Achieving Functional Correctness in Large Interconnect Systems

by

Rawan Abdel Khalek

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2015

Doctoral Committee:
Professor Valeria M. Bertacco, Chair
Assistant Research Scientist Reetuparna Das
Professor Mingyan Liu
Professor Scott Mahlke

© Rawan Abdel Khalek

All Rights Reserved
2015

To my family

Acknowledgments

I would like to thank my advisor, Professor Valeria Bertacco, for her continued support throughout my time at the University of Michigan. She offered me the freedom to explore and develop new and interesting research directions, while providing the guidance and knowledge to shape and develop my research work. I am also grateful for my committee members, Professor Mingyan Liu, Professor Scott Mahlke and Dr. Reetuparna Das, for their support and their valuable technical insights and feedback. I would also like to acknowledge a number of people in the CSE department, who I was fortunate to get to know over the years and who made this experience a memorable one. Thank you Ritesh Parikh, Biruk Mammo, Doowon Lee, Andrew DeOrio, Andrea Pellegrini, Debapriya Chatterjee, Ilya Wagner, Jason Clemons, Joseph Greathouse, Animesh Jain, and Faissal Sleiman. Finally, I would like to thank my family for always supporting and believing in me.

Preface

The design of digital systems has been driven by the continuous shrinking of transistor sizes, which has allowed the integration of more functionality and components onto a single chip. What started out as single core microprocessors has now become large chip-multiprocessors. In today's market, chip-multiprocessors (CMPs), consisting of up to hundreds of processor cores, are being developed to target high performance computing applications and highly parallel programs. Moreover, the semi-conductor industry has been progressing towards developing systems-on-chip (SoCs), in which different types components, ranging from processor cores, to accelerators, to signal processing units, as well as others, are integrated together onto a single chip. SoC designs are becoming highly ubiquitous, making their way to many electronic devices, such as tablets and smartphones, cars, gaming and multimedia set-top boxes.

In the context of such highly integrated CMPs and SoCs, the communication infrastructure is a key component of these systems. The large number of on-chip components requires an equally parallel interconnect design that is capable of delivering the necessary bandwidth and performance.Therefore, networks-on-chip (NoCs) emerged as the interconnect model of choice for such systems. The distributed and flexible nature of NoCs inherently makes them more scalable and critical in meeting the high on-chip communication demands resulting from this extreme integration. However, the continuous growth in size and complexity of NoC designs, coupled with short-time to market windows of modern computing systems, have placed a significant burden on the functional verification of the interconnect subsystem. Verification efforts during a product's developmenet are riddled with challenges that prevent the efficient and exhaustive validation of the interconnect. This in turn leads to potential design bugs escaping these verification efforts into the interconnect of released products, threatening the runtime operation of these designs. The manifestation of design bugs in the interconnect infrastructure not only affects the correctness of communication, but can compromise ther performance of the system, the integrity of users' data and the correct functioning of applications.

This dissertation presents solutions that address the functional verification of large

network-on-chip interconnects, through-out the various stages of its development, as well as during its runtime operation. Starting in the pre-silicon stage of designing and implementing the interconnect and the rest of the system, emulation platforms are heavily utilized as faster and more powerful alternatives to software-only simulation frameworks. These validation efforts continue through to the post-silicon stage, where silcon prototypes of the chip undergo extensive testing. Throughout this process, the effective validation of the interconnect subsystem is hindered by the lack of observability into the interconnect's internal operations during a test's execution. In this context, it becomes necessary to pinpoint the critical set of debug data that must be monitored in order to capture a comprehensive functional overview of the interconnect's execution. Once this data is identified, the next challenge is in developing ingenious methodologies to effectively extract this information and attain a high degree of internal observability. With the network's overall execution being a function of distributed and parallel operations occurring in the interconnect's subcomponents, the developed methodologies must be capable of collecting debug data at a sufficient rate and level of detail, while also adhering to the various constraints imposed by the emulation and post-silicon verification environments. Moreover, the verification process is further enhanced by equipping the interconnect and the verification frameworks with techniques to detect the occurrence of communication errors and help localize these errors to a more precise set of components within the interconnect and to a subset of in-flight data messages affected by the errors. In this dissertation, we explore and address the challenges outlined above, developing a synergistic set of techniques that boost observability and facilitate the detection, as well as debugging of functional errors in the interconnect's design and implementation. Additionally, recognizing the inherent incompleteness of design-time verification efforts and the potentially detrimental effects of designs bugs escaping into the interconnect of a released product, we also develop runtime verification techniques targeting these large NoC interconnects. The goal of these runtime solutions is to enhance the functional correctness guarantees of the interconnect throughout its lifetime. Therefore, we explore and develop light-weight mechanisms to flag communication errors on-the-fly and to successfully recover from them, while introducing minimal perturbation and overheads to the end-user, particularly in the absence of errors.

