PiP-SD*), and *Picture-in-Picture High Definition* (*PiP-HD*). The voltage and frequency of operation of the network was tuned individually for each application. The power consumption of the network for the various applications when the DPM policies are used, normalized with respect to that of the base system (where no DPM policy is used), is presented in Figure 8.22. The use of application level DPM policies results in an average of 57% reduction in power consumption of the NoC.

8.6.4 Experiments on Other Benchmarks

We performed experiments on the conservative and *T-error* designs on several other benchmarks:

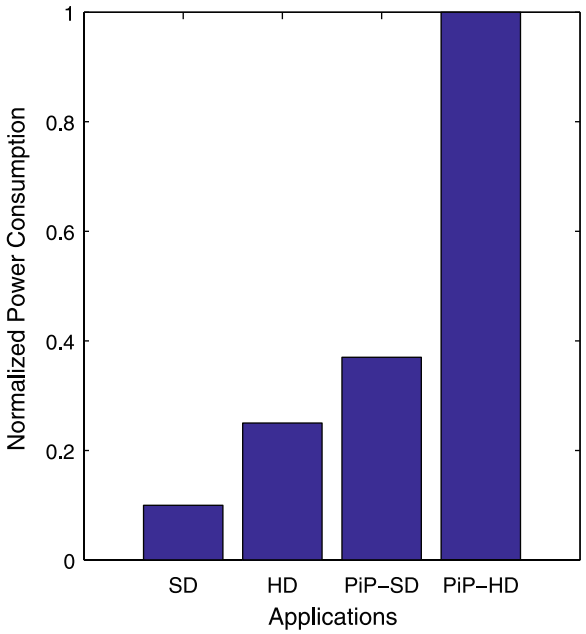
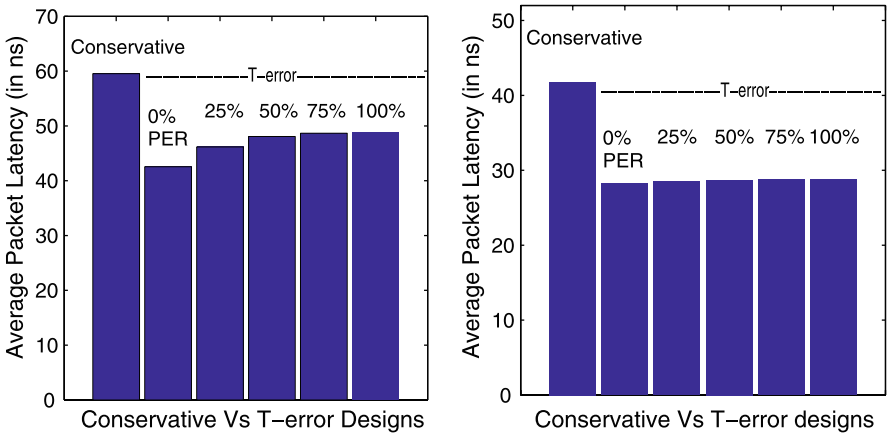


Fig. 8.22 Effect of DPM policies

- Matrix multiplication benchmark suite without shared memory (*MAT1*)
- Matrix multiplication benchmark suite with shared memory (*MAT2*)
- Fast Fourier transform benchmark suite using fixed point arithmetic (*FFT*)
- Quick sort benchmark suite (*Qsort*)

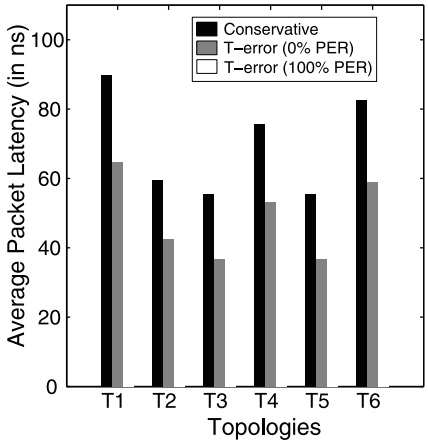
Many of these benchmarks are application kernels that can be used to inject different traffic rates onto the NoC and test various aspects of the NoC. We assume the delay to traverse the links in the NoC to be 2 cycles, i.e., the links have 2 pipeline stages. We conducted experiments varying the number of processor/memory cores used by the applications (application partitioning) and topologies of the NoC. For all the experiments, except for those presented in Section 8.6.6, we use the 3-entry *T-error* FIFO design. In Section 8.6.6, we compare the performance of the two *T-error* link designs.

In Figures 8.23(a) and (b), the average packet latency (in ns) observed for the conservative and *T-error* design for the *MAT2* benchmark for read (Figure 8.23(a)) and write transactions (Figure 8.23(b)) is presented. The read transactions require two way data transfer on the network: a request is sent by the processor and a response with the data item is sent back by the memory. The write transactions require only one way data transfer: the processor sends the data to be written to the memory. We denote the entire transaction latency for each data word by the average packet latency metric. Thus, the read transactions incur a higher latency for communication. As seen from the figures, for the *MAT2* benchmark, the *T-error* design results in a significant performance improvement, with the best case of 28.5% reduction in read latency (for 0% *PER*) and worst-case of 19.6% (for 100% *PER*).



(a) MAT2: read transactions

(b) MAT2: write transactions

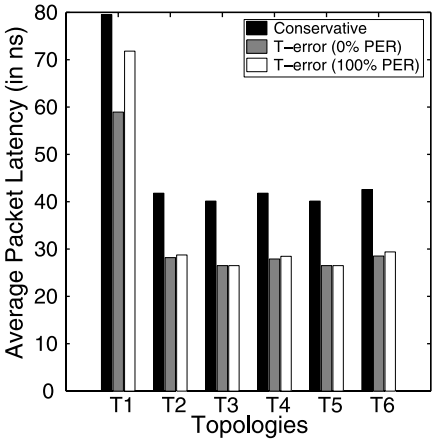


(c) Topology Effects: read transactions

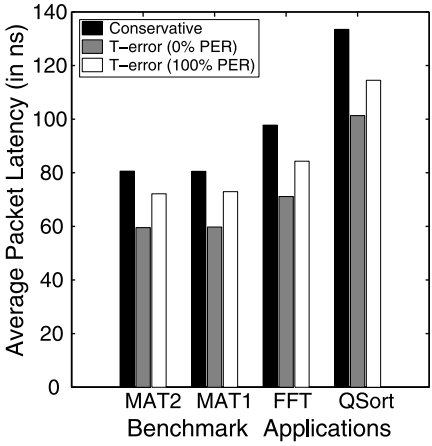
Fig. 8.23 Performance comparison of conservative and *T-error* designs for different *PER* values for read and write transactions

For the write transactions, the average reduction in latency for the *T-error* designs vary from 32.5% (for 0% *PER*) to 31.1% (for 100% *PER*). Note that the increase in latency due to the higher PER values is not overly significant, showing that the *T-error* scheme effectively hides much of the error recovery penalty under the network operation.

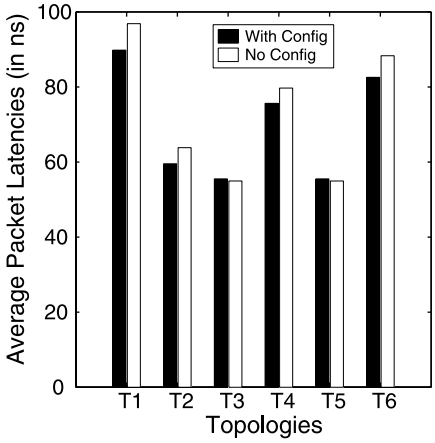
The performance of the *T-error* system for various topologies for the *MAT2* benchmark for read and write transactions are presented in Figures 8.23(a) and 8.24(a). The designs compared vary from small 7-core NoCs to 51-core NoCs with different application partitioning. The topologies vary from regular (like *mesh*)



(a) Topology Effects: write transactions



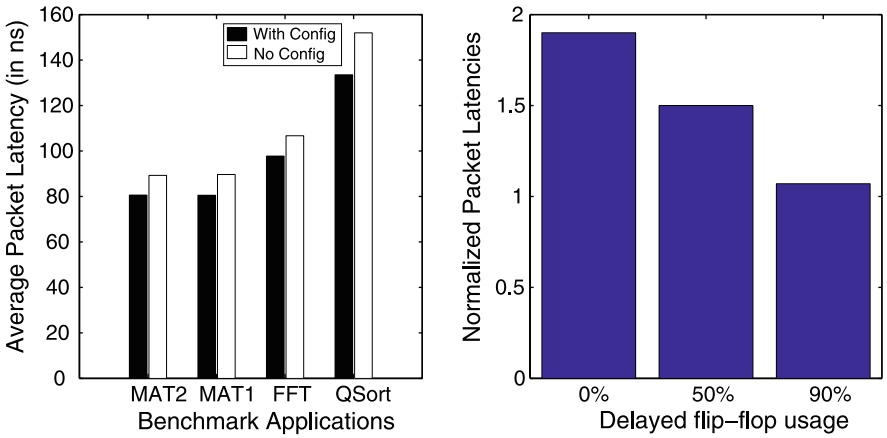
(b) Benchmark Effects: read and write transactions



(c) Topology Effects: NoC Configuration

Fig. 8.24 (**a**) and (**b**) Performance comparison for various topologies, benchmarks, and (**c**) Effect of dynamic NoC configuration

to custom, manually developed ones. As seen from the figures, for all the topologies for both read and write transactions, the *T-error* design results in significant performance improvement over the conservative design. In Figure 8.24(b), we present the average packet latencies (averaged across both read and write transactions) for the designs for several benchmark applications. The average reduction in latency for the benchmarks for the *T-error* designs varies from 25.7% (for 0% *PER*) to 12.7% (for 100% *PER*).



(a) Benchmark Effects: NoC Configuration

(b) Choice of Link Design Schemes

Fig. 8.25 (**a**) NoC configuration and (**b**) Choice of link design schemes

8.6.5 Effect of NoC Configuration

Dynamic configuration of the NoC is designed to avoid any latency penalty for the switch look-ahead mechanism under the *normal mode*, where the frequency of operation is $\leq$1 GHz. In Figures 8.24(c) and 8.25(a), we present the packet latencies for the NoC with and without the configuration mechanism for various topologies and benchmarks. The configuration mechanism results in significant reduction in packet latency (up to 13.8%) for the applications. This reduction is attributed to two reasons: one is the reduction in pipeline depth of the NoC (i.e., reduction in the number of cycles needed to transfer a packet under zero load conditions) and the other is the fact that congestion in the NoC reduces, as packets spend less time in the network.

8.6.6 Choice of Link Design Schemes

In Section 8.3, we presented two link design schemes with *scheme 1* having very little hardware overhead and *scheme 2* having higher performance. The efficiency of the schemes depends on the congestion levels in the NoC and the application's traffic patterns. For heavily congested NoCs, most of the traffic would be sampled through the *delayed flip-flops* in both schemes, resulting in similar performance. For uncongested networks supporting bursty application traffic, *scheme 2* has much higher performance than the *scheme 1* design. These effects are illustrated in Figure 8.25(b), where the average packet latencies in a mesh network using *scheme 1* design are presented. The latency values are normalized with respect to the latency

Table 8.1 Area overhead

Design	Area (mm^2)
Base NoC	4.90
T-error scheme 1 NoC	4.95
T-error scheme 2 NoC	5.10

incurred by the *scheme 2* design for an uncongested NoC. The traffic pattern is such that each core injects bursty traffic onto the NoC. For such a bursty traffic pattern, *scheme 2* design has minimum overhead for all congestion levels, while the performance of the *scheme 1* design depends on the particular congestion level. We varied the congestion in the network, which is represented in Figure 8.25(b) by the percentage of time data is sampled by the *delayed flip-flop*. As seen, as the congestion in the network starts to increase, the performance of *scheme 1* design approaches that of the *scheme 2* design. The different link design schemes can be used in different parts of the same NoC if needed, as they have the same interface to the switches/NIs. Thus, particular links that need higher performance can be designed using *scheme 2*.

8.6.7 Synthesis Results

Using Synopsys Design Compiler, we synthesized the *T-error* schemes to get area estimates of the proposed schemes. For synthesis, we use a UMC 0.13 μm technology library, a base NoC operating frequency of 1 GHz and an operating voltage of 1.2 V. Table 8.1 shows the area overhead for the different *T-error* schemes for 32-bit flit-size for a 5×5 mesh NoC. The base NoC area is the sum of the areas of switches, links, and NIs without the *T-error* design changes. As seen from the table, the schemes incur only a modest increase in area (around 4% increase in the base NoC area).

8.7 Summary

The use of conservative methods to design NoCs, that target *safe* operation under all conditions leads to suboptimal system performance. In this chapter, we have presented aggressive *Timing Error-Tolerant* (*T-error*) design methodologies for designing the switches, links, and NIs of NoCs. The NoC in the *T-error* system is designed aggressively to operate at frequencies higher than conservative designs and to recover from the resulting timing errors in an efficient manner. The error recovery mechanism is integrated with a new link-based flow control mechanism, so that most of the error recovery penalty is hidden under the network operation. Experiments show large performance improvements (up to $1.5\times$) for the communication architecture in the proposed system, when compared to traditional conservative designs. The methods are also applicable to remove timing errors in conservative designs.

Chapter 9
Analysis of NoC Error Recovery Schemes

Once the NoC components are made timing-error tolerant, we need to still handle other transient and permanent errors that can occur in the system, such as soft-errors. To handle such errors, we need support at the design level, as well as at the architectural level. In this chapter,[1] we present architectural level support for fault-tolerance, while in the next chapter, we present design level support. Please note that an additional level of error protection at the application level can also be used in conjunction with these two levels.

In order to protect the system from transient errors that occur in the communication sub-system, we can use error detection/correction mechanisms that are used in traditional macronetworks. The error detection/correction schemes can be based on end-to-end flow control (network level) or switch-to-switch flow control (link-level). In a simple retransmission scheme, error detection codes (*parity* or *Cyclic Redundancy Check* (*CRC*) *codes*) can be added to the original data by the sender and the receiver can check for the correctness of the received data. If an error is detected, it can request the sender to retransmit the data.

Alternatively, error correcting codes (such as *Hamming codes*) can be added to the data and errors can be corrected at the receiver. Hybrid schemes with combined retransmission and error correction capabilities can also be envisioned. As the error detection/correction capability, area-power overhead and performance of the various error detection/correction schemes differ, the choice of the error recovery scheme for an application involves multiple power-performance-reliability trade-offs that have to be explored.

In this work, we collectively relate these three major design constraints in an attempt to characterize efficient error recovery mechanisms for the NoC design environment. We explore error control mechanisms at the data link and network layers and present the architectural details of the schemes. We investigate the energy efficiency, error protection efficiency, and impact on performance of various error recovery mechanisms.

The objective of the work presented in this chapter is twofold: One is to identify the major power overhead issues of various error recovery schemes, so that efficient mechanisms can be designed to address them. The other objective is to provide the designer with the necessary information, aiding in the choice of appropriate error control mechanism for the targeted application. In practice, different network architectures (topologies, switch architecture, routing, flow control) exist, making generalized quantitative comparisons difficult. Nevertheless, this work presents a

[1]We would like to acknowledge the contributions of Dr. Theocharis Theocharides, Prof. N. VijayKrishnan, Prof. Mary J. Irwin, Prof. Luca Benini, and Prof. Giovanni De Micheli.

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

general methodology and attempts to provide comparisons based on reasonable assumptions on network architecture, incorporating features that have been successful in most existing NoC design methodologies. In this work, we explore error control mechanisms at the data link and network layers and investigate the energy efficiency, error protection efficiency, and impact on performance of the various schemes.

9.1 Switch Architecture Design

We identify 3 different classes of error recovery schemes as explained in the following subsections.

9.1.1 End-to-End Error Detection

In the *end-to-end error detection* (`ee`) scheme, parity (`ee-par`) or CRC codes (`ee-crc`) are added to the packet (refer to Figure 9.1(a)). A CRC or parity encoder is added to the sender NI and decoders are added at the receiver NI. The sender NI has one or more packet buffers in which it stores the packets that are transmitted. The receiver NI sends a *NACK* or an *ACK* signal back to the sender, depending on whether the data had an error or not. The *ACK/NACK* signal can be piggy-backed on the response packet, if this is a request-response transaction (as in Open Core

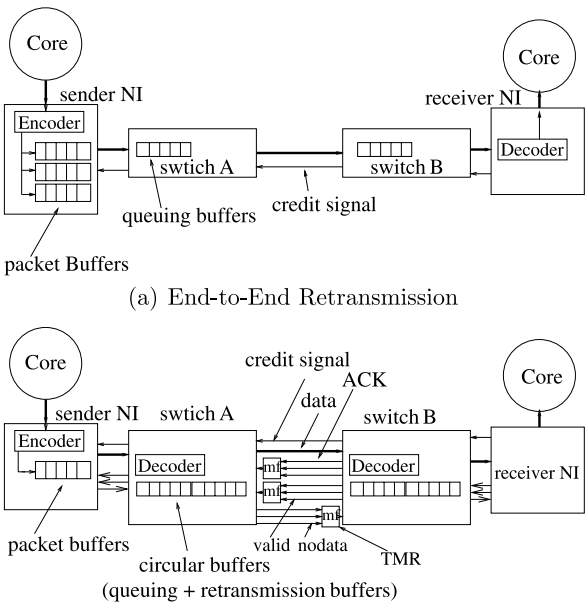


Fig. 9.1 Architecture for end-to-end and switch-to-switch retransmission

Protocol [91]). To account for errors on the *ACK*/*NACK* packets, we also have a time-out mechanism for retransmission at the sender. We use sequence identifiers for packets to detect reception of duplicate packets. As header flit carries critical information (like routing information), it is protected with parity/CRC codes that are checked at each hop traversal. If a switch detects an error on the header flit of a packet, then it drops the packet. Also, the flit-type (that identifies header, body or tail flit) bits are protected using redundancy.

9.1.2 Switch-to-Switch Error Detection

In switch-to-switch error detection schemes, the error detection hardware is added at each switch input and retransmission of data is between adjacent switches. Here, we identify two different schemes: parity/CRC at flit-level and at packet level. The switch architecture is modified to support these schemes, as shown in Figure 9.1(b). The additional buffers added at each input of the switch are used to store packets until an *ACK*/*NACK* comes from the next switch/NI. The number of buffers needed to support switch-to-switch retransmission depends on whether error detection is done at the packet level or flit level.

In the *switch-to-switch flit-level error detection* (`ssf`) scheme, the parity/CRC bits are added to each flit of the packet by the sender NI. At each input of the switch, there are two set of buffers: *queuing buffers* that are used for the *credit-based flow control* as in the base switch architecture and *retransmission buffers* for supporting switch-to-switch retransmission mechanism. Similar to the case of queuing buffers, the retransmission buffers at each switch input should have a capacity of $2N_l + 1$ flits for full throughput operation. We use redundancy (such as *Triple Modular Redundancy* (*TMR*)) to handle errors on the control lines (such as the *ACK* line).