Table of Contents

Dedication ii

Acknowledgments iii

Preface iv

List of Figures ix

List of Tables xi

Abstract xii

Chapter 1 Introduction 1
1.1 Interconnects in Modern CMP and SoC Systems 5
1.2 The Functional Verification of the Interconnect 7
1.2.1 Pre-Silicon Verification 7
1.2.2 Post-Silicon Validation 9
1.2.3 Runtime Verification 10
1.3 Overview of The Dissertation 11
1.4 Organization of the Dissertation 14

Chapter 2 Design-time Verification of the Interconnect subsystem 16
2.1 Design-time Verification Platforms 16
2.1.1 Emulation Platforms 17
2.1.2 Silicon Prototypes 18
2.2 Prior Solutions Addressing Emulation and Post-silicon Validation 19
2.3 Interconnect-targeted Error Detection Capabilities 20
2.3.1 Functional Errors in the Interconnect 20
2.3.2 Network-level End-to-End Error Detection 21
2.3.3 Error Detection through Localized Checkers 22
2.4 Summary 29

Chapter 3 Enhancing Observability of Emulation and Post-Silicon Verification 30
3.1 Identifying Critical Debug Information 30

3.2 Monitoring and Logging Debug Data 31
3.2.1 Packet Monitoring 32
3.2.2 Router Monitoring 36

Chapter 4 Packet Monitoring 39
4.1 DiAMOND Overview 40
4.1.1 Debug Data Collection 42
4.1.2 Error Detection 45
4.1.3 Debug Data Analysis 46
4.2 Experimental Evaluation 47
4.2.1 Implementation and Area Overhead 47
4.2.2 Evaluation of the Local Analysis Phase: Path Reconstruction 48
4.2.3 Evaluation of the Global Analysis Phase: Case Study 50
4.3 Summary 53

Chapter 5 Router Monitoring 54
5.1 Logging 55
5.1.1 Logging in the Routers 55
5.1.2 Log Storage in Local Caches of a CMP Design 57
5.2 Error Detection and Debugging 58
5.2.1 Local Checks 59
5.2.2 Global Checks 61
5.3 Experimental Evaluation 62
5.3.1 Error Detection 63
5.3.2 Error Diagnosis 64
5.3.3 Performance Evaluation 65
5.3.4 Area Overhead 67
5.4 Summary 68

Chapter 6 Runtime Correctness for NoC Interconnects 69
6.1 Background and Related Work 69
6.2 SafeNoC: Correctness without Packet Replication 71
6.2.1 Checker Network 72
6.2.2 Error Detection 73
6.2.3 Error Recovery 74
6.2.4 SafeNoC Implementation 76
6.2.5 Evaluation of SafeNoC's Performance and Error Recovery 81
6.2.6 Limitations of SafeNoC's Runtime Correctness 84
6.3 REPAIR: Correctness through Adaptive-Region Protection 84
6.3.1 Identifying Error-Prone Network Regions 86
6.3.2 Achieving Communication Correctness 89
6.3.3 Evaluation of REPAIR 92
6.4 Summary 96

Chapter 7 Conclusion 98

7.1 Summary of Contributions 98
7.2 Future Directions 100

Bibliography 102

List of Figures

Figure

1.1 The transition to communication-centric systems. . . . 2
1.2 General architecture of a virtual-channel router 5
1.3 The functional verification of the interconnect during the pre-silicon stage. . 8
1.4 Thesis overview 14