In the *packet level error detection* (`ssp`) scheme, the parity/CRC bits are added to the tail flit of the packet. For this scheme, the number of retransmission buffers needed at each switch input is $2N_l + f$, where f is the number of flits in the packet, as the error checking is done only when the tail flit reaches the next switch. We also need header flit protection, as in the `ee` scheme.

9.1.3 Hybrid Single Error Correcting, Multiple Error Detecting Scheme

In this scheme (`ec+ed`), the receiver corrects any single bit error on a flit, but for multiple bit errors, it requests end-to-end retransmission of data from the sender NI. We do not consider pure error correcting schemes in this work, as in such a scheme when a packet is dropped by a switch (due to errors in the header flit), it is difficult to recover from the situation as there is no mechanism for the sender to retransmit the packet.

9.2 Energy Estimation and Models

9.2.1 Energy Estimation

A generic energy estimation model in [148] relates the energy consumption of each packet to the number of hop traversals and the energy consumed by the packet at each hop. We expanded this estimation a step further by designing and characterizing the circuit schematics of individual components of the switch in 70 nm technology using Berkeley Predictive Technology Model [125]. From this, we estimated the average dynamic power as well as the leakage power per flit per component. We imported these values into the architectural level cycle-accurate NoC simulator and simulated all individual components in unison to estimate both dynamic and leakage power in routing a flit.

For correct functionality of the system, the error detection/correction circuitry and the retransmission buffers need to be error-free. We use relaxed scaling rules and soft-error tolerant design methodologies for designing these components [146]. In the power estimations, we take into account the additional overhead incurred in making these components error-free (which increases the power consumption of these components by around 8–10%).

9.2.2 Error Models

In order to analyze the error recovery schemes, we fix a constraint on the residual flit error probability, that is we impose each scheme to have the same probability of an undetected error (per flit) at the decoder side. We assume that an undetected error in the system causes the system to crash.

We consider two set of experiments: in one set of experiments we assume the operating voltage of the system (with different error recovery schemes) is varied to match a certain residual flit-error rate requirement. For this, we make use of the error models from [127]. In another set of experiments, we assume the voltage for the various schemes to be the same, but investigate the effect of different error rates on the schemes.

9.3 Experiments and Simulation Results

9.3.1 Power Consumption of Schemes for Fixed Residual Error Rates

In this subsection, we assume that the power supply voltage is chosen for each of the error detection/correction schemes based on the residual flit-error rate that the

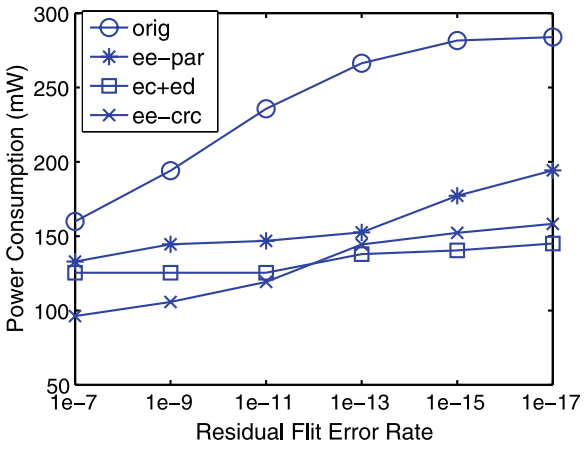


Fig. 9.2 Power consumption of schemes

system needs to support. We compare the power consumption of systems with parity based encoding, CRC based encoding and hybrid single error correcting multiple error detecting encoding with that of the original system (without error protection codes). As the objective is to compare the error protection efficiency of various coding schemes, we consider only end-to-end schemes in this subsection. We consider a 4×4 mesh network with 16 cores and 16 switches. We assume the number of flits in a packet to be 4 and the flit-size to be 64 bits. The network power consumption for the various schemes are presented in Figure 9.2, for an injection rate of 0.2 flits/cycle from each core and for uniform traffic pattern. The residual flit-error rates in the x-axis represent the Mean Time To Failure (MTTF) for the systems. As an example, a residual flit-error rate of 10^{-12} signifies that on average the system operates for 3.12^{11} cycles (assuming 16 cores, with each core injecting 0.2 flits/cycle, so that 10^{12} flits are generated in 3.12^{11} cycles), before an undetected error causes the system to crash. For a 200 MHz system, this represents an MTTF of 26 minutes. Note that for most applications reasonable MTTF values would be of the order of months or years. The power numbers are plotted for the original system (`orig`) that has no error control circuitry, parity-based end-to-end error detection scheme (`ee-par`), CRC based error detection scheme (`ee-crc`) and hybrid single error correcting, multiple error detecting scheme (`ec+ed`).

The `orig` and `ee-par` schemes have higher power consumption than the `ee-crc` and `ec+ed` schemes, as the error detection capability of these schemes is lower and hence they require a higher operating voltage to achieve the same residual flit-error rate. The hybrid `ec+ed` scheme has lower power consumption at high residual flit error rates and the `ee-crc` has lower power consumption for lower residual error rates. This is because at high error rates, in the `ee-crc` scheme there is more traffic injected in the network, thereby causing more power consumption than the `ec+ed` scheme. At lower error rates, the power overhead due to error correction in the `ec+ed` scheme is more than the power consumed in retransmission in the `ee-crc` scheme. Also, in the `ec+ed` scheme, the number of bits needed for error correction and detection codes is more than the pure detection scheme.

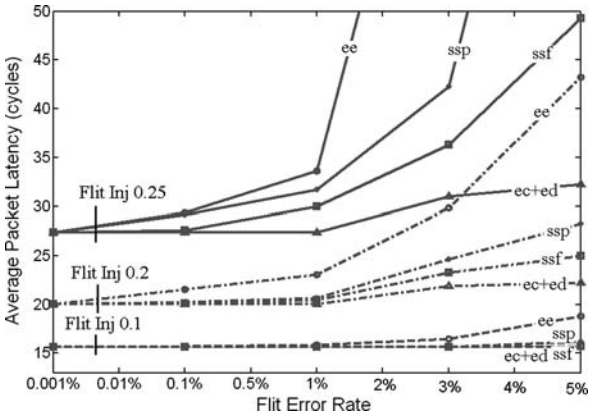


Fig. 9.3 Latency of error detection and correction schemes

9.3.2 Performance Comparison of Reliability Schemes

In this subsection, we investigate the performance of pure end-to-end and switch-to-switch error detection schemes (`ee`, `ssf`, `ssp`) and the hybrid error detection/correction scheme (`ec+ed`). We perform experiments on the 16-core mesh with varying injection rates for uniform traffic pattern. In this and following experiments, we assume that the operating voltage for the system is fixed at design time (to be equal to 0.85 V) and investigate the effect of varying error rates in the system. We use the flit-error rate (flit error rate is defined as the probability of one or more errors occurring in a flit) metric for defining the error rate of the system.

For low flit-error rate and low injection rate, the average packet latency for the various schemes (Figure 9.3) are almost same. However, as the error rate and/or the flit injection rate increases, the end-to-end retransmission scheme (`ee`) incurs a large latency penalty compared to the other schemes. The packet-based switch-to-switch retransmission scheme (`ssp`) has slightly higher packet latency than the flit-based switch-to-switch retransmission scheme (`ssf`), as in the flit-based scheme errors on packets are detected earlier. As expected, the hybrid single error correcting multiple error detecting scheme (`ec+ed`) has the least average packet latency of the schemes.

9.3.3 Power Consumption Overhead of Reliability Schemes

The power consumption of a switch (with 5 inputs, 5 outputs, $N_l = 2$), error detection/correction coders, retransmission, and packet buffers (for 50% switching activity at each component, each cycle) are presented in Table 9.1. We assume an operating frequency of 200 MHz, flit-size of 64 bits and packet size of 4 flits. In this chapter, we assume that the base NI power consumption (when there are no packet buffers for retransmission) is taken to be part of the processor/memory core power consumption, as it is invariant for all the schemes. To facilitate comparison of

Table 9.1 Component-wise power consumption

Component	Dynamic power (mW)	Static power (mW)
Switch (5×5)		
Buffers	13.10	1.69
Crossbar	4.57	–
Control	1.87	0.02
Total (P_{sw})	19.54	1.71
CRC encoder (P_{crce})	0.12	–
CRC decoder (P_{crcd})	0.15	–
SEC encoder (P_{sece})	0.15	–
SEC decoder (P_{secd})	0.22	–
Switch retrans.	0.52	0.07
Flit buffer (1 flit) (P_{srfb})		
Packet buffer (1 packet) (P_{pb})	2.29	0.31

the various error recovery schemes, we analyze the power overhead associated with the schemes for error detection and recovery. We need the following definitions to formulate analytical expressions for the power overhead for the schemes.

Let *inj_rate* be the traffic injected by each of the NI. For the `ee`, `ec+ed` schemes, let the number of packet buffers required at each NI for retransmission be N_{pb}. Let *sw_traf* be the rate of traffic injected at each switch. For the `ee` scheme, let the increase in traffic at each switch due to retransmission be represented by *sw_incrtraf*. Let $P_{packetsizeinc}$ be the total power overhead due to increase in packet size due to addition of code words and other control bits.

In the formulation of the power overhead, for simplicity of notation, we represent the parameters (such as traffic rate, link length, buffering) to be same for all the NIs and all the switches. Also, for simplicity of notation we represent both dynamic and static power consumption by single set of variables (refer Table 9.1 for notations). It is assumed that when the power numbers are scaled based on the traffic through the components, only the component of dynamic power consumption is scaled.

In the above set of parameters, the traffic rates from/to the NIs, switches and the traffic overhead for retransmission (in `ee`, `ec+ed` schemes) are obtained from simulations. The link lengths are decided by the physical implementation of the topology. The number of packet buffers required in the `ee` scheme to support an application performance level can be obtained from (possibly multiple sets of) simulations.

The power overhead associated with the `ee` scheme is given by:

$$P_{overhead\_ee} = \sum_{\forall\ NIs} \left(inj\_rate \times (P_{crce} + P_{crcd} + N_{pb} \times P_{pb})\right) + \sum_{\forall\ Switches} (sw\_incrtraf \times P_{sw}) + P_{packetsizeinc} \tag{9.1}$$

In this equation, there are two major components of power overhead: One is the power overhead associated with the packet buffers at the NIs for retransmission and the other is due to the increase in power consumption due to increased network traffic. For the `ee` scheme to work, we need to have sequence identifiers for packets and mechanisms to detect reception of duplicate packets. We consider the power consumption due to look-up tables and control circuitry associated with these mechanisms to be part of the packet buffer power consumption (these typically increase packet buffer power overhead by 10%). The increase in traffic in the `ee` scheme is due to two reasons:

(a) when *ACK/NACK* packets cannot be piggy-backed to the source (as an example, "writes" to memory locations normally do not require response back to the source), they need to be sent as separate packets. An optimization can be performed in this case, as the *ACK/NACK* packet needs to be only one flit long. Even with this optimization, we found that total power consumption increases by 10–15% due to this overhead.

(b) at higher error rates, the network traffic increases due to retransmission of packets. However, even at flit error rates of 1%, we found that this increase has much lower impact than the above case.

As the $P_{packetsizeinc}$ affects the schemes almost in a similar manner (as the `ssf` needs code bits on each flit, while the `ee` scheme needs additional information for packet identification, header flit protection, and packet code words) this has lesser effect on deciding the choice of scheme.

The power overhead of the `ssf` scheme is represented by:

$$P_{overhead\_ssf} = \sum_{\forall\ NIs} (inj\_rate \times P_{crce}) + \sum_{\forall\ switches} \left(sw\_traf \times \left((2N_l + 1) \times P_{srfb} + P_{crcd}\right)\right) + P_{packetsizeinc} \tag{9.2}$$

The power consumption of the switch retransmission buffers is the major component of the overhead, and it depends linearly on the link lengths. The power overhead of (`ssp`) and `ec+ed` schemes can be easily derived from the overhead equations for `ssf` and `ee` schemes, respectively.

The network power consumption for the various error recovery schemes for the 16-core mesh network is presented in Figure 9.4. We assumed the link lengths to be 2 cycles long. We performed simulations with uniform traffic pattern, with each core injecting 0.1 flits/cycle. For the `ee` and `ec+ed` schemes, the number of packet retransmission buffers needed to support the application performance level were obtained from simulations (which turned out to be 2 packet buffers/NI). For this experiment, we observe that the power consumption of switch-based error detection schemes (`ssf`, `ssp`) is higher than end-to-end retransmission schemes (`ee`, `ec+ed`). This is attributed to two factors: (a) the switch buffering needed for retransmission in `ssf`, `ssp` schemes for this set-up is large compared to the packet buffering needs of the `ee`, `ec+ed` schemes (b) due to uniform traffic pattern, the

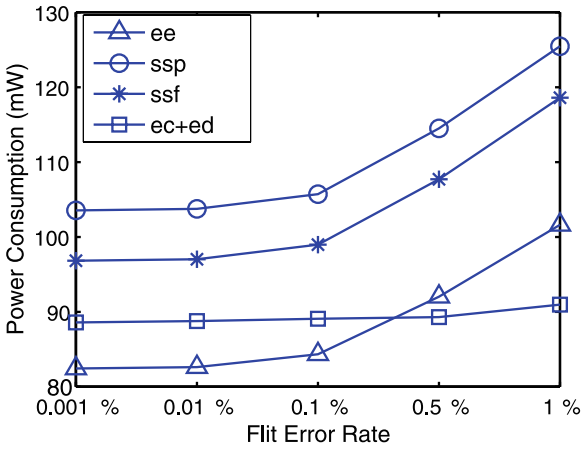


Fig. 9.4 Power consumption of error recovery schemes (0.1 flits/cycle)

traffic through each switch is more (as the average number of hops is more), thus increasing `ssf` and `ssp` retransmission overhead. We examine these two points in detail in the following subsection.

9.3.4 *Effect of Buffering Requirements, Traffic Patterns and Packet Size*

One of the major power overheads for the schemes is the amount of packet and switch buffering needed for retransmission. To see the impact of buffering requirements, we performed experiments on the mesh network, varying the number of packet buffers and link lengths (and hence the number of retransmission buffers for `ssf` scheme). The results are presented in Tables 9.2 and 9.3. For small link lengths and when the packet buffering requirements of the `ee` scheme is large, the `ssf` scheme is more power efficient than the `ee` scheme. On the other hand, when the link lengths are large, `ee` scheme is more power efficient. But in the realm where link lengths are short and packet buffering needs are small, it is difficult to make generalization on the efficiency of the schemes. However, if the parameters (such as link length, packet buffering needs, etc.) are obtained from user input and simulations, they can be fed into the above methodology to compare the error recovery schemes.

Another important parameter that affects the choice of the schemes is the application traffic characteristics. To see the impact of various traffic scenarios, we performed experiments varying the average hop delay for data transfer. The power overhead of the `ee` and `ssf` schemes (assuming $N_{pb} = 2$, $N_l = 2$) for the different scenarios is shown in Figure 9.5. In the figure, average hop count of 2 corresponds to neighbor traffic pattern and other hop delay numbers can be interpreted as representing other traffic patterns. As the average hop count for data transfer increases, the power overhead of `ssf` increases rapidly, as more traffic passes through each

Table 9.2 Packet buffers, with $N_l = 2$

N_{pb}	ee power (mW)
1	76
2	84
3	93
4	102
5	111
6	120

Table 9.3 Link length

N_l (cycles)	ee, $N_{pb} = 2$ power (mW)	ssf power (mW)
1	65.1	59.2
2	84.0	97.0
3	102.8	134.8
4	121.8	172.5
5	141.2	216.5

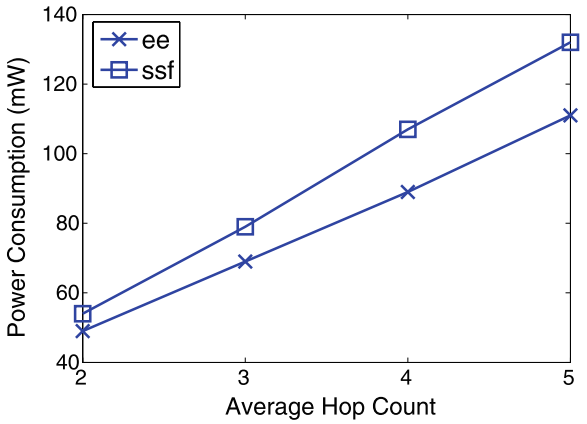


Fig. 9.5 Effect of hop count

switch, thereby consuming more power on the switch retransmission buffers. Thus, for traffic flows that traverses longer number of hops or when the network size is large, switch-to-switch retransmission schemes incur a large power penalty.

The power consumption of the flit-based (ssf) and packet-based (ssp) schemes for varying number of flits/packet is presented in Figure 9.6. In this experiment, we assume that the packet size is kept constant (256 bits) and we vary the number of flits/packet. As the number of flits/packet increases, the buffering needs of the packet-based scheme increases, hence the power consumption of the packet based scheme increases rapidly. The flit-based scheme also incurs more power consump-

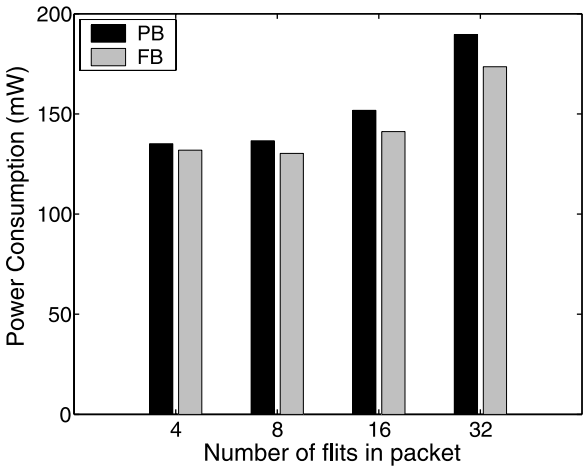


Fig. 9.6 Flit vs. packet schemes

Table 9.4 NoC area

Scheme	Area mm^2
`orig`	3.36
`ee`	4.41
`ssf`	5.76
`ec+ed`	5.32

tion with increasing flits/packet as the ratio of useful bits to overhead bits (i.e., the CRC code bits) decreases as flits/packet increases. However, for reasonable flit-sizes, we found that flit-based scheme is more power efficient than the packet based scheme.

The area of network components (of switches and the additional hardware for error recovery) for various schemes for the 16-node mesh network (with $N_{pb} = 2$ and $N_l = 2$) is presented in Table 9.4. The area overhead of the schemes are comparable.

9.4 Summary

For the `ee` and `ec+ed` schemes, the major components of power overhead are the packet buffering needs at the NIs and the increase in network traffic due to *ACK/NACK* packets. For the `ssf` and `ssp` schemes, the major power overhead is due to the retransmission buffers needed at the switches. Design methodologies that trade-off application performance for the buffering needs would result in smaller power overhead. Methods from queuing theory can be explored to design these buffers. Methods that reduce the *ACK/NACK* traffic (such as multiple packets sharing a single *ACK/NACK* signal) would be interesting to explore. Another avenue is to explore mechanisms that reduce the control overhead associated with duplicate packet reception in the `ee` scheme.

From the experiments, we observe that for networks with long link lengths or hop counts, end-to-end detection schemes are power efficient. Switch level detection mechanisms are power-efficient when the link lengths are small and when the end-to-end scheme needs large packet buffering at the NIs. At low error rates, the average latencies incurred in all the schemes are similar. At higher error rates, a hybrid error detection and correction mechanism has higher performance than other schemes. As the `ee` scheme uses a subset of the hardware resources used for the `ec+ed` scheme, depending on the error rates prevailing in the system, the error correction circuitry can be selectively switched on/off. For hierarchical networks, switch based error control can be implemented for local communication and end-to-end error control can be implemented for global communication (that traverses longer links and hop counts).

Chapter 10
Fault-Tolerant Route Generation

In this chapter, we present design level support for handling temporary and permanent errors in the NoCs. We present routing mechanisms that achieve an application-specific reliability level against temporary and permanent failures.

The routing scheme used in the NoC can be either static or dynamic in nature. In static routing, one or more paths are selected for the traffic flows in the NoC at design time.

In the case of dynamic routing, the paths are selected based on the current traffic characteristics of the network. Due to its simplicity and the fact that application traffic can be well characterized for most SoC designs, static routing is widely employed for NoCs [43]. When compared to static single-path routing, the static multipath routing scheme improves path diversity, thereby minimizing network congestion and traffic bottlenecks. When the NoC is predesigned, with the NoC having a fixed operating frequency, data width, and hence bandwidth (bandwidth available on each network link is the product of the link data width and the NoC operating frequency), reducing congestion results in improved network performance.

For most SoC designs, the NoC operating frequency can be set to match the application requirements. In this case, reducing the traffic bottlenecks leads to lower required NoC operating frequency, as traffic is spread evenly in the network, thereby reducing the peak link bandwidth needs. A reduced operating frequency translates to a lower power consumption in the NoC. As an example, consider a MPEG video application mapped onto a 4×3 mesh NoC. Detailed analysis of the application and the performance of traditional single path schemes and the proposed multipath scheme are presented later in this chapter. When the NoC operating frequency for the schemes is set so that both schemes provide the same performance level (same average latency for traffic streams), the multipath scheme results in 35% reduction in network operating frequency, leading to 22.22% reduction in network power consumption (after accounting for the overhead involved in the multipath scheme). Another important property of the multipath routing strategy is that there is spatial redundancy for transporting a packet in the on-chip network. A packet can be sent across multiple paths for achieving resiliency against transient or permanent failures in the network links.

Many of today's NoC architectures are based on static single path routing. This is because with multipath routing, packets can reach the destination in an out-of-order fashion due to the difference in path lengths or due to difference in congestion levels on the paths. For many applications, such out-of-order packet delivery is not acceptable and packet reordering is needed at the receivers. As an example, in chip multiprocessor applications for maintaining coherency and consistency, packets reaching the destination need to be in-order. In video and other multimedia

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

Table 10.1 Re-order buffer overhead

Design	Area (sq mm)	Power (mW)
1. Base NoC	1.04	183.75
2. 2 buffers/core	1.14	201.02
3. 10 buffers/core	1.65	281.25

applications, packet ordering needs to be maintained for displays and for many of the processing blocks in the application.

With multipath routing, packet reorder buffers can be used at the receiver to re-order the arriving packets. However, the reorder buffers have large area and power overhead and deterministically choosing the size of them is infeasible in practice. In Table 10.1, the area and power consumption of a 4×3 mesh NoC (the area-power values include the area-power of the switches, links, Network Interfaces (NIs)) with different number of packet buffers in the receiving NIs is presented. The network operating frequency is assumed to be 500 MHz with 50% switching activity at the subcomponents. The flit size is assumed to be 16 bits, with the base switch/NI having 4 flit queuing buffers at each output. The base NoC component area, power values are obtained from synthesizing the component designs that are based on the architecture presented in Chapter 2. As seen from the table, a NoC design with 10 packet reorder buffers/core has 59% higher NoC area and 43% higher NoC power consumption when compared to the base NoC without reorder buffers. Another important point is that at design time it is not possible to size the reorder buffers to prevent packets from being dropped at the receiver. As an example, if a packet travels a congested route and takes an arbitrarily long time to reach the destination, several subsequent packets that take a different route can reach the destination before this packet. In this case, the reorder buffers, unless they have infinite storage capacity, can be full for a particular scenario and can no longer receive packets. This leads to dropping of packets to recover from the situation and requires end-to-end *ACK*/*NACK* protocols for resuming the transaction.

End-to-end *ACK*/*NACK* protocols are used in most macronetworks for error recovery, and in such networks these protocols are extended to handle this packet buffering problem as well [112]. However, such protocols have significant overhead in terms of network resource usage and congestion. Thus, they are not commonly used in the NoC domain [112, 130]. Moreover, the performance penalty to recover from such a situation can be very high and most applications cannot tolerate such variations in performance. This motivates the need to find efficient solutions to the packet reordering problem for the on-chip domain.

In this chapter, we present a multipath routing strategy with guaranteed in-order packet delivery (without packet dropping) for on-chip networks [19]. It is based on the idea of routing packets on *nonintersecting* paths and re-building packet order at *path reconvergent* nodes. By using an efficient flow control mechanism, the routing strategy avoids the packet dropping situation that arises in the traditional multipath routing schemes. We present algorithms to find the set of paths in a NoC topology

to support the routing strategy and present a method to split the application traffic across the paths to obtain a network with minimum power consumption. We explore the use of temporal and spatial redundancy during multipath routing to provide resilience against temporary and permanent errors in the NoC links. When sending multiple copies of a packet, it is important to achieve the required reliability level for packet delivery with minimum data replication. We integrate reliability constraints in the multipath design methods to provide a reliable NoC operation with the least increase in network traffic.

Experiments on several benchmarks show large power savings for the proposed scheme when compared to traditional single-path schemes and multi-path schemes with reorder buffers. The area overhead of the proposed scheme is small. Hence, it is practical to be used in the on-chip domain.

10.1 Multi-Path Routing with In-Order Delivery

In this section, we present the conceptual idea of the multipath routing strategy with in-order packet delivery. For analysis purposes, we define the NoC topology by the NoC topology graph.

Definition 23 The topology graph is a directed graph $G(V, E)$ with each vertex $v_k \in V$ representing a switch/NI in the topology and the directed link (or edge) $e_l \in E$ representing a direct communication between two switches/NIs. We represent the traffic flow between a pair of cores in the NoC as a commodity i, with the source switch/NI of the commodity being s_i, and the destination of the commodity being d_i. Let the total number of commodities be I. The rate of traffic transferred by commodity i is represented by r_i.

The traffic rate for each commodity (r_i) can either be the average rate of communication between the source and destination of the commodity or can be obtained in an efficient manner that considers the *Quality-of-Service* (*QoS*) provisions for the application. We define the paths for the traffic flow of a commodity as follows.

Definition 24 Let the set SP_i represent the set of all paths for the commodity i, $\forall i \in 1, \ldots, I$. Let P_i^j be an element of SP_i, $\forall j \in 1, \ldots, |SP_i|$. Thus, P_i^j represents a single path from the source to destination for commodity i. Each path P_i^j consists of a set of links.

We define a set of paths to be *nonintersecting* if the paths originate from the same source vertex, but do not intersect each other in the network, except at the destination vertex.Consider packets that are routed on the two *non-intersecting* paths. Note that with worm-hole flow control [94], packets of a commodity on a particular path are in-order at all time instances. However, packets on the two different paths can be out-of-order.

To implement the reordering mechanism at network reconvergent nodes, the following architectural changes to the switches/NIs of the NoC are required (shown in Figure 10.1). We assume that the packet is divided into multiple flow control units called *flits*. The first flit of the packet (known as the *header* flit) has the routing information for the packet. To support multi-path routing, individual packet identifiers are used for packets belonging to a single commodity. At the *reconvergent* switch, we use a look-up table to store the identifier of the next packet to be received for the commodity. Initially (when the NoC is reset), the identifiers in the look-up tables are set to 1 for all the commodities. When packets arrive at the input of the *reconvergent switch*, the identifier of the packet is compared with the corresponding look-up table entry. If the identifiers match, the packet is granted arbitration and the look-up table identifier value for this commodity is incremented by 1. If the identifiers do not match, then this is an out-of-order packet and access to the output is not granted by the arbiter circuit, and it remains at the input buffer.

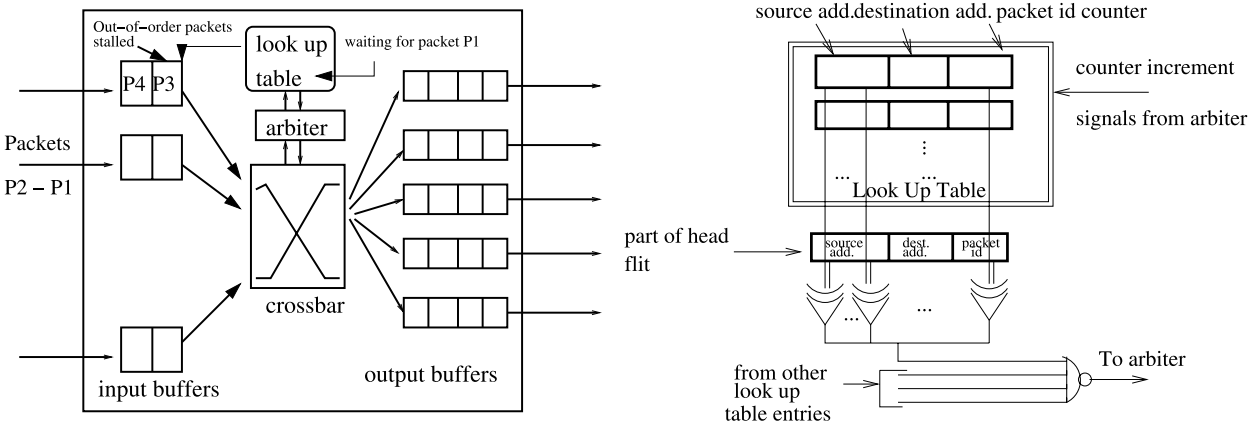
As the packets on a particular path are in-order, the mechanism only stalls packets that would also be out-of-order if they reach the switch. Due to the disjoint property of the paths reaching the switch, the actual packet (matching the identifier on the look-up table) that needs to be received by the switch is on a different path. As a result, such a stalling mechanism (integrated with *credit-based* or *on-off* flow control mechanisms [94]) does not lead to packet dropping, which is encountered in traditional schemes when the reorder buffers at the receivers are full. Note that routing-dependent deadlocks that can occur in the network can be avoided using virtual channel flow control [94].

To avoid deadlocks, when more than one commodity converges at the same switch/NI, we may need to use separate input queues for each commodity. Also, in such a scheme, the waiting packets do not block other packets that are traversing the switch.

10.2 Path Selection Algorithm

In this section, we describe the algorithms that can be used to efficiently find *non-intersecting paths* for each commodity of the NoC. As, in general, the number of paths between a source and destination vertex of a graph is exponential, we present heuristic algorithms to compute the paths [95]. For each commodity, we first find the set of all possible paths for the commodity. Then from the chosen paths, we find those paths that are nonintersecting. We use such a two-phase approach to achieve fast heuristic solutions to tackle the exponential problem complexity.

Consider a source vertex s_i (which corresponds to the source core/NI that sends a packet) and destination vertex d_i of a commodity i. Algorithm 7 is used to find the set of possible paths between the two vertices. The Example 10 presented below illustrates how the working algorithm works. The objective of the algorithm is to find maximum number of paths possible, so that large path diversity is available for the traffic flow. In the algorithm, after finding a path, we remove one of the edges of the path so that the same path is not considered in further iterations. As most NoC



(a) Switch Architecture

(b) Look-up Table: Each re-converging commodity's source address and the identifier of the next packet to be received are stored in the look-up table

Fig. 10.1 Switch Design to support multipath routing with in-order packet delivery

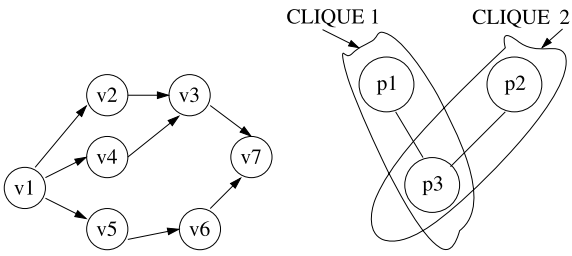


Fig. 10.2 Path selection and compatibility graph generation

(a) Example graph for path selection

(b) Compatibility graph for the example

vertices have only a small degree, we remove one of the middle edges (instead of the edges at the source and destination), because it helps in increasing the number of paths found. As in each iteration of the algorithm we remove an edge, the number of iterations (and hence the maximum number of paths found) is at most $|E|$ for one pair of source and destination vertices.

Example 10 Consider the NoC topology graph presented in Figure 10.2(a). The vertices represent switches/NIs in the NoC. Let v_1 and v_7 be the source and destination vertices of a traffic flow. In the first iteration of the algorithm, one of the paths (e.g., the path $v1 - v2 - v3 - v7$) is chosen and the middle edge (edge from $v2 - v3$) is removed. In the next iteration of the algorithm, another path ($v1 - v4 - v3 - v7$) is chosen and the edge $v4 - v3$ is removed. In the last iteration $v1 - v5 - v6 - v7$ is chosen and the edge $v5 - 6$ is removed, after which no more paths exist from $v1$ to $v7$. Note that if we had removed the edge $v1 - v2$ or $v3 - v7$ in the first iteration, we would have obtained only two paths (instead of three paths).

The paths resulting from the algorithm may converge at one or more vertices. In order to obtain *nonintersecting* paths, we form a *compatibility graph*, with each vertex of the graph representing a path. An edge between two vertices in the graph implies that the corresponding paths do not intersect. An example *compatibility graph* for the paths from Example 1 is shown in Figure 10.2(b). The objective is to obtain the maximum number of nonintersecting paths from the set of paths. This is equivalent to finding the maximum size clique[1] in the *compatibility graph*, which is a well-known NP-Hard problem [95]. We use a commonly used heuristic algorithm for finding the maximum clique (refer Algorithm 8) [18]. The working of the algorithm is illustrated in Example 2. We repeat the two algorithms for all the commodities in the NoC. When applying Algorithm 7 for each commodity, we start with the original topology graph.

[1] Clique of a graph is a fully connected subgraph.

Algorithm 7 Path selection algorithm for a single commodity

1. Choose a path from the source to destination of the commodity using Depth First Search (DFS).
2. Remove one of the middle edges of the chosen path.
3. Repeat the above steps until there are no paths between the vertices.

Algorithm 8 Determining nonintersecting paths for a single commodity

1. Build a compatibility graph for the paths and initialize the set MAX_CLIQUE to NULL.
2. Add vertex with largest degree to MAX_CLIQUE.
3. From remaining vertices, choose vertex that is adjacent to all vertices in set MAX_CLIQUE and add it to the set.
4. Repeat the above step until no more vertex can be added.

Example 11 The compatibility graph for the 3 paths from Example 10 is shown in Figure 10.2(b). The vertex $p1$ represents the path $v1 - v2 - v3 - v7$, $p2$ represents the path $v1 - v4 - v3 - v7$ and $p3$ represents the path $v1 - v5 - v6 - v7$. As the paths $v1 - v2 - v3 - v7$ and $v1 - v5 - v6 - v7$ do not intersect each other, there is an edge between $p1$ and $p3$ in the compatibility graph. Similarly, for the paths $v1 - v4 - v3 - v7$ and $v1 - v5 - v6 - v7$, there is an edge between $p2$ and $p3$. There are two maximum size cliques in the graph (formed by $p1$, $p3$ and $p2$, $p3$) and one of them is arbitrarily chosen (say, $p1$, $p3$). Thus, the paths $v1 - v2 - v3 - v7$ and $v1 - v5 - v6 - v7$ are used for the traffic flow between vertices $v1$ and $v7$.

The time complexity of each iteration in Algorithm 7 is dominated by the *Depth-First Search* (*DFS*) procedure and is $O(|E| + |V|)$ [95]. The number of iterations is $O(|E|)$. The time complexity of the maximum clique calculation step is $O(|E|^2)$. The algorithms are repeated once for each commodity. Therefore, the run time of the nonintersecting path finding algorithms is $O(|I||E|(|E| + |V|))$. In practice, the run-time of the algorithms is less than few minutes for all the experimental studies we have performed.

In cases where we are interested in having as many minimum paths as possible, we can modify the call to DFS in Algorithm 7 to a call to Dijkstra's shortest path algorithm, choosing shorter paths first. Then Algorithm 8 can also be modified to first choose the minimum paths that are nonintersecting and then choosing nonminimum paths that are nonintersecting with each other and with the chosen minimum paths. Note that in situations where we only need few paths for each commodity (to have small route look-up tables), we can select the needed number of paths from the above algorithms. Similarly, for networks that require deadlock avoidance using restricted routing functions, we can use turn models to select the paths [43, 94], with only a marginal increase in the complexity of the presented algorithms.

10.3 Multipath Traffic Splitting

Once we have obtained the set of nonintersecting paths for each commodity, we need to determine the amount of flow of each commodity across the paths that minimizes congestion. Then we can assign probability values for each path of every commodity, based on the traffic flow across that path for the commodity. At run time, we can choose the path for each packet from the set of paths based on the probability values assigned to them. To achieve this traffic splitting, we use a *Linear Programming* (*LP*) based method to solve the corresponding multicommodity flow problem. The objective of the LP is to minimize the maximum traffic on each link of the NoC topology, satisfying the bandwidth constraints on the links and routing the traffic of all the commodities in the NoC. The LP is represented by the following set of equations:

$$\min t \tag{10.1}$$

$$\text{s.t.} \sum_{\forall j \in 1,\ldots,|SP_i|} f_i^j = r_i \quad \forall i \tag{10.2}$$

$$\sum_{\forall i} \sum_{\forall j, e_l \in P_i^j} f_i^j = flow_{e_l} \quad \forall e_l \tag{10.3}$$

$$flow_{e_l} \leq bandwidth_{e_l} \quad \forall e_l \tag{10.4}$$

$$flow_{e_l} \leq t \quad \forall e_l \in P_i^j, \ \forall i, j \tag{10.5}$$

$$f_i^j \geq 0 \tag{10.6}$$

In the objective function, we use the variable t to represent the maximum flow on any link in the NoC (refer to equations (10.1) and (10.5)). Equation (10.2) represents the constraint that the NoC has to satisfy the traffic flow for each commodity, with the variable f_i^j representing the traffic flow on the path P_i^j of commodity i. The flow on each link of the NoC and the bandwidth constraints are represented by equations (10.3) and (10.4). Other objectives (such as minimizing the sum of traffic flow on the links) and constraints (like latency constraints for each commodity) can also be used in the LP. As an example, the latency constraints for each commodity can be represented by the following equation:

$$\sum_{\forall j \in 1,\ldots,|SP_i|} \left(f_i^j \times l^j\right) \Big/ \sum_{\forall j \in 1,\ldots,|SP_i|} f_i^j \leq d_i \tag{10.7}$$

where d_i is the hop delay constraint for commodity i and l^j is the hop delay of path j. Once the flows on each path of a commodity are obtained, we can order or assign probability values to the paths based on the corresponding flows.