3.1 Packet Monitoring 33
3.2 Monitored packets observed at routers. 35
3.3 Router monitoring 37

4.1 Overview of DiAMOND 40
4.2 DiAMOND's validation flow 41
4.3 DiAMOND's debug data collection at NoC routers 43
4.4 DiAMOND's modes of operation 44
4.5 Example of reconstruction of packet interactions 50
4.6 Average packet latency at internal routers 51
4.7 Average packet latency for a sample benchmark. 52
4.8 Packet latency at router 49 for dedup benchmark. 52

5.1 Execution flow of the NoC debug platform. 55
5.2 Logging 56
5.3 Information logged in each snapshot. 57
5.4 Local check algorithm 61
5.5 Error detection rate 62
5.6 Error Diagnosis 63
5.7 Performance with varying snapshot intervals and local check sampling rates 66

6.1 High-level overview of SafeNoC. 72
6.2 SafeNoC recovery process 74
6.3 Packet reconstruction algorithm 75
6.4 SafeNoC's hardware implementation. 77
6.5 SafeNoC recovery time for each benchmark 82
6.6 SafeNoC recovery time by bug 82
6.7 High-level overview of REPAIR 85

6.8 Identifying congested regions and peripheral routers. 87
6.9 Applying acknowledgment-retransmission at a peripheral router 90
6.10 Traffic workloads considered for evaluating REPAIR 92
6.11 Execution time of REPAIR compared to source-based retransmission 95
6.12 Buffering requirements of REPAIR vs. source-based retransmission 95

List of Tables

Table

4.1 Log_buffer. 48
4.2 Additions to header flits. 48
4.3 Average path reconstruction 49

6.1 SafeNoC area overhead 78
6.2 Functional bugs injected in SafeNoC 81
6.3 Performance overhead in the absence of bugs 83
6.4 Description of modeled bugs 93
6.5 REPAIR's bug recovery 94

Abstract

Today's computing devices consist of complex systems-on-chip (SoCs) consisting of a wide range of on-chip components, all integrated on a single chip and supporting the efficient and high performance execution of a myriad of applications. SoCs are now an integral part of many day-to-day products, ranging from mobile computing platforms to many other types of embedded devices. Additionally, chip-multiprocessor designs consititute another important segment of the computer industry, as they integrate a large number of processing cores and deliver the computational power required to run high performance workloads that are the basis of many scientific and big data applications. This high degree of integration in both CMPs and SoCs has essentially transitioned these systems from being computation-centric to communication-centric, making the interconnect subsystem a crucial element in such designs. The sheer size of the CMP and SoC designs themselves and the intricacy of the communication patterns they exhibit have propelled the development of network-on-chip (NoC) interconnect designs. NoCs provide a flexible, distributed, and highly parallel infrastructure that can meet the communication demands of these systems. However, faced with the inteconnect's growing size and complexity, several challenges hinder the effective validation of the communication infrastructure. During the development phases of a design, the functional verification process has witnessed a growing reliance on the use of emulation platforms and on post-silicon validation, in an effort to expedite verification and enhance correctness guarantees. Nevertheless, detecting and debugging errors on these platforms is a difficult endeavor due to the limited observability, and in turn the low verification capabilities they provide. Additionally, with the inherent incompleteness of design-time validation efforts, the potential of design bugs escaping into the interconnect of a released product is also a palpable concern. As the central medium of communication in CMP and SoC designs, functional bugs manifesting in the interconnect's execution can threaten the viability of the entire system.

The work presented in this dissertation provides solutions to enable the development of functionally correct interconnect designs. We first address the various challenges encountered during design-time verification efforts, by providing two complementary mech-

anisms that allow emulation and post-silicon verification frameworks to capture a detailed overview of the functional behavior of the interconnect. Our first solution logs debug information pertaining to packet traversals through the network and collects them by re-purposing the contents of in-flight packets to extract this data from the interconnect's execution. This approach enables the validation of the interconnect using synthetic traffic workloads, while attaining over 80% observability of the paths followed by in-flight packets and capturing valuable information to facilitate the debugging of a wide spectrum of communication errors. We also develop an alternative mechanism to boost observability that relies on taking periodic snapshots of execution. These snapshots are then utilized to reconstruct packet traversals and the network's overall operations. Using such an approach, the capabilities of the verification framework are extended to run both synthetic traffic and real-application workloads, while observing over 50% of the network's traffic, and reconstructing at least half of each of their routes through the network. Moreover, this dissertation develops light-weight error detection and recovery solutions to address the threat of design bugs escaping into the interconnect's runtime operation. Our runtime techniques can overcome communication errors, without the need to store replicates of all in-flight packets, thereby achieving communication correctness at minimal area and performance costs.