10.4 Fault-Tolerance Support with Multipath Routing

The errors that occur on the NoC links can be broadly classified into two categories: transient and permanent errors.

10.4.1 Resilience Against Transient Errors

To recover from transient errors, error detection or correction schemes can be utilized in the on-chip network [130]. Forward error correcting codes such as Hamming codes can be used to correct single-bit errors at the receiving NI. However, the area-power overhead of the encoders, decoders, and control wires for such error correcting schemes increases rapidly with the number of bit errors to be corrected. In practice, it is infeasible to apply forward error correcting codes to correct multibit errors [130]. To recover from such multibit errors, switch-to-switch (link-level), or end-to-end error detection and retransmission of data can be performed. This is applicable to normal data packets. However, control packets such as interrupts carry critical information that need to meet real-time requirements. Using retransmission, mechanisms can have significant latency penalty that would be unacceptable to meet the real-time requirements of critical packets. Error resiliency for such critical packets can be achieved by sending multiple copies of the packets across one or more paths. At the receiving switch/NI, the error detection circuitry can check the packets for errors and can accept an error free packet. When sending multiple copies of a packet, it is important to achieve the required reliability level for packet delivery with minimum data replication. We formulate the mathematical models for the reliability constraints and consider them in the LP formulation presented in previous section, as follows.

Definition 25 Let the transient *Bit-Error Rate* (*BER*) encountered in crossing a path with maximum number of hops in the NoC be β_t. Let the bit-width of the link (also equal to the flit-width) be W.

The maximum probability of a single-bit error when a flit reaches the destination is given by:

$$\text{P(Single-bit error in a flit)} = C_1^W \times \beta_t^1 \times (1-\beta_t)^{W-1} \tag{10.8}$$

That is, any one of the W bits can have an error, while the other bits should be error free.

We assume a single-bit error correcting Hamming code is used to recover from single-bit errors in the critical packets and packet duplication is used to recover from multibit errors. The probability of having two or more errors in a flit received at the receiving NI is given by

$$\text{P}(\geq 2 \text{ errors}) = \gamma_t = \sum_{k=2}^{W} C_k^W \times \beta_t^k \times (1-\beta_t)^{W-k} \tag{10.9}$$

We assume that the error rates on the multiple copies of a flit are statistically independent in nature, which is true for many transient noise sources such as soft errors (for those transient errors for which such statistical independence cannot be assumed, we can apply the method for recovering from permanent failures presented later in this section). When a flit is transmitted n_t times, the probability of having two or more errors in all the flits is given by

$$\theta_t = \gamma_t^{n_t} \tag{10.10}$$

As in earlier works [126–130], we assume that an undetected or uncorrected error causes the entire system to crash. The objective is to make sure that the packets received at the destination have a very low probability of undetected/uncorrected errors, ensuring the system operates for a pre-determined *Mean Time To Failure* (*MTFF*) of few years. The acceptable residual flit error-rate, defined as the probability of one or more errors on a flit that can be undetected by the receiver is given by the following equation:

$$Err_{\text{res}} = T_{\text{cycle}}/(\text{MTTF} \times N_c \times inj) \tag{10.11}$$

where T_{cycle} is the cycle time of the NoC, N_c is the number of cores in the system and *inj* is the average flit injection rate per core. As an example, for 500 MHz system with 12 cores, with an average injection rate of 0.1 flits/core and MTTF of 5 years, the Err_{res} value is 1.07×10^{-17}. Each critical packet should be duplicated as many times as necessary to make the θ_t value to be greater than the Err_{res} value, i.e.,

$$\theta_t = \gamma_t^{n_t} \geq Err_{\text{res}}$$
$$\text{i.e., } n_t \geq \ln(Err_{\text{res}})/\ln(\gamma_t)$$

The minimum number of times the critical packets should be replicated to satisfy the reliability constraints is given by

$$n_t = \left\lceil \ln(Err_{\text{res}})/\ln(\gamma_t) \right\rceil \tag{10.12}$$

To consider the replication mechanism in the LP, the traffic rates of the critical commodities are multiplied by n_t and equation (10.2) is modified for such commodities as follows:

$$\sum_{\forall j \in 1,\ldots,|SP_i|} f_i^j = n_t \times r_i \quad \forall i, \text{ critical} \tag{10.13}$$

10.4.2 Resilience Against Permanent Errors

To recover from permanent link failures, packets need to be sent across multiple nonintersecting paths. The nonintersecting nature of the paths makes sure that a

link failure on one path does not affect the packets that are transmitted on the other paths. As in the transient error recovery case, the critical packets can be sent across multiple paths. For noncritical packets, we assume that hardware mechanisms such as presented in [17] exist to detect and inform the sender of a permanent link failure in a path. Then the sender does not consider the faulty path for further routing and retransmits the lost flits across other nonintersecting paths. The probability of a path failure in the NoC is given by

$$\text{P(path failure)} = \gamma_p = \sum_{k=1}^{W} C_k^W \times \beta_p^k \times (1-\beta_p)^{W-k} \tag{10.14}$$

where β_p is the maximum permanent bit-error rate of any path in the NoC.

The maximum number of permanent path failures for each commodity (denoted by n_p) can be obtained similar to the derivation of n_t, and is given by

$$n_p = \left\lceil \ln(Err_{\text{res}}) / \ln(\gamma_p) \right\rceil \tag{10.15}$$

Let the total number of paths for a commodity i be denoted by $n_{\text{tot},i}$. Once the number of possible path failures is obtained, we have to model the system such that for each commodity, any set of $(n_{\text{tot},i} - n_p)$ paths should be able to support the traffic demands of the commodity. Thus, even when n_p paths fail, the set of other paths would be able to handle the traffic demands of the commodity and proper system operation would be ensured. We add a set of $n_{\text{tot},i}!/(n_p! \times (n_{\text{tot},i} - n_p)!)$ linear constraints in place of equation (10.2) for each commodity in the LP, with each constraint representing the fact that the traffic on $(n_{\text{tot},i} - n_p)$ paths can handle the traffic demands of the commodity. As an example, when $n_{\text{tot},i}$ is 3 and n_p is 1 (which means that any 1 path can fail from the set of 3 paths) for a commodity i, we need to add the following 3 constraints:

$$f_i^1 + f_i^2 \geq r_i$$
$$f_i^2 + f_i^3 \geq r_i$$
$$f_i^1 + f_i^3 \geq r_i$$

Thus, the paths of each commodity can support the failure of n_p paths for the commodity, provided more than n_p paths exist. When we introduce these additional linear constraints, the impact on the run-time of the LP is small (for the experiments, we did not observe any noticeable delay in the run-time). This is due to the fact that the number of paths available for each commodity is usually small (less than 4 or 5), and hence only a few tens of additional constraints are introduced for each commodity. Note that we can modify the mapping procedures to ensure that each commodity has more than n_p paths available for data transfer. In cases where the mapping procedure cannot produce more than n_p paths for some commodities, we can introduce additional links between switches to get such additional paths for the commodity. Modifying the NoC mapping and topology design processes to achieve these effects is beyond the scope of this chapter.

10.5 Simulation Results

10.5.1 Area, Power and Timing Overhead

The estimated power overhead (based on gate count and synthesis results for switches/NIs) at the switches/NIs to support the multipath routing scheme for the 4×3 mesh network considered earlier (in Table 10.1) is found to be 18.09 mW, which is around 5% of the base NoC power consumption. For the power estimation, without loss of generality, we assume that 8 bits are used for representing the source and destination addresses and 8-bit packet identifiers are utilized. The power overhead accounts for the look-up tables and the combinational logic associated with multipath routing scheme. The numbers assume a 500 MHz operating frequency for the network. The estimated area overhead (from gate and memory cell count) for the multipath routing scheme is low (less than 5% of the NoC component area). The maximum possible frequency estimate of the switch design with support for the multipath routing tables is above 500 MHz.

10.5.2 Case Study: MPEG Decoder

We assume that the tasks of the MPEG application are assigned to processor/memory cores and the best mapping (minimizing the average communication hop delay) onto a mesh network is obtained using the SUNMAP tool (presented in Chapter 2). The communication between the various cores and the resulting mapped NoC are presented in Figures 10.3 and 10.4. The various paths obtained using the algorithms presented earlier, for some of the commodities, are presented in Table 10.2. Applying the LP procedure to split the traffic across the obtained paths results in 35% reduction in the bandwidth requirements for the application when compared to single-path routing (both dimension-ordered and minimum-path routing). The bandwidth savings translates to 35% reduction in the required NoC operating frequency. For 16-bit link data width, the multipath routing requires 300 MHz frequency for the NoC to support the application traffic, while the single-path routing schemes require a NoC frequency of 405 MHz. The NoC frequency reduction leads to 22.22% reduction in power consumption of the NoC for the multi-path scheme compared to single-path schemes, after accounting for the power overhead of the multipath scheme. The average packet latencies incurred for the MPEG NoC for *dimension ordered* (*Dim*), *minimum path* (*Min*) and the proposed *multipath* (*Multi*) strategy for the MPEG NoC is presented in Figure 10.5(a). The simulations are performed on a flit-accurate NoC simulator designed in C++. The multi-path routing strategy results in reduced frequency requirements to achieve the same latency as the single-path schemes for a large part of the design space.

When compared to the multipath routing scheme with re-order buffers (10 packet buffers/receiver), the current scheme results in 28.25% reduction in network power consumption.

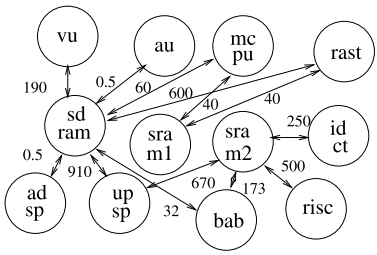


Fig. 10.3 A MPEG decoder application

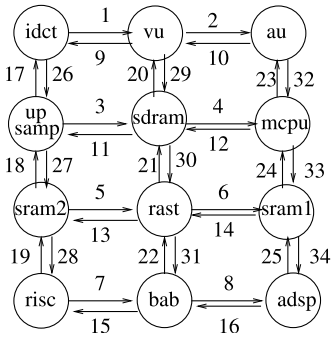


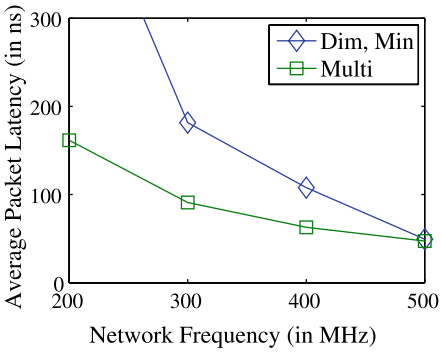
Fig. 10.4 Mapped onto a mesh NoC. The edges of the mesh are numbered

Table 10.2 Sample paths

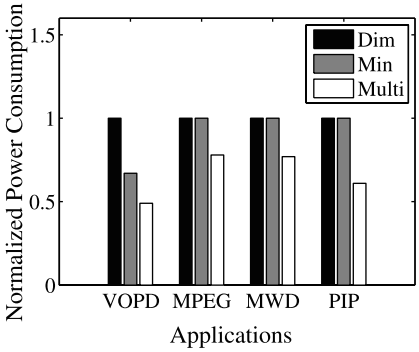
Comm.	Source	Dest.	Paths (edges traversed)
1	au	sdram	32–12, 10–29
2	mcpu	sdram	12, 23–10–29, 33–14–21
3	upsamp	sram2	27, 3–30–13
4	risc	sram2	19, 7–22–13

10.5.3 Comparisons with Single-Path Routing

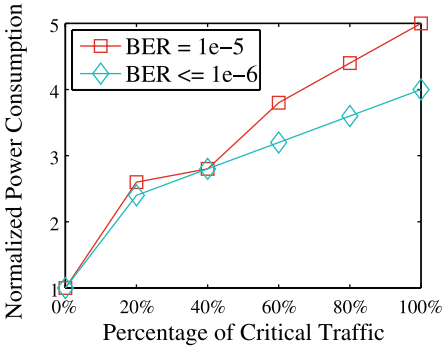
The network power consumption for the various routing schemes for different applications is presented in Figure 10.5(b). The numbers are normalized with respect to the power consumption of dimension-ordered routing. We use several benchmark applications for comparison: *Video Object Plane Decoder* (*VOPD-mapped onto 12 cores*), *MPEG decoder* (*MPEG-12 cores*), *Multi-Window Display application* (*MWD-12 cores*) and *Picture-in-Picture* (*PIP-8 cores*) application. Without loss of generality, we assume that the applications are mapped onto mesh topologies, although the multipath routing strategy can be used for any topology. By using the proposed routing scheme, on average we obtain 33.5% and 27.52% power savings compared to the *dimension ordered* and *minimum path routing*, respectively. The total run time for applying the proposed methodology (includes the run time for path selection algorithms for all commodities and for solving the resulting LP) is less than a few minutes for all the benchmarks, when run on a 1 GHz Sun workstation.



(a) Average latency for MPEG NoC



(b) Routing effects on applications



(c) Effect of error resiliency

Fig. 10.5 (**a**) Performance of routing schemes for MPEG NoC. (**b**), (**c**) Effect of routing and fault-tolerance on NoC power consumption

10.5.4 Effect of Fault-Tolerance Support

The amount of power overhead incurred in achieving fault-tolerance against temporary errors depends on the transient bit-error rate (β_t) of each link and the amount of data that is critical and needs replication. The effect of both factors on power consumption for the MPEG decoder NoC is presented in Figure 10.5(c). The power consumption numbers are normalized with respect to the base NoC power consumption (when no fault-tolerance support is provided). As the amount of critical traffic increases, the power overhead of packet replication is significant. Also, as the bit-error rate of the NoC increases (higher BER value in the figure, which imply a higher probability of bit-errors happening in the NoC), the amount of power overhead increases. We found that for all BER values lower than or equal to 1e−6, having a single duplicate for each packet was sufficient to provide the required MTTF of 5 years. Adding support for resiliency against a single-path permanent failure for

each commodity of the MPEG NoC resulted in a 2.33× increase in power consumption of the base NoC.

10.6 Summary

In this chapter, we have presented a multipath routing strategy that guarantees in-order packet delivery at the receiver. We introduced a methodology to find paths for the routing strategy and to split the application traffic across the paths to obtain a network operation with minimum power consumption. With technology scaling, reliable operation of on-chip wires is also rapidly deteriorating and various transient and permanent errors can affect them. With the proposed multipath routing strategy, we explored the use of spatial and temporal redundancy to tolerate transient as well as permanent errors occurring on the NoC links. The proposed method results in large NoC power savings for several SoC designs when compared to traditional single-path systems.

Chapter 11
NoC Support for Reliable On-Chip Memories

One of the key elements in MPSoCs that are affected by variability in sub-micron technologies are on-chip memories [142]. The on-chip memories are especially susceptible to Single Event Upsets (SEUs) such as soft errors, as the transient noise sources can flip the bits in the memory cells. Since the memories store the instructions and data that are used by the processors, having permanent or temporary failures in memories can result in complete failure of the system. Current memories already include extensive mechanisms to correct transient single-bit errors, e.g., by using error-correcting codes such as Hamming code [143] in the memory arrays. However, these mechanisms are expensive and the overhead in area, power, and delay to be implemented massively inside memories to automatically recover from multi-bit errors would be very high [142]. Hence, suitable system-level support to provide efficient fault-tolerant mechanisms for memories will be mandatory to ensure proper operation of future MPSoC designs.

From the hardware point of view, the use of NoCs helps the designer to overcome the reliability issues of future technologies. The high flexibility of NoCs allows the designer to add redundant cores in the same chip (e.g., processing elements, backup memories) without largely increasing the design complexity. From the software point of view, the type of applications that will be present in future MPSoC designs are various multimedia services, such as scalable video rendering, video-games, etc. In order to provide the required reliability level for the system, the characteristics of these applications need to be studied in detail.

As a major contribution of this chapter,[1] we address the design of a reliable on-chip memory subsystem. The key idea of the proposed approach is to automatically keep backup copies of critical data on a reliable memory; upon a fault event, data is transparently fetched from the backup copy in hardware, without any software intervention. To achieve this, we present a novel hardware solution that utilizes NoCs to provide a scalable, efficient and transparent mechanism for fault tolerance. At the software level, we characterize the application data into two different types: *critical* and *non-critical*. We show that for many multimedia applications, for proper system operation, we only need to back up the critical data (and the instructions), which is a small fraction of the total. We validate this by presenting case studies on two real-life multimedia applications. We handle two kinds of faults in the memories: intermittent and permanent. When a permanent or intermittent fault occurs on the main memory, the NoC is dynamically reconfigured to switch all critical transactions to the backup memory. For an intermittent failure, when the main memory recovers from the failure, the NoC switches back all transactions to the main memory.

[1] This chapter is primarily based on the work by Dr. Federico Angiolini.

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

The use of NoCs to provide fault tolerance has several implications. First, we achieve a modular and scalable design. Thus, backup devices can be added to existing designs without increasing the design complexity. It also allows us to add communication bandwidth where needed in the chip, so that performance bottlenecks due to replicated traffic is minimized. Second, we are able to provide dynamic fault tolerance support that is decoupled from the software. This implies that the processors are unaware of the memory failures. Thus, for reliability support, only a limited effort by the application designer is required. Third, the NoC paradigm makes it very easy to place the main and backup memories far away in the chip floorplan; this is a key point to counter failures due to phenomena such as thermal hot-spots. Finally, the NoC architecture enables the decoupling of the frequency of the interconnect from those of the attached cores, allowing for clocking backup memories at a lower frequency. Thus, we can use backup memories that are slower and hence more reliable than the main memories, without additional components for clock conversion.

We implement the proposed fault tolerance mechanism on top of the existing NoC platform and integrate it within the Netchip tool flow (presented in Chapter 2). We perform experiments on realistic multimedia applications, which show that the performance penalty in adding the fault tolerance support is negligible. We present several experiments to explore various parameters that impact the performance and area overhead of the fault tolerance mechanism. We synthesize the additional hardware components that are added in the NoC to provide the fault tolerance support. The silicon overhead is less than 10% the area of an extra backup memory bank itself (assuming a 32 kB size), which represents the baseline requirement for any replication-based fault tolerance strategy.

11.1 Analysis of Multimedia Software

New multimedia applications cover a wide range of functionality (video processing, video conferencing, games, etc.); one of their main common features is that they process large amounts of incoming data in a streaming-based way (e.g., a continuous flow of frames). We can observe that certain parts of these streams are essential to produce a correct output, while others are not so critical and only partially affect the deployed quality to the user. In many multimedia applications, it is possible to distinguish critical from non-critical data because each type is stored in different classes and variables within the applications. Let us briefly illustrate these characteristics in the implementation of a real-life multimedia application that is used as one of the case studies in Section 11.4, i.e., an MPEG-4 Video Texture Coder (VTC). VTC is the part of the MPEG4 standard that deals with still texture object decoding. It is a wavelet transform coder, which can be seen as a set of filter-banks [144] sent in a stream of packets. Each packet represents a portion of an image in different subbands, i.e., at different resolutions.

Fig. 11.1 Complete 2D wavelet decomposition in VTC for one image encoded with DC and 3 AC levels

As an example, a portion of an input image with 3 levels of resolution is shown in Figure 11.1. As this figure depicts, the first packet of the stream includes the basic elements of the image, but at low resolution. This part is called the DC SUBBAND of the wavelet (top-left corner of Figure 11.1). If the data that represents the DC subband is lost, the image cannot be reconstructed. As typical of critical data in streaming applications, it is very small in size (few kBytes for 800×640 images) and is stored in a dedicated variable and class within the VTC code.

The following packets of the stream are called AC OR SPATIAL LEVELS and contain additional details about the image. They have a much larger size than the DC subband, but they only refine the image represented by the DC subband. If data representing these levels is lost, the user still sees an image, just at lower resolution. Moreover, whenever a new frame arrives, the previous (faulty) picture is to be updated with the newly received information. Hence, any low resolution output only lasts a very limited amount of time.

From this example, we can derive fault tolerance requirements for the data set of typical multimedia applications. Only a small subset of the data structures is critical to the quality of output as perceived by the user, while most of the data set to be processed is actually of little importance in this respect. Therefore, it is essential to preserve correct copies only of the former data set, while faults in the latter may be safely accepted.

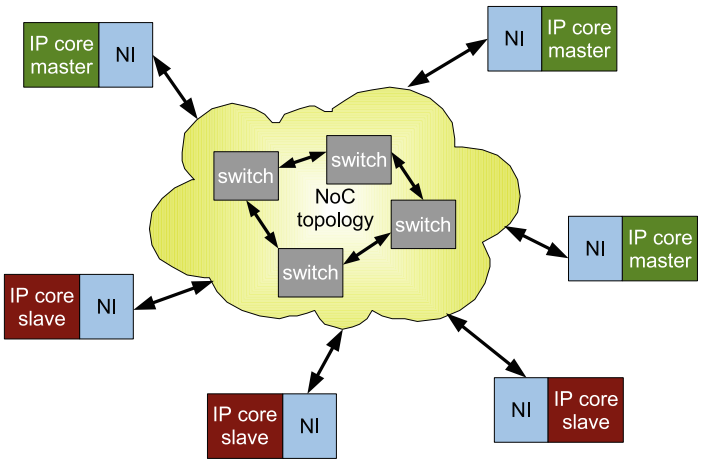


Fig. 11.2 General view of a NoC

11.2 Baseline SoC Architecture and Extensions

11.2.1 SoC Template Architecture

The reference SoC that we consider is composed of computation cores, a communication backbone implemented by means of the ×pipes NoC, and a set of system memories.

A typical NoC is built around three main conceptual blocks: NETWORK INTERFACE (NI), SWITCH (also called router) and LINK (Figure 11.2). Network interfaces perform protocol conversion from the native pin-out of IP cores to NoC packets; routers deliver packets to their recipients; and finally, links connect the previous blocks to each other, handling propagation delay issues. We modify the NoC architecture by extending its building blocks to support reliability-aware features.

For the reference system, we assume the availability of at least two specific classes of memories: "error-detecting" and "reliable." While not specifically designed to prevent data corruption, error-detecting memories, which can be commonly found today, are at least capable of detecting such occurrences, for example, by CRC codes, and notifying them to their controller. We also postulate the availability of memories with much higher reliability for backing up critical data. This assumption is motivated by ad-hoc circuit level solutions and strengthened by three design choices we enable for these memories: (i) they have a small capacity, (ii) they are run at a lower-than-usual clock frequency (in this chapter, we assume one-half that of regular memories), (iii) during typical system operation, they face a smaller workload than regular memories.

We assume the existence of *main* memories having error detection capability; normal SoC operation leverages upon them, including storage of critical and non-critical data. We add smaller spare *backup* memories, featuring higher reliability, to

hold shadow copies of critical data only. Each main memory requires the existence of one such backup, although a single storage device can hold backups for multiple main memories.

To identify the critical data set, we assume that the programmer defines the set of variables to be backed up, and maps them to a specific memory address range. This address range is then used to configure the NoC, either at design time or at runtime during the boot of the system. The accesses to this particular memory region are thereafter handled with the proposed schemes, improving the fault tolerance of the MPSoC design. Application code is assumed to be a vital resource, too. Therefore, instructions are always treated in the same way as the critical data; in the remainder of the chapter, we will not mention this distinction for the sake of simplicity. Note that the classification of data into critical and noncritical can also be done using efficient compiler support. In this case, the user can define the critical data using special macros and the compiler can map the data to a specific address range.

The size of the critical data set will depend on the application at hand, and is impossible to predict in general. We aim this work at streaming applications, mostly in the multimedia field, for which the amount of critical information can be safely assumed to be small in percentage. These applications do however represent a significant slice of the embedded device market.

11.2.2 Proposed Hardware Extensions

To implement the approach, we perform changes to the NoC building blocks. The flexible packet-switching design of NoCs ensures that these changes are transparent to the transport layer (switches and links), but NIs need to be made aware of fault events. Two NIs exist natively: *initiator NI* (attached to a system master, such as a processor) and *target NI* (attached to a system slave, such as a memory). Both follow some connection protocol specification at the IP core side, such as OCP 2.0 [91], and perform source routing by checking the target of the transaction against a routing lookup table.

The target NI is devoted most of the attention, as can be seen by comparing Figure 11.3(a) (native) and Figure 11.3(b) (extended for reliability purposes). The original target NI is still plugged to backup memories, while the extended version is used for main memories. A plain target NI features an input request channel, where request transactions from system masters are conveyed, and an output response channel, where memory responses are packeted and pushed toward the NoC. A third channel (redundancy channel) is now added in the extended target NI; this channel is an output, and reinjects some of the request packets back again into the NoC. By this arrangement, critical-data accesses to the memory (i.e., within a predefined address range) can be forwarded to the backup storage element. Not all packets are forwarded; during normal operation, that is until a fault is detected, only writes to critical address regions follow this path. This ensures that the backup memory is kept up to date with changes in critical data, but minimizes network traffic overhead

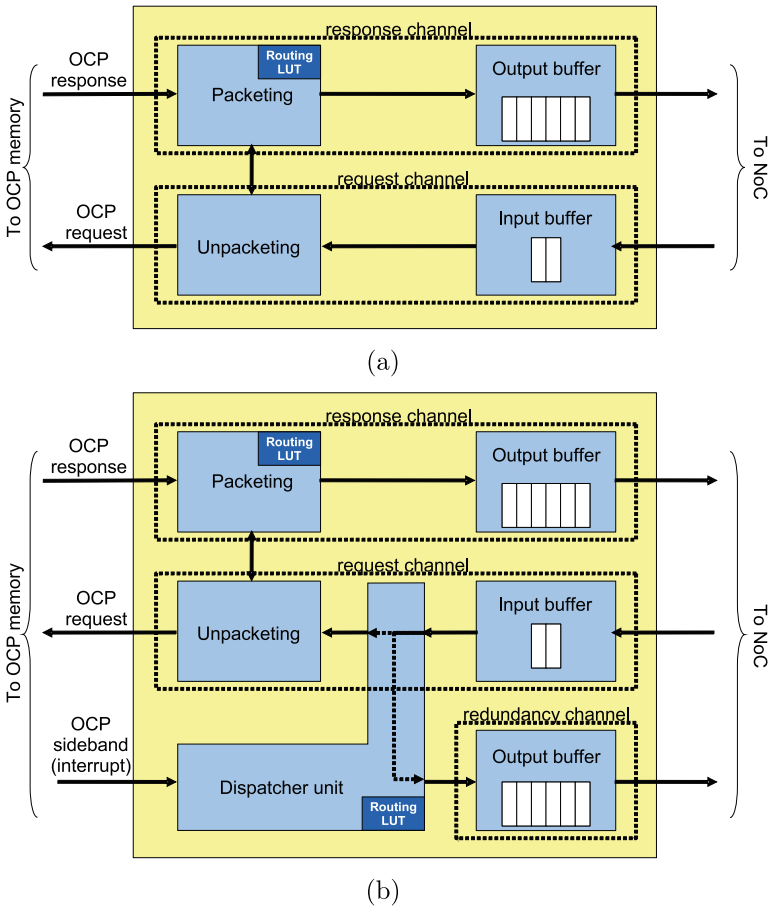


Fig. 11.3 (**a**) Plain target NI architecture, (**b**) Extended target NI architecture

and increases reliability of the backup memory, which faces less workload than the main device. Since the backup memory only receives write commands, it remains silent, i.e., it does not send unneeded messages onto the NoC. This prevents conflicts such as two memories responding to the same processor request. As a result, the flow of packets with the forwarding mechanism being active during normal operation is depicted in Figure 11.4(a). The forwarding behavior is controlled by a DISPATCHER block that sits in the middle of the NI and supervises input and output packet flows. This block also takes care of flow control issues. For example, when a packet has to go toward both the memory and the redundancy channel, and one of them is busy, the dispatcher issues the proper stalls. For the purposes of packet forwarding, we add an extra routing lookup table, which however consists of a single entry since there is only one backup memory per each main memory.

The extended target NI also features an extra interrupt interface by the memory side. Whenever a fault is detected, the memory can raise its interrupt wire. This

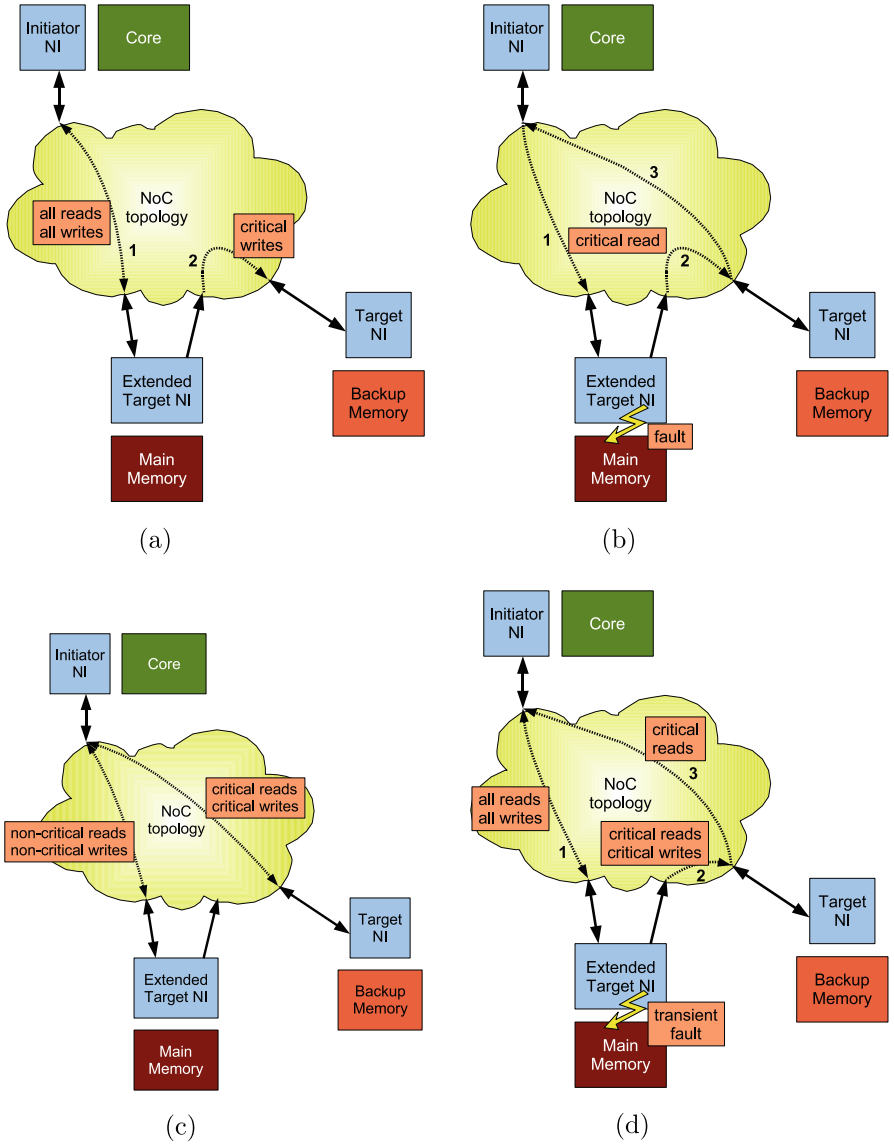


Fig. 11.4 Handling of packet flow in the system. (**a**) Normal operation with backup, (**b**) First phase of recovery for permanent and transient failures: read transaction handling upon fault occurrence, (**c**) Final operation mode after recovery from permanent failure, and (**d**) Operation mode while a transient failure is pending

triggers a change in the activity of the dispatcher, which responds by beginning to forward critical read packets to the backup memory according to the extra routing table entry. In this way, reads that would fail due to data corruption are instead

transparently forwarded to the backup memory and safely handled (see Section 11.3 for more details). Critical writes continue to be forwarded, as they already were before the fault occurrence.

The initiator NI is also extended in two ways. First, it checks all outgoing requests for their target address. If the address falls in the specific range provided by the application designer as storage of critical data, then a flag bit is set in the packet header. This allows the dispatcher in the extended target NI to very easily decide whether to forward packets or not. A second change in this NI involves an extra entry in its routing lookup table, and a very small amount of extra logic that checks the header field of response packets. Like request packets, response packets contain a `SourceID` field. This means that the initiator NI can detect whether a read request it sent got a response from the intended slave or from a different one. As we will show in this approach, upon a fault, critical reads receive responses from the backup memory instead of the main one. Therefore, noticing a mismatch is an indirect indicator of whether there was a fault in the main memory. This can trigger different actions depending on the type of error that needs to be handled, as described in Section 11.3.

11.3 Run-Time Fault Tolerant Schemes

Two types of errors can occur in on-chip memories of MPSoC designs, namely, intermittent or permanent. We assume that the system is able to recognize transient errors by detecting some known combination of parameters, either upon the error event itself or even before any error appears. For example, a thermal sensor detecting that a threshold overheating temperature has been surpassed may signal a "transient error" condition before any real fault is observed. The "transient error" condition would be deasserted once the temperature returns to acceptable levels. The same prevention or detection principle could be applied to other electrical or functional parameters that may indicate that a critical point of operation is being approached, such as an increased delay in the toggling of some wires, or possibly the insertion of more wait states by the memory before responding. In the case of highly fault tolerant systems where the main memory is itself equipped with error correction (not only detection) logic, any internal correction event could be pessimistically assumed as a hint of a possible imminent failure; this hypothesis could be reversed after a configurable period of time, once the isolated correction event can be safely assumed to be an occasional glitch, or when the functionality of the main memory can be somehow again assumed as reliable, e.g., thanks to some (self-)testing routine. Any known-critical or unexpected events should however be treated by the system as permanent faults, and accordingly handled.

In the following subsections, we describe how the proposed extensions can be used to design schemes capable of handling both transient and permanent failures in a way that is transparent to the software designer. As a common feature in both cases, the backup memories do not contain any data at the beginning of the execution and are filled at run-time by copying data from the coupled main memory. The recovery process is carried on in two phases.

11.3.1 Permanent Error Recovery Support

In the case of permanent errors, as soon as the error is identified, the recovery begins. During the first phase, critical write operations continue to be forwarded to the main memory as in normal operation (see Section 11.2.2), but the extended target NI now also starts diverting the read requests to the associated backup memory. From this moment on, the backup memory, which had been silent, begins to generate responses as a reaction to the master reads. At the same time, requests being diverted, the main memory stops replying to the initiator for accesses into the critical address range. Since NoC responses are sent to a device or another depending on the `SourceID` field of request packets, and the diverted packets keep this field unchanged, the backup memory automatically sends its reply to the system master that had originally asked for it. This does not require any lookup conversion in the NIs of the backup memories. Figure 11.4(b) shows the handling mechanism of a read transaction upon a fault.

Since going through the main memory and then the backup memory to fetch data is time consuming, the second phase of the recovery process for permanent faults tries to minimize the performance impact of this three-way handling of critical reads. To this end, the extended initiator NI (Section 11.2.2) is able to identify whether the source of read responses is the main or the backup memory. The first critical read after the fault occurrence triggers a mismatch detection, which in turn forces the initiator NI to access a different entry within its routing lookup table. Hence, all following memory reads within the critical address range are directly sent to the backup memory after the fault. This clearly improves latencies during the remainder of operations. The resulting flow of packets is shown in Figure 11.4(c).

It is worth to stress that the approach does not introduce any data coherency issue. During normal operation, the forwarding of write transactions guarantees that critical data is always consistent among the main and backup memories. Writes are forwarded just before hitting the main memory bank, not after having been performed; in this way, a faulty main memory has no chance of polluting the backup copy of the data. The contents of the backup memory are updated after a slight delay, but this causes no issue as the sequence of packets is strictly maintained. Upon a fault occurrence, transactions are initially directed to the main memory, and only afterward, when needed, are routed to the backup device; this arrangement avoids skipping transactions and guarantees that all pending transactions (reads and/or writes) are completed on the correct copy of the data. Therefore, proper functionality is strictly maintained when introducing the extra storage bank.

Similarly, when adding the backup memory to the NoC, deadlock issues do not arise given a proper design of the NoC routing scheme. In this respect, the NoC designer must accommodate for one extra IP core and some extra routing paths during the deadlock-free NoC mapping stage. Remarkably, under certain common circumstances (such as $X-Y$ routing on regular mesh topologies), no extra effort is required to the designer at all. No specific traffic priority mechanisms are required in the NoC, even though the designer may choose to prioritize some traffic flows according to specific needs, as in any interconnect fabric.

11.3.2 Intermittent Error Recovery Support

In the case of transient errors (e.g., due to overheating detection), the first phase of the recovery process is the same as in the case of the permanent errors, namely, the read transactions are automatically forwarded to the backup memory, which automatically responds to the initiator. However, the second phase differs due to the nature of transient failures, where the main memory is supposed to recover complete functionality at a certain moment in time. All traffic, including the critical one, continues to be sent from the processor to the main memory. The extended target NI, being aware that a fault condition is pending, diverts all critical reads toward the backup memory, but let's critical writes be performed toward both the main and backup locations. When the main memory detects that it is able to return to normal operation (e.g., after the temperature has returned to normal levels), it is allowed to issue a different interrupt to indicate the new condition. The extended target NI at this point resumes normal operation.

The main assumption in this approach is that updates to the critical data set in main memory can be successfully performed even during the "transient fault" state. This might be allowed, e.g., by choosing conservative temperature thresholds to assert the fault warning. Otherwise, if this solution is not acceptable and the designer does not want to consider the fault permanent, we assume that a higher-level protocol (e.g., MAC or Network layer) will transfer the safe backup copies of critical data back to the main memory after its return to full functionality.