Chapter 1
Introduction

The design of computer systems throughout the past several decades has been driven by the continuous scaling of silicon technology, which permitted the integration of an increasing number of transistors on a single chip to provide greater performance and functionality. This trend was first predicted by what is known as Moore's law, which states that the number of transistors in integrated circuits doubles approximately every two years[60]. Initially, the focus was placed on designing high-performing single core processors. However, as these single-core architectures hit a wall of diminishing performance gains, there has been a shift towards multi-core processor designs. In today's computing landscape, multicore designs, or chip multiprocessors (CMPs), have become ubiquitous, ranging from processors with a handful of cores to high performance computing (HPC) platforms with hundreds of cores. The shift towards multicore designs also forked out into two other design paradigms: systems-on-chip (SoCs) and heterogeneous CMPs, which deviate from the use of a homogenous set of cores to incorporate non-identical cores, as well as other types of components such as GPUs, DSPs, *etc*.

Communication-centric designs. One consequence of the advancements of homogeneous and heterogeneous CMPs and SoCs is that they brings into focus the communication infrastructure as a critical component in the chip, as illustrated in Figure 1.1. The overall performance and functionality of the system is highly dependent on the interconnect subsystem that connects the on-chip components and manages the communication between them. The large number of highly parallel cores in CMPs requires an interconnect design that is capable of delivering data at a high enough rate for the cores to be efficiently utilized. Similarly, the integrated components in an SoC require an interconnect that can handle the variety of data transfer patterns among these different components. These requirements have effectively led to a shift in the design's approach: from computation-centric systems to communication-centric ones, and have made high performance interconnects a necessity to capitalize on the computational power that can be delivered by such systems. Network-on-chip interconnects have emerged as the scalable interconnect model that can meet the

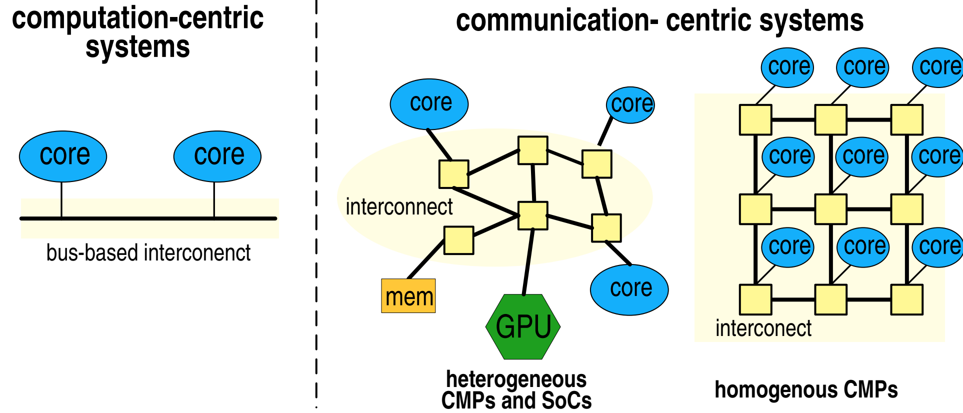

Figure 1.1 The transition to communication-centric systems. With the development of large chip-multiprocessors (CMPs) and systems-on-chip (SoCs), many design approaches have transitioned from computation-centric systems to communication-centric ones, relying on more complex interconnect infrastructures.

growing communication demands of CMP and SoC designs. Consisting of a set of routers that are connected via links and that discerningly forward messages between on-chip components, the NoC constitutes a flexible, distributed, and high-bandwidth fabric. The NoC interconnect model also offers opportunities for innovations in its design and implementation, which, in turn, makes it pertinent to the enabling of current and future computer systems. Going forward, as higher performance, higher efficiency, and more functionality are expected to be reaped from CMP and SoC designs, networks-on-chip constitute a dominant choice for the interconnects of these large and highly integrated systems.