11.4 Experimental Results

To assess the validity of the approach, we employ two different benchmarks from the multimedia domain. The first one is the MPEG-4 VTC application already described in Section 11.1. As a second test, we use one of the subalgorithms of a 3D Image Reconstruction algorithm [145], 3DR for short, where the relative displacement between every two frames is used to reconstruct the third dimension. Similarly to the VTC benchmark, the amount of critical data that stores control information about the matching process (e.g., 160 kB for images of 640×480 pixels) is much smaller than the overall input data per each 2-frame matching process (2 MB of data at the same resolution), and it is stored in two data structures which are easily identifiable by the application designer.

In the experiments, we run the 3DR and the VTC benchmarks on top of three reliability-enhanced topologies, as shown in Figure 11.5. Both benchmarks are implemented using 10 processing cores and a single main memory. The first topology is a NoC crossbar, the second is a star, and the third is a mesh. The topologies and benchmarks are chosen to illustrate different situations of performance penalty for adding reliability support, since the applications demand different features. In fact, 3DR tends to saturate the main memory bandwidth, while VTC is less demanding. The NoC is simulated within a cycle-true simulation environment. We clock the NoCs at 900 MHz, twice the frequency of the cores and memories.

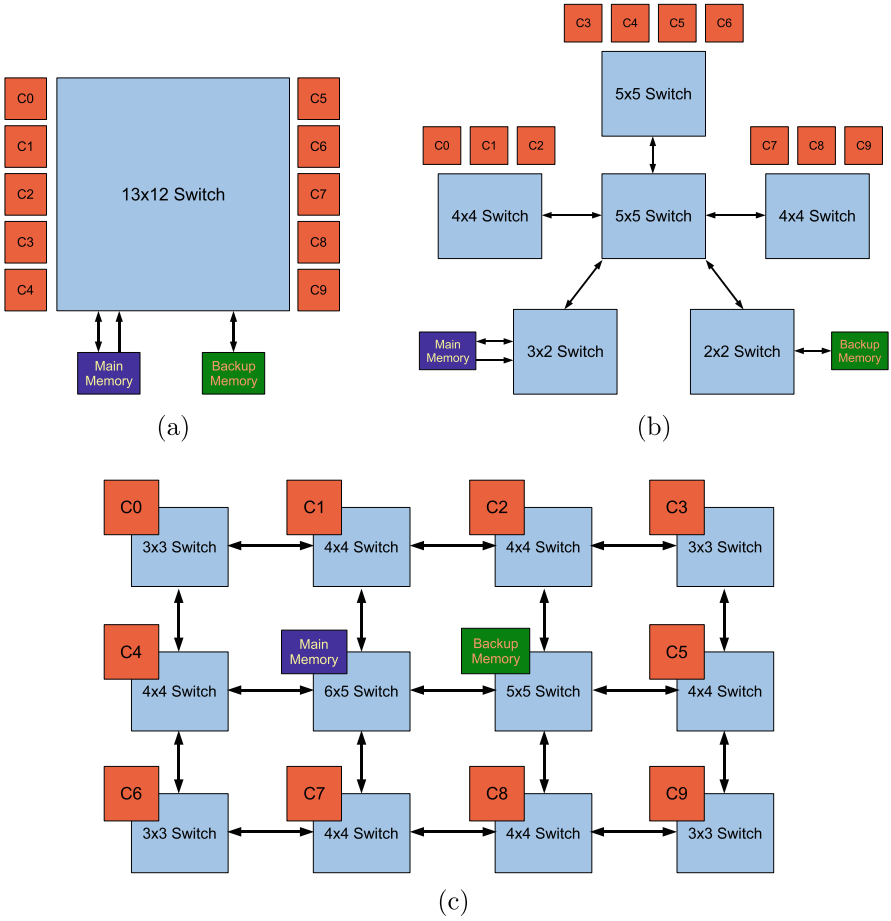


Fig. 11.5 The three topologies under test: (**a**) crossbar, (**b**) star, and (**c**) mesh

11.4.1 Performance Studies

We run the benchmarks in five different setups. The first two are reference baselines, the remaining ones represent the proposed scheme.

- *Reference-Unreliable*: The reference run is a system without reliability support at all, where accesses are to a fast (450 MHz) main memory. No faults are supposed to happen.
- *Reference-Robust*: we model the same system with a reliable main memory running at a lower frequency, therefore, minimizing error occurrences [141] and accounting for the overhead of extra circuitry. System performance is obviously impacted, but robust operation can be assumed.
- *Proposed-Replication*: We create a system with a fast main memory and deploy a slow backup memory, but we do not yet inject any fault in the system. As a

result, the overhead for the backup of critical data can be observed. We assume the backup memory to be clocked at half the clock speed of regular memories, for the same reasons outlined in the previous setup.

- *Proposed-Permanent*: We create a system with a fast main memory and deploy a slow backup memory, then inject a permanent fault right at the beginning of the simulation. This enables the evaluation of the impact of accessing the backup copy of critical data.
- *Proposed-Transient*: We create a system with a fast main memory and deploy a slow backup memory, then inject a transient fault right at the beginning of the simulation, and never recover from it. This analysis helps to understand what happens to system performance during the period where the main memory is accessed first, but critical traffic needs to be rerouted to the backup memory.

Figure 11.6 reports performance, measured in completed transactions per second, for all three topologies, both benchmarks and all five fault scenarios.

The system throughput of most of the scenarios is close, with *Reference-Robust* being much worse than average and *Proposed-Permanent* performing much better, at least in the 3DR case, than even the *Reference-Unreliable* scenario. We explain these major effects by observing that both benchmarks, like most multimedia applications, place heavy demands in terms of memory bandwidth; this is a very logical consequence of parallel computing on a 10-core system. In *Reference-Robust* the available memory bandwidth is decreased to provide more reliability, which causes performance to worsen dramatically. In the tests, VTC throughput drops by about 24% and 3DR by as much as 43%, since 3DR is even more demanding. For the same reasons, the *Proposed-Permanent* scenario, where critical data is stored in a separate device, actually guarantees a performance boost related to load balancing among the two memories; the boost is barely 1% for VTC, but around 40% for 3DR, which is more bandwidth-limited and more heavily accesses the memory regions tagged as critical. Under less demanding applications, we expect *Reference-Robust* to get closer to the reference system and *Proposed-Permanent* to perform more or less on par with it.

The *Proposed-Replication* scenario exhibits a minimal penalty compared to the unreliable case, since the traffic associated to shadowing of the critical writes is well handled by the NoC. VTC rarely accesses critical addresses, so no overhead is noticeable, while in 3DR the throughput decrease is of 1% to 9%, with the star topology experiencing the worst congestion.

The *Proposed-Transient* case exhibits a performance level close to *Reference-Unreliable*, because noncritical traffic behaves exactly as in the base scenario, but several effects related to critical traffic have to be accounted for. On the one side, critical traffic creates NoC congestion and incurs a latency overhead. On the other hand, the main memory does not have to process critical reads, therefore the noncritical transactions can be executed with less delay. In VTC, the overall balance is roughly even. In 3DR, where a larger amount of critical reads (e.g., instruction cache refills) takes place, the main memory benefits from large latency gains, boosting performance.

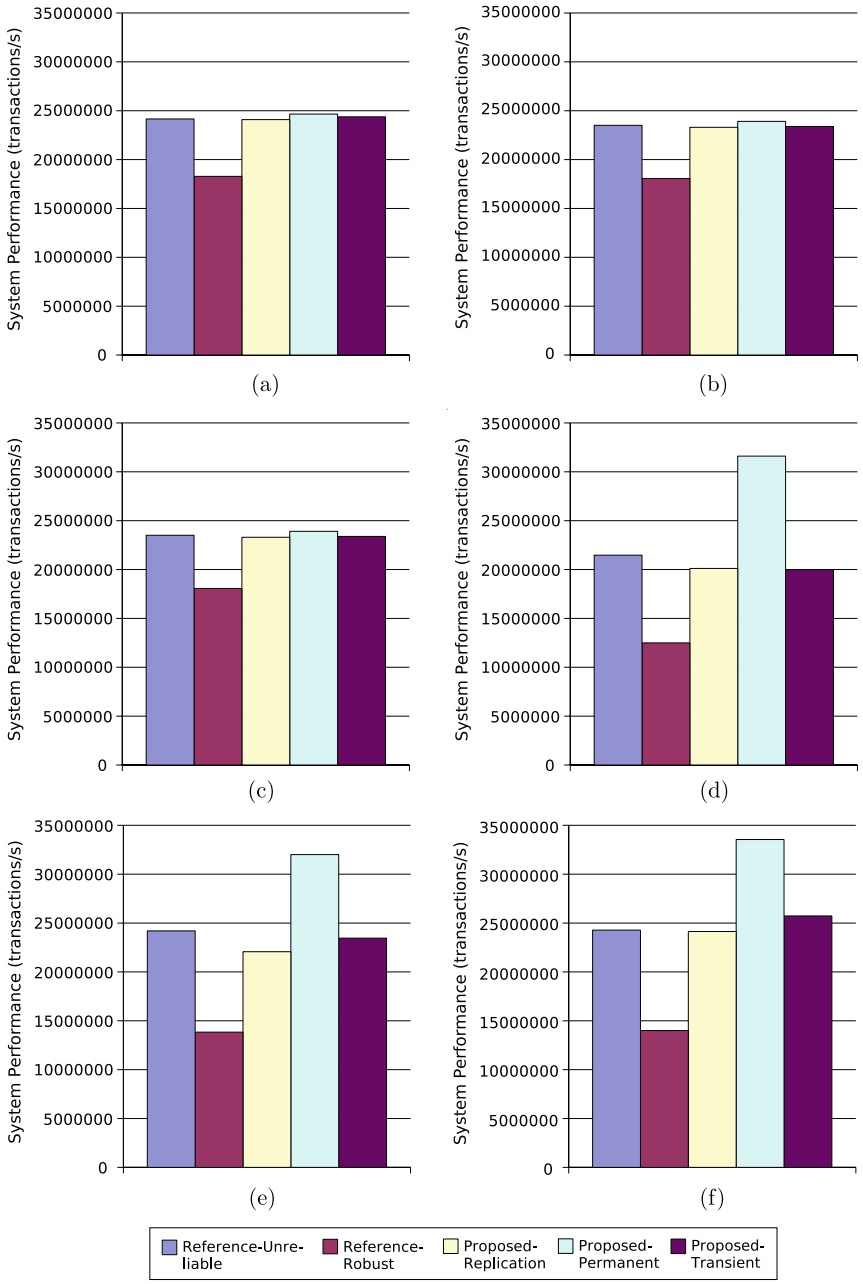


Fig. 11.6 Comparative performance of adding reliability support for (**a**, **b**, **c**) the VTC benchmark on crossbar, star, and mesh topologies, respectively, (**d**, **e**, **f**) the 3DR benchmark on crossbar, star, and mesh topologies, respectively

Experimental results show that in order to improve system reliability, deploying a single highly fault tolerant main memory (*Reference-Robust*) may not be a wise choice in terms of performance within complex multimedia systems. In the proposed architecture, the main memory is left running at a high frequency, and a slower secondary memory bank is added. This choice incurs minor throughput overheads both during normal operation and after fault occurrences. These results justify the feasibility of deploying the architecture even in throughput-constrained environments.

The gains we outline for the *Proposed-Permanent* scenario suggest that always mapping critical information to a separate reliable memory, without intermemory transactions, may be a simpler yet efficient, due to load balancing, approach. However, such a choice does not improve reliability as much as the backup mechanism, due to two main factors. First, having two copies of critical data is certainly more reliable than having a single one. Second, using the main memory as the default resource permits a lower workload for the backup memory during normal operation (only write transactions need to be processed), which further increases its reliability. Since the focus of this work is high fault tolerance, we feel that a redundant data mapping is justified, and our aim is simply to verify that performance is not seriously impacted as a result. Performance optimizations through reduction of local congestion can, in any case, be achieved by the system designer by tuning the memory hierarchy, which includes deploying multiple storage elements; these steps can be taken in combination with the proposed approach, and are out of the scope of this work.

11.4.2 Architectural Exploration of NoC Features

We extend the analysis to different NoC-based hardware architectures using the same NoC backbone. We vary some parameters of the baseline topologies. First, we modify the star topology of Figure 11.5(b) by attaching the backup memory beyond a further dedicated switch. The total distance from the central hub is therefore of two hops instead of one. In this way, we model backup memories further apart from main memories in the chip floorplan, which improves tolerance in case of overheating. Performance is unchanged under the *Reference* scenarios, where the backup memory is never accessed. In *Proposed* scenarios, where the backup storage is in fact accessed, throughput worsens by less than 0.3%. This is because the latency to go through an extra hop in the NoC is very small, provided there is limited congestion as in this star topology. If the number of hops needed to reach the backup memory is large enough to potentially affect latency, or if such hops suffer from heavy congestion, the topology designer may want to add dedicated NoC links.

To test the dependency of performance on the buffer depth of the redundancy channel, we try a sweep by setting this parameter within the extended target NI from 3 to 6 stages. The results indicate that, both in VTC and 3DR, deep FIFOs only improve system performance by less than 2%, which indicates that large buffering is not mandatory in the extended target NI. Hence, area can be saved.

To validate the effectiveness of the routing shortcut that is enabled in the initiator NI after permanent faults, we measure the latency of two different transactions on the star topology: (1) a critical read going from the core to a faulty main memory, bouncing toward the backup memory, and from there to the processor again and (2) a read directly toward the backup memory after the processor has updated its internal lookup tables. The minimum latency is cut from 78 to 68 (−13%) clock cycles, and the average one goes down from 103 to 95 (−8%). Although this metric is topology-dependent, it shows the advantage of updating the routing decisions of the initiator upon the occurrence of permanent faults.

11.4.3 Effects of Varying Percentages of Critical Data

An important topic is the exploration of different reliability/performance trade-offs according to the amount of variables that are considered critical: the more data needs to be backed up, the larger the safe backup memories need to be. Since backup memories are supposed to be reliable also thanks to being smaller, slower, and relatively little accessed, the effect of having large backups upon reliability is unclear. To shed some light onto the performance side of the issue, we analyze the behavior under different rates of possible critical vs. noncritical data in Figure 11.7. The star topology is taken as an example. In the plots, the *Reference-Unreliable* bar can be assumed to represent an ideal case where no data is critical. For the *Proposed* cases, we configure two different memory spaces to be protected against faults: the actual critical set of the benchmark (the same of the studies in Figure 11.6, labeled "critical set"), and as an extreme bound, the whole address space ("all set").

The first interesting remark is that the *Proposed-Replication* performance, i.e., the system throughput before any fault occurrence, but in presence of the backup overhead, is only moderately impacted by the size of the critical data set. In VTC, *Proposed-Replication* performance is always close to the baseline case; in 3DR, which is more bandwidth-limited, even backing up the whole address space incurs a penalty of just 18%.

As expected, in case of a fault occurrence, the size of the protected memory space is a key performance parameter. While choosing a small critical set allows for very good throughput, extending the fault tolerance to the whole main memory content incurs a dramatic penalty. This is however to be fully expected. In the *Proposed-Permanent* case, all traffic is redirected to the backup memory, which is running at a lower frequency: Therefore, throughput becomes identical to the *Reference-Robust* baseline. The *Proposed-Transient* scenario is even slightly worse, since the same traffic has to go through an extra hop first.

This bracket of results frames the applicability of this approach. If the critical set of the application can be kept small, throughput penalties are minimal and advantages are clear. Otherwise, performance degrades up to a worst case equivalent to a system with a single reliable memory.

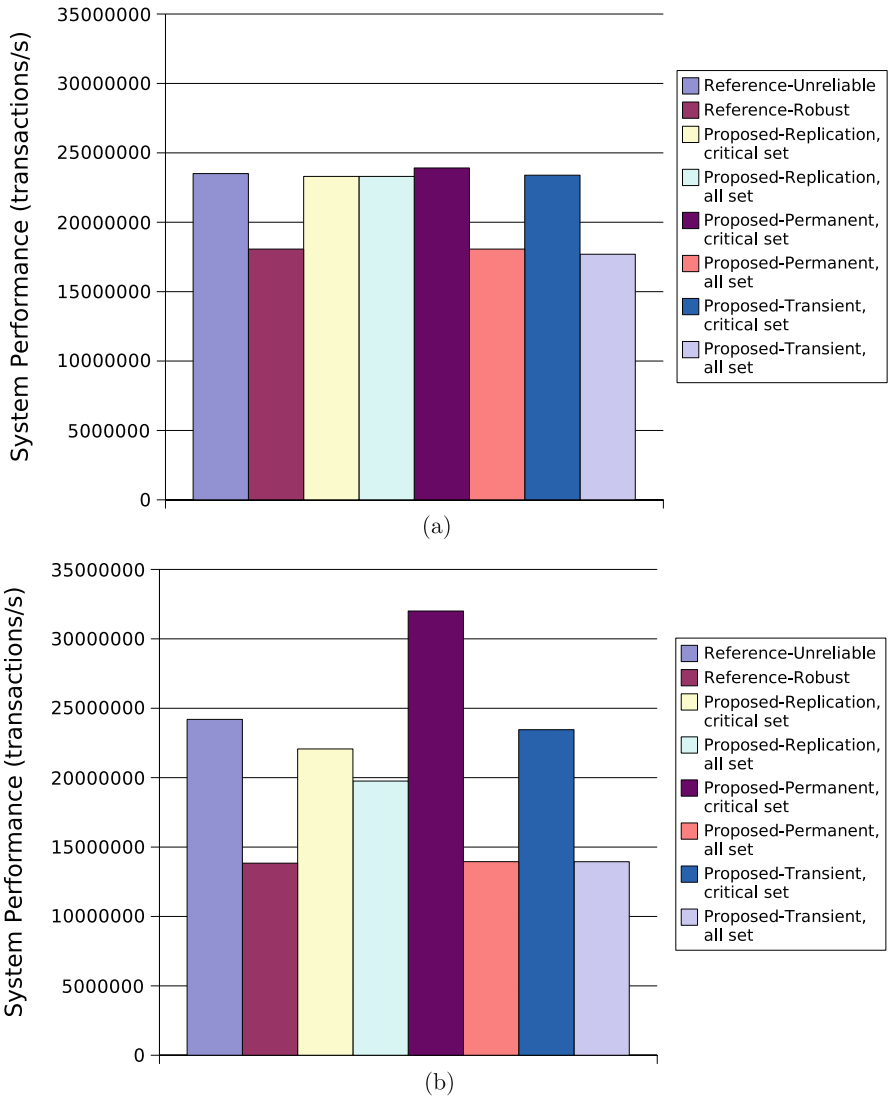


Fig. 11.7 Impact of adding reliability support on the star, with different sizes of the critical data set for (**a**) VTC and (**b**) 3DR

11.4.4 Synthesis Results

Regarding the modifications in the NoC to support a backup memory, four changes are needed: (i) the NI associated to the main memory must be augmented, (ii) the backup memory needs an extra (plain) target NI device, (iii) the initiator NI becomes a bit more complex, and (iv) extra links and switch ports may be needed for routing data to the backup memory.

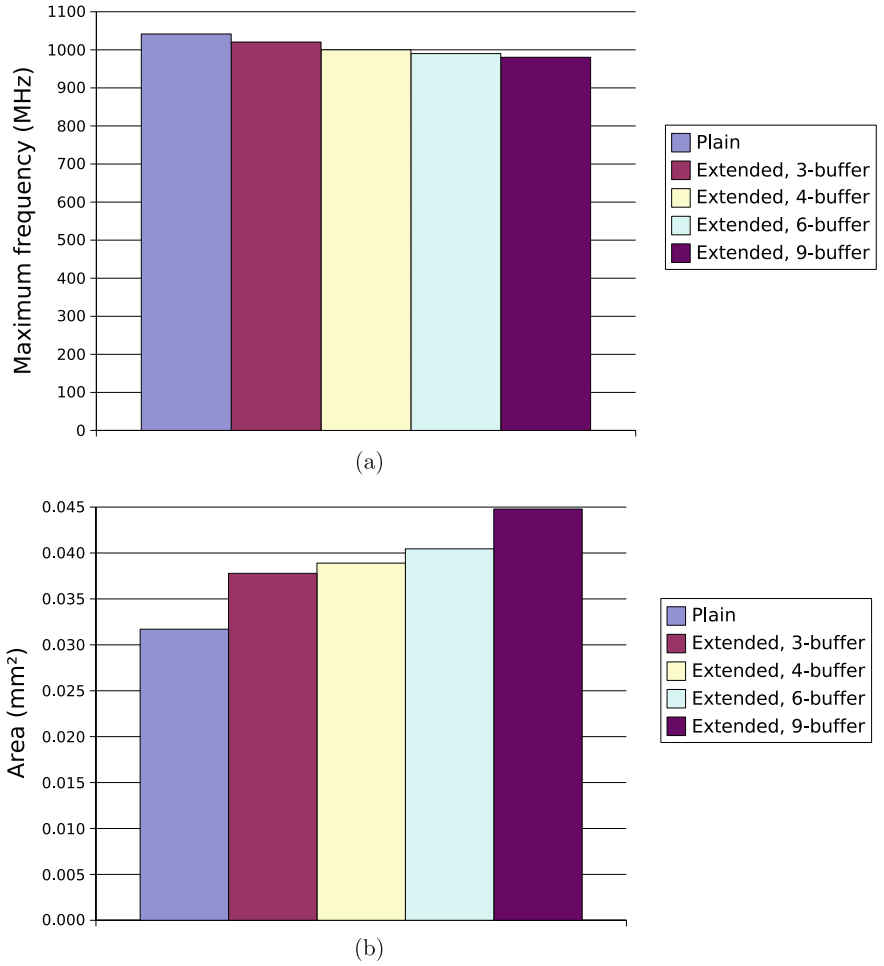


Fig. 11.8 Comparative (**a**) maximum operating frequency and (**b**) area for plain and extended target NIs

To assess the silicon cost of adding reliability features to the NIs, we perform a full synthesis of both the original and extended ones with a 0.13 µm UMC technology library. Initiator NIs (not depicted) experience no operating frequency penalty to support the reliability functionality, while area increases by about 7% (0.031 mm^2 against 0.029 mm^2). Results for target NIs are plotted in Figure 11.8(a) and (b). The two figures show the results for a reference target NI with a 4-slot output buffer in the response channel, contrasted against the extended target NIs, having 3- to 9-slot buffers in the extra redundancy channel. The impact on maximum achievable frequency is just of 2% to 6%. This penalty can be negligible in a NoC where the maximum frequency is determined by the switches and not by the NIs [88]. By

adopting a buffer as deep as that of the response channel (4 slots), area is increased from 0.032 mm^2 to 0.039 mm^2.

As a result, the area cost due to NI changes is 0.041 mm^2. Overall, even including other possible overheads in the NoC (i.e., extra ports in switches and extra links), the final overhead is still small in comparison to the area of the extra backup memory bank itself, which can take on average 1 mm^2 of area for a 32 kB on-die SRAM in 0.13 μm technology.

11.5 Summary

One of the main challenges for designers will be the deployment of fault tolerant architectures. In this chapter, we have presented a complete approach to countering transient and permanent failures in on-chip memories, by taking advantage of the communication infrastructure provided by the reliable NoC backbone presented in the preceding chapters. The design is based on modular extensions of the network interfaces of the cores, and is transparent to the software designer. The only activity required by the programmer is minimal code annotation to tell the compiler which parts of the data set are critical. The extensions are integrated within the NoC mapping flow, which transparently handles instantiation issues. The experimental results show that the proposed approach has a very limited area overhead compared to non-reliable designs, while being scalable for any number of cores. The experiments also show the power of NoCs in handling reliability and scalability challenges of SoCs.

Chapter 12
Conclusions and Future Directions

In this chapter, we summarize the major contributions of this thesis and show how the NoC design methods and reliability mechanisms are integrated in the design flow.

12.1 Putting It All Together

The reliability enhanced Netchip tool flow is presented in Figure 12.1. Initially, along with the application traffic characteristics, the system reliability specifications and requirements are also taken as inputs to the tool flow. In Chapter 11, we had presented the mechanisms to provide NoC support for using back-up memories. The number of back-up memories used and the additional traffic flow rates related to them are given as part of the system reliability specifications. In the Netchip flow, we automatically design the NoC to meet the bandwidth demands of the additional traffic that is generated due to the use of multiple memories. The tool flow also ensures that the traffic streams to the back-up memories do not create deadlocks with the other traffic flows.

We had presented methods to achieve tolerance against temporary errors in Chapters 8 and 9. The *T-error* scheme presented in Chapter 8, is required when multi-bit timing errors can occur in the system. Based on the error-rates of the system, which is given as part of the system reliability specifications, we determine whether *T-error* scheme is needed for the NoC. We also determine the most power optimal encoding needed to tolerate other transient errors in the system (such as soft-errors). To determine the best scheme, we use the analysis method presented in Chapter 9.

Once the error recovery schemes needed for the NoC components are determined, we proceed with the design of the NoC topology. The systems that utilize NoCs can be broadly classified into two types: *Application-Specific Systems-on-Chip* (ASSoCs) and *Chip Multiprocessors* (CMPs). In ASSoCs, single or a fixed set of applications are statically mapped onto the different processor and hardware cores in the design. In CMPs, software tasks are dynamically assigned to the cores. We distinguish three major application classes for NoCs here: (1) ASSoCs that run a single application, (2) ASSoCs that run multiple applications, and (3) CMPs that run general software tasks.

For designing ASSoCs that run a single application, we apply the SUNMAP and SUNFLOOR tools presented in Chapters 4 and 5. The SUNMAP tool is used to design a standard topology (such as a mesh, torus) for the application, while the SUNFLOOR tool designs a custom topology. For designing ASSoCs that support multiple applications, we apply the extended synthesis procedure presented in Chapter 6. When the design is a CMP that runs software tasks, we apply the synthesis

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

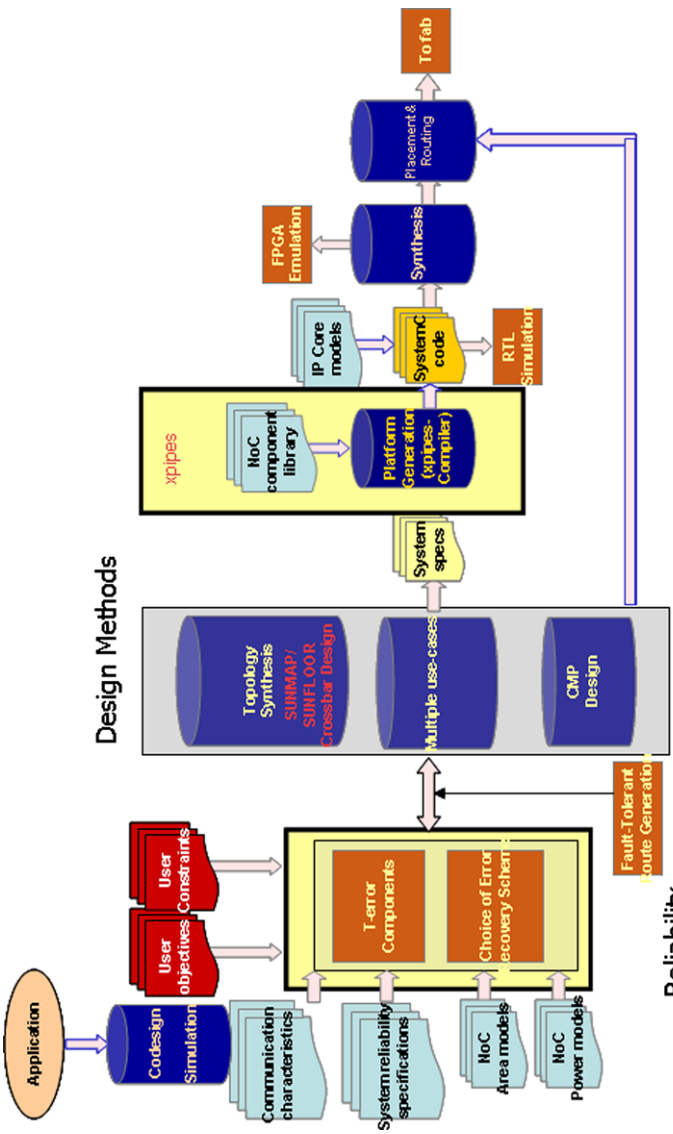


Fig. 12.1 Reliability enhanced Netchip design flow

approach presented in Chapter 7. The individual crossbar switches of the NoC can be further optimized using the method presented in Chapter 2.

When choosing paths for traffic flows, we determine whether it is efficient to apply the multipath routing presented in Chapter 10. We also determine the fault-tolerance level achievable using such a method. We iterate between the error recovery methods chosen in the previous steps with the multipath method, until a design with least area-power overhead that still satisfies the reliability constraints is obtained.

Once a NoC topology that satisfies the reliability constraints is obtained, we proceed to generate the RTL design of the NoC components, as presented in Chapter 3. For this, we use $\times$pipes, a library of SystemC soft macros for the network components, and the associated tool $\times$pipesCompiler to generate the entire design. We proceed with the RTL simulation, synthesis, emulation and layout of the design using standard tool chains. This automates some of the most critical and time intensive NoC design steps such as topology synthesis, core mapping, crossbar sizing, route generation, resource reservation, achieving fault-tolerance, RTL code, and layout generation.

Bibliography

1. W. Wolf, “The future of multiprocessor systems-on-chips”, Proc. DAC, pp. 681–685, June 2004.
2. More information on AMBA AXI from ARM corporation is available at http://www.arm.com/products/solutions/AMBAHomePage.html.
3. More information on STBus from STMicrolectronics is available at http://www.st.com/stonline/prodpres/dedicate/soc/cores/stbus.htm.
4. http://www.sonicsinc.com/.
5. W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, M. Diaz-Nava, “Component-based design approach for multi-core SoCs”, Proc. DAC, pp. 789–794, June 2002.
6. A. Hansson et al., “A unified approach to constrained mapping and routing on network-on-chip architectures”, pp. 75–80, Proc. ISSS, 2005.
7. P. Guerrier, A. Greiner, “A generic architecture for on-chip packet switched interconnections”, DATE 2000, pp. 250–256, March 2000.
8. E. B. Van der Tol, E. G. T. Jaspers, “Mapping of MPEG-4 decoding on a flexible architecture platform”, SPIE 2002, pp. 1–13, January 2002.
9. W. J. Dally, S. Lacy, “VLSI architecture: past, present and future”, Conf. Adv. Research in VLSI, pp. 232–241, 1999.
10. F. Karim et al., “On-chip communication architecture for OC-768 network processors”, Design Automation Conference, pp. 678–678, June 2001.
11. S. Murali et al., “Mapping and physical planning of networks on chip architectures with quality-of-service guarantees”, Proc. ASPDAC, 2005.
12. J. Kim et al., “A low latency router supporting adaptivity for on-chip interconnects”, Proc. DAC, June 2005.
13. J. Hu, R. Marculescu, “DyAD—smart routing for networks-on-chip”, Proc. DAC, June 2004.
14. Y. Aydogan et al., “Adaptive source routing in multistage interconnection networks”, Proceedings of the 10th International Parallel Processing Symposium, 1996.
15. S. Manolache et al., “Fault and energy-aware communication mapping with guaranteed for applications implemented on NoC”, Proc. DAC, 2005.
16. W. J. Dally et al., “The Avici terabit switch/router”, Proc. Hot Interconnects, August 1998.
17. B. G. Stunkel et al., “The SP2 communication subsystem”, IBM Technical Report, August 22, 1994.
18. G. De Micheli, “Synthesis and Optimization of Digital Circuits”, McGraw-Hill, New York, 1994.
19. S. Murali, D. Atienza, L. Benini, G. De Micheli, “A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip”, DAC, 2006.
20. L. Benini, G. De Micheli, “Networks on chips: a new SoC paradigm”, IEEE Computers, pp. 70–78, January 2002.
21. D. Wingard, “MicroNetwork-based integration for SoCs”, Proc. DAC, pp. 673–677, January 2001.
22. W. Dally, B. Towles, “Route packets, not wires: on-chip interconnection networks”, Proc. DAC, pp. 684–689, June 2001.
23. M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli, “Addressing the system-on-a-chip interconnect woes through communication-based design”, Proc. DAC, pp. 667–672, June 2001.
24. H. Zhang et al., “A 1 V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing”, IEEE Journal of Solid State Circuits, Vol. 35, No. 11, pp. 1697–1704, 2000.

S. Murali, *Designing Reliable and Efficient Networks on Chips,*
Lecture Notes in Electrical Engineering 34,

25. X. Zhu, S. Malik, “A hierarchical modeling framework for on-chip communication architectures”, ICCD 2002, pp. 663–671, November 2002.
26. I. Saastamoinen, D. Siguenza-Tortosa, J. Nurmi, “Interconnect IP node for future system-on-chip designs”, Proc. of The First IEEE International Workshop on Electronic Design, Test and Applications, pp. 116–120, January 2002.
27. S. J. Lee et al., “An 800 MHz star-connected on-chip network for application to systems on a chip”, Digest of Technical Papers, ISSCC 2003, pp. 468–469, February 2003.
28. A. Jantsch, H. Tenhunen, “Networks on Chip”, Kluwer Academic, Dordrecht, 2003.
29. E. D. Taillard, “Robust tabu search for the quadratic assignment problem”, Parallel Computing, Vol. 17, pp. 443–455, 1991.
30. E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, E. Waterlander, “Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip”, Proc. DATE, pp. 350–355, March 2003.
31. P. Guerrier, A. Greiner, “A generic architecture for on-chip packet switched interconnections”, Proc. DATE, pp. 250–256, March 2000.
32. S. Kumar, A. Jantsch, M. Millberg, J. Oberg, J.-P. Soininen, M. Forsell, K. Tiensyrja, A. Hemani, “A network on chip architecture and design methodology”, ISVLSI 2002, pp. 105–112, 2002.
33. D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, G. De Micheli, “NoC synthesis flow for customized domain specific multiprocessor systems-on-chip”, IEEE Transactions on Parallel and Distributed Systems, Vol. 16, No. 2, pp. 113–129, 2005.
34. S. Stergiou et al., “×pipesLite: a synthesis oriented design library for networks on chips”, Proc. DATE, pp. 1188–1193, 2005.
35. T. Yen, W. Wolf, “Communication synthesis for distributed embedded systems”, Proc. ICCAD, pp. 288–294, November 1995.
36. J. Daveau, T. Ismail, A. Jerraya, “Synthesis of system-level communication by an allocation based approach”, Proc. ISSS, pp. 150–155, September 1995.
37. M. Gasteier, M. Glesner, “Bus-based communication synthesis on system level”, ACM Transactions on Design Automation of Electronic Systems, ACM TODAES, Vol. 4, No. 1, pp. 1–11, 1999.
38. K. Ryu, V. Mooney, “Automated bus generation for multiprocessor SoC design”, Proc. DATE, pp. 282–287, March 2003.
39. K. Lahiri, A. Raghunathan, S. Dey, “Design space exploration for optimizing on-chip communication architectures”, IEEE TCAD, Vol. 23, No. 6, pp. 952–961, 2004.
40. K. Lahiri, A. Raghunathan, G. Lakshminarayana, S. Dey, “Design of high-performance system-on-chips using communication architecture tuners”, IEEE TCAD, Vol. 23, No. 5, pp. 620–636, 2004.
41. R. Ravi et al., “Approximation algorithms for degree-constrained minimum-cost network design problems”, Algorithmica, Vol. 31, No. 1, pp. 58–78, 2001.
42. E. Bolotin, I. Cidon, R. Ginosar, A. Kolodny, “QNoC: QoS architecture and design process for network on chip”, The Journal of Systems Architecture, Vol. 50, No. 2–3, pp. 105–128, 2004.
43. J. Hu, R. Marculescu, “Energy-aware mapping for tile-based NOC architectures under performance constraints”, Proc. ASPDAC, pp. 233–239, January 2003.
44. J. Hu, R. Marculescu, “Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures”, Proc. DATE, pp. 10688–106993, March 2003.
45. S. Murali, G. De Micheli, “Bandwidth constrained mapping of cores onto NoC architectures”, Proc. DATE, pp. 20896–20902, February 2004.
46. S. Murali, G. De Micheli, “SUNMAP: a tool for automatic topology selection and generation for NoCs”, Proc. DAC, pp. 914–919, June 2004.
47. A. Pinto, L. Carloni, A. Sangiovanni-Vincentelli, “Constraint-driven communication synthesis”, Proc. DAC, pp. 783–788, June 2002.
48. A. Pinto, L. Carloni, A. Sangiovanni-Vincentelli, “Efficient synthesis of networks on chip”, Proc. ICCD, pp. 146–150, October 2003.