Functional verification of the interconnect subsystem. Alongside the changes in design trends, verification has been playing an integral role throughout the design process. In particular, significant effort is spent on functional verification, which is the process of ensuring that the design is free of any functional bugs and is operating according to its specifications. Functional bugs occur if erroneous behaviour is introduced at any stage of the design's implementation or if the original specification is erroneous or incomplete. Despite the many advancements in the verification process over the years, it remains riddled with numerous challenges, especially in the face of growing design complexity and size. This is further exacerbated by the shorter design cycles of released products, many of which are in the range of 6 months to a year. As a result, over the years, the fraction of time spent on functional verification, as well as the resources and the manpower dedicated to this process, have multiplied. Nevertheless, incomplete functional verification remains a persistent threat, as the number of bugs discovered in products after their release has been on the rise. For example, data extracted from Intel processors' errata documents [34, 3, 33], which list bugs discovered after each processor's release, show a clear increase in design bugs as pro-

cessors transition from single core to more complex multi-core designs. The concern of design bugs manifesting in released products has in turn fueled the emergence of runtime verification solutions to complement the design-time verification efforts. A runtime verification solution is intended to accompany a hardware product throughout its lifetime and act as a last line-of-defense that catches and overcomes any latent design bug.

In the context of todays communication-centric CMP and SoC systems, functional verification can not be limited to only verifying the stand-alone on-chip components. In fact, functional correctness can not be ensured unless system integration and on-chip communication is also exhaustively verified. The interconnect subsystem is the infrastructure that integrates the on-chip components and is the physical layer that implements the on-chip communication protocols. Thus, ensuring the functional correctness of the interconnect is a critical and indispensable step in verifying the design. Moreover, as the sole medium of communication in the system, the interconnect constitutes a single point of failure, and its failure halts the functioning of the whole system. Therefore, design bugs escaping into the interconnect of a released product have the potential to threaten the viability of the whole system. From the perspective of the end-user, these errors can lead to the failure of applications, the disruption of system operations, performance losses, as well as the compromise of user's data. On the other hand, design houses can suffer from extensive financial and market value losses, especially when these bugs lead to product recalls and patch releases.

Network-on-chip interconnects pose unique challenges to the verification process. Over the years, the complexity of network-on-chip designs has skyrocketed, as more features and optimizations are incorporated into its design to extract performance, energy-efficiency and reliability. In addition to increasing complexity, the size of these subsystems is continuously growing as we integrate more and more components on-chip. Lastly, being a distributed communication fabric, the NoC's overall operations are a function of local operations within its sub-components, as well interactions between them, and thus its full functional correctness can not be guaranteed unless it is designed and verified as a whole. Functional verification efforts during the design phases of the NoC interconnect consist of subjecting the network to a wide variety of testing executions that range from randomly generated test cases to real applications. During validation, the interconnects operations are monitored to identify functional errors. Once errors are detected, debug information that was extracted from execution is used to diagnose, localize and debug the detected error. However, the size, design complexity and distributed nature of NoC designs make it difficult to carry out these steps in an efficient and systematic manner. First, high design complexity translates to a large number of design signals, which is prohibitive to effectively observe during validation. Moreover, the distributed nature of NoC operations further com-

plicates the issue. Error detection and debugging requires mechanisms that monitor the behaviour and extract debug information from components and signals that are distributed throughout the NoC design. Lastly, after errors are detected and debug data is successfully collected, solutions are needed to efficiently and quickly make sense of the data to diagnose and debug the errors. Furthermore, challenges are not limited to verification efforts during the design stages of NoC interconnects, but they also extend to runtime verification solutions intended to protect the released product from latent bugs that have escaped into the NoC design and implementation. In fact, runtime conditions introduce even more stringent constraints, as any runtime verification solution should incur minimal area and performance overheads. In this context, the first challenge is in developing low-overhead mechanisms that can monitor the distributed operations of the NoC and identify erroneous behaviour. Then, after an error is flagged, the goal is to trigger a recovery mechanism that can correct or bypass the bug. In the face of these challenges, during both the design-time and runtime verification of NoC interconnects, the concern of latent functional design errors is an impending reality with potentially detrimental effects. To allow NoC interconnects to continue to be the interconnect model of choice for future systems, we need innovations that can address the challenges that are hindering the NoCs functional verification.