49. T. Ahonen, D. Signza-Tortosa, H. Bin, J. Nurmi, “Topology optimization for application specific networks on chip”, Proc. SLIP, pp. 53–60, February 2004.
50. K. Srinivasan, K. Chatha, G. Konjevod, “An automated technique for topology and route generation of application specific on-chip interconnection networks”, Proc. ICCAD, pp. 231–237, November 2005.
51. W. H. Ho, T. M. Pinkston, “A methodology for designing efficient on-chip interconnects on well-behaved communication patterns”, Proc. HPCA, pp. 377–388, February 2003.
52. S. Murali, G. De Micheli, “An application-specific design methodology for STBus crossbar generation”, Proc. DATE, pp. 1176–1181, March 2005.
53. S. Pasricha, N. Dutt, E. Bozorgzadeh, M. Ben-Romdhane, “Floorplan-aware automated synthesis of bus-based communication architectures”, Proc. DAC, pp. 65–70, June 2005.
54. S. Murali, M. Coenen, A. Radulescu, K. Goossens, G. De Micheli, “Mapping and configuration methods for multi-use-case networks on chips”, Proc. ASPDAC, pp. 146–151, January 2006.
55. S. Murali, M. Coenen, A. Radulescu, K. Goossens, G. De Micheli, “A methodology for mapping multiple use-cases onto networks on chips”, Proc. DATE, March 2006.
56. S. Pasricha, N. Dutt, M. Ben-Romdhane, “Constraint-driven bus matrix synthesis for MPSoC”, Proc. ASPDAC, pp. 30–35, January 2006.
57. M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, R. Zafalon, “Analyzing on-chip communication in a MPSoC environment”, Proc. DATE, pp. 20752–20757, February 2004.
58. R. Ho, K. Mai, M. Horowitz, “The future of wires”, Proc. of the IEEE, pp. 490–504, April 2001.
59. ILOG CPLEX, http://www.ilog.com/products/cplex/.
60. V. Vaizirani, “Approximation Algorithms”, Springer, Berlin, 2004.
61. M. Dallosso et al., “`xpipes`: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs”, ICCD, pp. 536–539, 2003.
62. A. Jalabert et al., “`xpipesCompiler`: a tool for instantiating application specific networks on chips”, Proc. DATE, 2004.
63. J. G. Kim, Y. D. Kim, “A linear programming-based algorithm for floorplanning in VLSI design”, IEEE Transactions on CAD, Vol. 22, No. 5, pp. 584–592, 2003.
64. M. Garey, D. Johnson, “Computers and Intractability, a Guide to the Theory of Np-Completeness”, Freeman, New York, 1979.
65. D. Culler, J. P. Singh, A. Gupta, “Parallel Computer Architecture, a Hardware/Software Approach”, Morgan Kaufmann, San Mateo, 1999.
66. K. Compton, S. Hauck, “Reconfigurable computing: a survey of system and software”, ACM Computing Surveys, Vol. 34, No. 2, pp. 171–210, 2002.
67. D. Whelihan, H. Schmit, “Memory optimization in single chip network fabrics”, Proc. DAC 2002, pp. 530–535, June 2002.
68. H. S. Wang et al., “Orion: a power-performance simulator for interconnection networks”, MICRO, November 2002.
69. D. Wiklund, D. Liu, “SoCBUS: switched network on chip for hard real time embedded systems”, Proc. International Parallel and Distributed Processing Symposium 2003, pp. 78–85, 2003.
70. L. P. Carloni, K. L. McMillan, A. L. Sangiovanni Vincentelli, “Theory of latency-insensitive design”, IEEE Transactions on CAD of ICs and Systems, Vol. 20, No. 9, pp. 1059–1076, 2001.
71. Tensilica Offload Engine, http://www.tensilica.com/html/pr_2003_05_12.html.
72. S. Mitra, N. Seifert, M. Zhang, Q. Shi, K. S. Kim, “Robust system design with built-in soft error resilience”, IEEE Computer, Vol. 38, No. 2, pp. 43–52, 2005.
73. G. De Micheli, L. Benini, “Networks on Chips: Technology and Tools”, First Edition, Morgan Kaufmann, San Mateo, 2006.
74. J. Liang, S. Swaminathan, R. Tessier, “aSOC: a scalable, single-chip communications architecture”, The IEEE International Conference on Parallel Architectures and Compilation Techniques, pp. 524–529, October 2000.

75. C. A. Zeferino, A. A. Susin, “SoCIN: a parametric and scalable network-on-chip”, Proceedings 16th Symposium on Integrated Circuits Systems 03, pp. 169–174, September 2003.
76. T. Bjerregaard, J. Spars, “A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chips”, Proc. DATE 2005, pp. 1226–1231, March 2005.
77. J. Hu et al., “System-level point-to-point communication synthesis using floorplanning information”, Proc. ASPDAC ’02.
78. J. Xu, W. Wolf, J. Henkel, S. Chakradhar, “A design methodology for application-specific networks-on-chip”, ACM Transactions on Embedded Computing Systems (TECS), Vol. 5, No. 2, pp. 263–280, 2006.
79. T. T. Ye et al., “Analysis of power consumption on switch fabrics in network routers”, Proc. DAC ’03.
80. N. Banerjee et al., “A power and performance model for network-on-chip architectures”, Proc. DATE ’04.
81. G. Palemoro, C. Silvano, “PIRATE: a framework for power/performance exploration of network-on-chip architectures”, PATMOS, 2004.
82. S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, L. Raffo, “Designing application-specific networks-on-chips with floorplan information”, ICCAD, November 2006.
83. S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, L. Raffo, “Designing message-dependent deadlock free networks on chips for application-specific systems on chips”, VLSI-SoC, October 2006.
84. Y. H. Song, T. M. Pinkston, “A progressive approach to handling message-dependent deadlock in parallel computer systems”, IEEE TPDS, Vol. 14, No. 3, pp. 259–275, 2003.
85. G. Chiu, “The odd-even turn model for adaptive routing”, IEEE TPDS, Vol. 11, No. 7, pp. 729–738, 2000.
86. J. Duato, “A new theory of deadlock-free adaptive routing in wormhole networks”, IEEE TPDS, Vol. 8, No. 8, pp. 790–802, 1997.
87. D. Starobinksi et al., “Application of network calculus to general topologies using turn-prohibition”, IEEE/ACM Transactions on Networking, Vol. 11, No. 3, pp. 411–421, 2003.
88. F. Angiolini et al., “Contrasting a NoC and a traditional interconnect fabric with layout awareness”, Proc. DATE, pp. 124–129, 2006.
89. P. Meloni, S. Carta, R. Argiolas, L. Raffo, F. Angiolini, “Area and power modeling methodologies for networks-on-chip”, Proc. of Nanonets, September 2006.
90. A. Pullini, F. Angiolini, D. Bertozzi, L. Benini, “Fault tolerance overhead in network-on-chip flow control schemes”, Proc. SBCCI, pp. 224–229, 2005.
91. www.ocpip.org.
92. S. N. Adya, I. L. Markov, “Fixed-outline floorplanning: enabling hierarchical design”, IEEE Transactions on VLSI Systems, Vol. 11, No. 6, pp. 1120–1135, 2003. URL: http://vlsicad.eecs.umich.edu/BK/parquet/.
93. B. Hendrickson, R. Leland, “The Chaco user’s guide: Version 2.0”, Sandia Tech Report SAND94–2692, 1994. URL: http://www.cs.sandia.gov/~bahendr/chaco.html.
94. W. J. Dally, B. Towles, “Principles and Practices of Interconnection Networks”, Morgan Kaufmann, San Mateo, 2003.
95. T. H. Cormen et al., “Introduction to Algorithms”, MIT Press, Cambridge, 1990.
96. K. Goossens et al., “A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification”, DATE, pp. 1182–1187, 2005.
97. J. Rabaey et al., “Digital Integrated Circuits”, Prentice Hall, New York, 2002.
98. www.synopsys.com.
99. www.cadence.com.
100. M. Taylor et al., “The raw microprocessor: a computational fabric for software circuits and general purpose programs”, IEEE Micro, April 2002.
101. K. Mai et al., “Smart memories: a modular reconfigurable architecture”, Proc. ISCA, June 2000.
102. K. Sankaralingam et al., “Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture”, IEEE Micro, November/December 2003.

103. R. Kalla et al., “IBM Power5 chip: a dual-core multithreaded processor”, IEEE Micro, March/April 2004.
104. T. Bjerregaard, J. Sparso, “Virtual channel designs for guaranteeing bandwidth in asynchronous network-on-chip”, Proc. NORCHIP, pp. 269–272, November 2004.
105. M. Millberg et al., “Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip”, Proc. DATE, 2004.
106. H. Wang et al., “A technology-aware and energy-oriented topology exploration for on-chip networks”, Proc. DATE, 2005.
107. W. J. Dally, “Performance analysis of k-ary n-cube interconnection networks”, IEEE TPDS, pp. 775–785, June 1990.
108. B. Towles, W. J. Dally, “Worst-case traffic for oblivious routing functions”, Proc. SPAA, pp. 1–8, August 2002.
109. B. Towles et al., “Throughput-centric routing algorithm design”, Proc. SPAA, pp. 200–209, June 2003.
110. M. B. Taylor, “The RAW processor specification”, available at http://cagwww.lcs.mit.edu/raw/documents/index.html.
111. R. Ho, K. Mai, M. Horowitz, “Efficient on-chip global interconnects”, IEEE Symposium on VLSI Circuits, June 2003.
112. V. Karamcheti, A. A. Chien, “Do Faster Routers Imply Faster Communication?”, Lecture Notes in Computer Science, Vol. 853, pp. 1–15, Springer, Berlin, 1994.
113. D. Ernst, N. S. Kim, S. Pant, S. Das, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, T. Mudge, “Razor: a low-power pipeline based on circuit-level timing speculation”, Proc. of the International Symposium on Microarchitecture, pp. 7–18, December 2003.
114. T. Austin, D. Blaauw, T. Mudge, K. Flautner, “Making typical silicon matter with razor”, IEEE Computer, Vol. 37, No. 3, pp. 57–65, 2004.
115. A. Uht, “Going beyond worst-case specs with TEAtime”, IEEE Computer, Vol. 37, No. 3, pp. 51–56, 2004.
116. P. Franco, E. J. McCluskey, “On line delay testing of digital circuits”, Proc. VLSI Test Symposium, pp. 167–173, 1994.
117. M. Favalli, C. Metra, “Low-level error recovery mechanism for self-checking sequential circuit”, Proc. DFT, pp. 234–242, October 1997.
118. M. Singh, S. M. Nowick, “MOUSETRAP: ultra-high-speed transition signaling asynchronous pipelines”, Proc. ICCD, pp. 9–17, September 2001.
119. E. Dupont, M. Nicolaidis, P. Rohr, “Embedded robustness IPs for transient error free ICs”, IEEE Design and Test of Computers, Vol. 19, No. 3, pp. 56–70, 2002.
120. W. J. Dally, J. W. Poulton, “Digital Systems Engineering”, Cambridge University Press, Cambridge, 1998.
121. Y. Eo, S. Shin, W. Eisenstadt, J. Shim, “A decoupling technique for efficient timing analysis of VLSI interconnects with dynamic current switching”, IEEE Transactions on CAD, Vol. 23, No. 9, pp. 1321–1337, 2004.
122. D. Wang, W. McNall, “A statistical model based ASIC skew selection method”, IEEE Workshop on Microelectronics and Electron Devices, pp. 64–66, 2004.
123. L. Chen, M. Marek-Sadowska, F. Brewer, “Coping with buffer delay change due to power and ground noise”, Proc. DAC, pp. 860–865, June 2002.
124. P. J. Restle, K. A. Jenkins, A. Deutsch, P. W. Cook, “Measurement and modeling of on-chip transmission line effects in a 400 MHz microprocessor”, IEEE Journal of Solid-State Circuits, Vol. 33, No. 4, pp. 662–665, 1998.
125. “Berkeley predictive technology model”, available at http://www-device.eecs.berkeley.edu/~ptm/.
126. R. Hegde, N. R. Shanbhag, “Toward achieving energy efficiency in presence of deep submicron noise”, IEEE Transactions on VLSI Systems, Vol. 8, No. 4, pp. 379–391, 2000.
127. D. Bertozzi, L. Benini, G. De Micheli, “Low power error-resilient encoding for on-chip data buses”, Proc. DATE, pp. 102–109, March 2002.

128. P. Vellanki, N. Banerjee, K. Chatha, “Quality-of-service and error control techniques for network on chip architectures”, Proc. GLSVLSI, pp. 45–50, April 2004.
129. H. Zimmer, A. Jantsch, “A fault model notation and error-control scheme for switch-to-switch buses in a network-on-chip”, Proc. ISSS/CODES, pp. 188–193, September 2003.
130. S. Murali, T. Theocharides, N. VijayKrishnan, M. J. Irwin, L. Benini, G. De Micheli, “Analysis of error recovery schemes for networks-on-chips”, IEEE Design and Test of Computers, Vol. 22, No. 5, pp. 434–442, 2005.
131. M. R. Stan, W. P. Burleson, “Bus-invert coding for lowpower I/O”, IEEE Transactions on VLSI, Vol. 3, No. 1, pp. 49–58, 1995.
132. L. Li, N. VijayKrishnan, M. Kandemir, M. J. Irwin, “A cross-talk aware interconnect with variable cycle transmission”, Proc. DATE, pp. 1012–1017, February 2004.
133. K. Patel, I. Markov, “Error-correction and cross-talk avoidance in DSM busses”, IEEE Transactions on VLSI, Vol. 12, No. 10, pp. 1076–1080, 2004.
134. S. Srinivas, N. R. Shanbhag, “Coding for system-on-chip networks: a unified framework”, Proc. DAC, pp. 103–106, June 2004.
135. K. Hirose, H. Yasuura, “A bus delay reduction technique considering cross-talk”, Proc. DATE, pp. 441–445, March 2000.
136. P. Sotiriadis, “Interconnect modeling and optimization in deep sub-micron technologies”, Ph.D. Dissertation, Massachusetts Institute of Technology, 2002.
137. R. Tamhankar, S. Murali, G. De Micheli, “Performance driven reliable link design for networks on chips”, Proc. ASPDAC, pp. 749–754, January 2005.
138. R. Marculescu, “Networks-on-chip: the quest for on-chip fault-tolerant communication”, Proc. IEEE ISVLSI, pp. 8–12, February 2003.
139. F. Worm, P. Ienne, P. Thiran, G. De Micheli, “A robust self-calibrating transmission scheme for on-chip networks”, IEEE Transactions on VLSI, Vol. 13, No. 1, pp. 126–139, 2005.
140. M. Pirretti, G. Link, R. Brooks, N. VijayKrishnan, M. Kandemir, M. J. Irwin, “Fault tolerant algorithms for network-on-chip interconnect”, Proc. ISVLSI, pp. 46–51, February 2004.
141. T. M. Austin et al., “Opportunities and challenges for better than worstcase design”, Proc. ASP-DAC, 2005.
142. H. Wang et al., “Systematic analysis of energy and delay impact of very deep submicron process variability effects in embedded sram modules”, DATE, 2005.
143. S. Xiao et al., “A generalization of the single b-bit byte error correcting and double bit error detecting codes for high-speed memory systems”, IEEE Transactions on Computer, 1996.
144. I. Sodagar et al., “Scalable wavelet coding for synthetic and natural hybrid images”, IEEE Transactions on Circuits and Systems for Video Technology, 1999.
145. M. Pollefeys et al., “Metric 3D surface reconstruction from uncalibrated image sequences”, Proc. SMILE, 1998.
146. V. Narayanan, Y. Xie, “Computing in the presence of soft errors”, Tutorial, ASPLOS XI, October 2004.
147. N. R. Shanbhag, “A mathematical basis for basis for power-reduction in digital VLSI systems”, IEEE Transactions on Circuits and Systems, Part II, Vol. 44, No. 11, pp. 935–951, 1997.
148. H. S. Wang et al., “Power-driven design of router micro-architectures in on-chip networks”, Proc. of the 36th MICRO, November 2003.

The list of papers in which the author has been involved and that are related to the chapters in this book:

Journal Publications

J1. Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna Tamhankar, Stergios Stergiou, Luca Benini, Giovanni De Micheli, “NoC synthesis flow for customized domain spe-

cific multiprocessor systems-on-chip", IEEE Transactions on Parallel and Distributed Systems, Vol. 16, No. 2, pp. 113–129, Feb. 2005.
J2. Srinivasan Murali, Theocharis Theocharides, Luca Benini, Giovanni De Micheli, N. Vijaykrishan, Mary Jane Irwin, "Analysis of error recovery schemes for networks on chips", IEEE D&T, Vol. 22, No. 5, pp. 434–442, Sep./Oct. 2005.
J3. Rutuparna Tamhankar, Srinivasan Murali, Stergios Stergiou, Antonio Pullini, Federico Angiolini, Luca Benini, and Giovanni De Micheli, "Timing error tolerant network-on-chip design methodology," IEEE Transactions on Computer Aided Design, Vol. 26, No. 7, pp. 1297–1310, July 2007.
J4. Srinivasan Murali, Paolo Meloni, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, Luigi Raffo, "Synthesis of predictable networks-on-chip based interconnect architectures for chip multi-processors," IEEE Transactions on VLSI, Vol. 15, No. 8, pp. 869–880, August 2007.
J5. Srinivasan Murali, Luca Benini, Giovanni De Micheli, "An application-specific design methodology for on-chip crossbar generation," IEEE Transactions on Computer Aided Design, Vol. 26, No. 7, pp. 1283–1296, July 2007.
J6. Srinivasan Murali, David Atienza, Luca Benini, and Giovanni De Micheli, "A method for routing packets across multiple paths in nocs with in-order delivery and fault-tolerance guarantees," VLSI-Design Journal, Hindawi Publications, 2007.

Conference Publications

C1. Antoine Jalabert, Srinivasan Murali, Luca Benini, Giovanni De Micheli, "xpipesCompiler: a tool for instantiating application specific networks on chip", Proc. DATE 2004.
C2. Srinivasan Murali, Giovanni De Micheli, "Bandwidth constrained mapping of cores onto networks on chips", Proc. DATE 2004.
C3. Srinivasan Murali, Giovanni De Micheli, "SUNMAP: a tool for automatic topology selection and generation for NoCs", Proc. DAC 2004.
C4. Srinivasan Murali, Luca Benini, Giovanni De Micheli, "Mapping and physical planning of networks on chip architectures with quality-of-service guarantees", Proc. ASPDAC 2005.
C5. Rutuparna Tamhankar, Srinivasan Murali, Giovanni De Micheli, "Performance driven reliable link design for networks on chips", Proc. ASPDAC 2005.
C6. Srinivasan Murali, Giovanni De Micheli, "An application specific design methodology for stbus crossbar generation", Proc. DATE 2005.
C7. Srinivasan Murali, Martijn Coenen, Andrei Radulescu, Kees Goossens, Giovanni De Micheli, "Mapping and configuration methods for multi-use-case networks on chips", ASP-DAC 2006.
C8. Srinivasan Murali, Martijn Coenen, Andrei Radulescu, Kees Goossens, Giovanni De Micheli, "A methodology for mapping multiple use-cases onto networks on chips", DATE 2006.
C9. Srinivasan Murali, David Atienza, Luca Benini, Giovanni De Micheli, "A multi-path routing strategy with guaranteed in-order packet delivery and fault tolerance for networks on chips", DAC 2006.
C10. Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, Luigi Raffo, "Designing message-dependent deadlock free networks on chips for application-specific systems on chips", VLSI-SoC 2006.
C11. Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, Luigi Raffo, "Design of application-specific networks on chips with floorplan information", Proc. ICCAD 06.

C12. Federico Angiolini, David Atienza, Srinivasan Murali, Luca Benini, Giovanni De Micheli, “Reliability support for on-chip memories using networks-on-chips”, Proc. ICCD 06.
C13. Srinivasan Murali, Rutuparna Tamhankar, Federico Angiolini, Antonio Pullini, David Atienza, Luca Benini, Giovanni De Micheli, “Comparison of a timing-error tolerant scheme with a traditional re-transmission mechanism for networks on chips,” International Symposium on Systems on Chips, 2006.