Thesis overview. This thesis enables the development of correct network-on-chip interconnect designs, by providing a synergistic set of solutions to effectively verify the interconnect's functional correctness throughout different phases of its design and deployment. We first address the challenges encountered during design-time verification. In the presence of countless NoC design elements and data that can potentially be monitored and extracted during validation, we identify the set of critical debug information that encapsulates the functional behaviour of the network. Based on that, we then introduce low-overhead methodologies to extract this distributed debug data during the network's execution. We first develop a methodology that observes the interconnect's execution through monitoring the progress of traffic and re-purposing the traffic's data contents to store a detailed log of execution events. As such, our traffic monitoring approach targets the validation phases during which the interconnect is tested using randomly generated traffic patterns. Second, we develop a solution that boosts observability through the periodic sampling of execution from the perspective of the network routers. In this latter approach, network traffic is not perturbed permitting the validation of the NoC by running real-world applications as well as synthetic traffic workloads. Additionally, in both mechanisms, we utilize the collected debug information to reconstruct an overview of the network's operations throughout execution, providing valuable debug capabilities that would not be attainable otherwise. We also ensure interconnect correctness beyond the design phase

by introducing verification solutions to protect the interconnect from latent functional bugs throughout its runtime operation. We equip the network with light-weight checking mechanisms to identify erroneous behaviour along with recovery mechanisms to successfully bypass the errors, all while incurring minimal area and performance costs. With the communication infrastructure being the backbone of current and future CMPs and SoCs, the work presented in this thesis sustains the continued innovations in interconnect designs, by enhancing the guarantees of the interconnect's functional correctness.

1.1 Interconnects in Modern CMP and SoC Systems

The high degree of integration of on-chip components in modern designs has triggered a shift in the interconnect model that connects these components and permits communication between them. Traditionally, with a few on-chip elements, a shared bus was the interconnect fabric of choice that provided a simple and effective communication interface. However, the high-level of computational parallelism brought about by the integration of a large number of homogenous and heterogeneous on-chip elements necessitated an equally parallel and high performing interconnect model. Therefore, network-on-chip (NoC) interconnects were developed to meet the increasing bandwidth, scalability and performance demands.

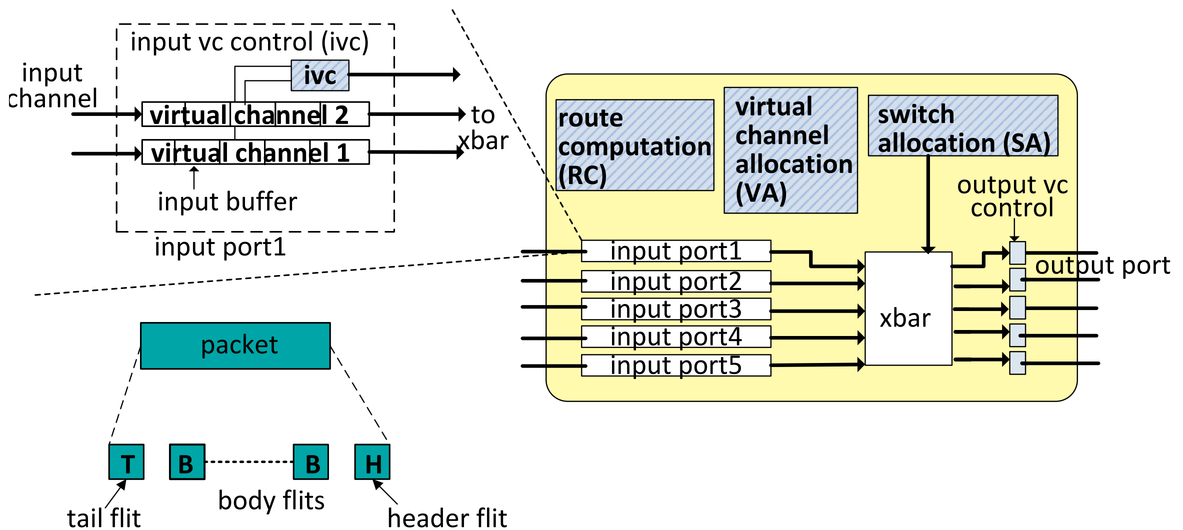

Figure 1.2 General architecture of a virtual-channel router. The figure shows a router with two virtual channels per input port. A received packet is first stored in one of the input buffers. The route computation (RC) stage determines the output port to which it will be sent. The virtual channel allocation (VA) unit assigns an output virtual channel and the switch allocator (SA) arbitrates for the use of the crossbar.

A NoC consists of a set of routers connected via links. These routers can be organized in different topologies, ranging from simple ring networks to 2D meshes and other more intricate organizations. Each on-chip component connects to a router through a dedicated network interface unit, which buffers outgoing messages, segments them into packets, and injects them into the network. After packets are injected into the network, they are transmitted to their destination nodes through a series of routers, along a path that is determined by the network's routing protocol. The NoC's flow-control mechanism governs how the network's resources are allocated to in-flight packets. One commonly-used mechanism is known as wormhole flow-control. In wormhole flow-control, packets are partitioned into smaller equal-sized blocks called flits, as illustrated in Figure 1.2. The first flit, the header flit, marks the beginning of the packet and contains the address of the destination node. The header is then followed by several body flits that contain the packet's data contents, terminating with a tail flit that marks the end of the packet. When a packet first arrives at a router, it is queued in one of the router's input buffers. In a virtual channel router, there are multiple virtual channels per port, and the flits of an incoming packet are all stored in the same input buffer corresponding to one of the virtual channels. Figure 1.2 shows a general router architecture, consisting of 5 input ports with 2 virtual channels per port.

Within a wormhole virtual channel router, each packets passes through 3 main pipelined stages: route computation, virtual channel allocation, and switch allocation.

Route computation (RC): Route computation starts when the header flit of a packet reaches the head of the input buffer in which it resides. Based on the packet's destination and the network's routing protocol, the route computation unit determines the output port (and therefore the downstream router) to which the packet will be sent.

Virtual channel allocation (VA): Packets that have completed route computation request an available virtual channel in the input port of their chosen downstream router. The availability of virtual channels in neighboring routers is typically maintained using a credit-based mechanism, where each router keeps track of the number of flits sent to each of its neighboring routers and the free buffer space in each neighboring router's virtual channels. Once an output virtual channel has been granted to a packet, it is reserved for that packet, until all of the packet's flits have been transferred to the downstream router. Therefore, only a packet's header flit is required to go through route computation and virtual channel allocation, and once the output port and output virtual channels have been determined, the remaining flits proceed directly to the switch allocation stage.

Switch allocation (SA): Flits at the heads of the router's input buffers individually arbitrate for the use of the crossbar. Once the requested input-output crossbar connection is granted, the corresponding flit traverses the crossbar and is forwarded to the downstream router.

1.2 The Functional Verification of the Interconnect

Throughout the development of any hardware design, functional verification constitutes a critical task that is tightly coupled with the various phases of design. The design development cycle typically begins with a set of specifications, from which a functional model of the design is written in a high-level language. After these specifications are ready, designers create a Register-Transfer Level (RTL) model using a hardware description language (HDL), such as Verilog or VHDL. The RTL model is then synthesized into a gate-level netlist, which in turn is passed on to the physical design process that maps the netlist to a circuit layout by performing floor planning, partitioning, and placement and routing. The circuit is then fabricated into silicon and the product is released to the market. In each step of this process, extensive efforts are spent on ensuring that the design is operating correctly, relative to the original specifications, and that it is free of any functional design bugs. Functional bugs occur when erroneous behavior is introduced into the design or its implementation, at any stage of the design process, or if the original specification is itself incomplete. In particular, functional verification efforts can be categorized into three main stages: pre-silicon verification, post-silicon verification, and runtime verification.

1.2.1 Pre-Silicon Verification

The goal of pre-silicon verification is to ensure that the design's functional model, the RTL model, and the netlist are all operating correctly, both in terms of functionality and performance, according to the design's specifications. Pre-silicon verification techniques consist of the use of formal verification tools, software-based simulations, as well as emulation. **Formal verification tools** are used to prove that the design meets certain correctness properties. As a result, formal verification is a powerful and complete verification methodology that can guarantee correctness over the entire design state space and over all execution scenarios. However, formal verification engines suffer from the problem of state space explosion and are unable to scale to large designs. Moreover, writing properties to be checked by a formal tool is in itself a difficult and error-prone process. On the other hand, in **software-based simulation** methods, the design is simulated using a software-level verification environment. Software-based simulations provide complete control and visibility over the design's operations, resulting in relatively easy error detection and debugging. Nevertheless, simulations are slow, in the order of 1-10Hz, which is orders of magnitudes slower than any hardware design being manufactured today. Therefore, software-based methods are incapable of exhaustively verifying large and complex designs for the entire

range of their executions. Faced with the limitations of these formal and simulation-based methods and with the growing complexity and size of current hardware designs, pre-silicon verification approaches based on **emulation platforms** have emerged. In emulation, the design is mapped onto a configurable hardware platform, such as an FPGA. The advantages of such platforms is the higher execution speeds, 10KHz - 100MHz, allowing a more thorough exploration of the design's state space. However, with tests running on hardware, emulation platforms offer limited observability of a design's signals and operations. This approach introduces a new challenge, where errors detected in this phase are now difficult to diagnose and debug, without a methodology that can either automatically localize the error or provide enough debug information for verification engineers to analyze.

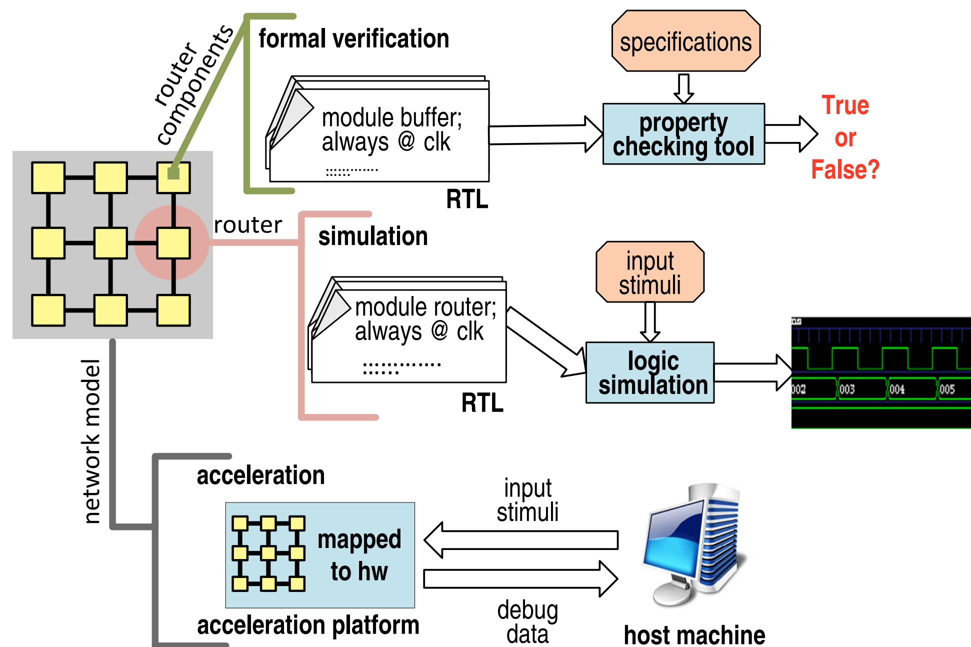

Figure 1.3 The functional verification of the interconnect during the pre-silicon stage. The pre-silicon verification stage consists of formal verification, software-based simulations, and verification on hardware emulation platforms.

During the pre-silicon verification of NoC-based interconnects, formal methods and simulation-based verification efforts tend to focus mostly on the router's sub-components and, in some cases, the individual router design itself, if it is simple enough to be practically manageable by the simulation and formal verification tools. However, these initial efforts are not enough, as the functional correctness of the full interconnect cannot be presupposed by verifying the operations of just one router or a small subset of routers connected together, and eventually, the entire network must be modeled and exhaustively validated as a stand-alone fabric and as part of the entire system. This is due to the fact that the overall

execution of the network is a result of network-level interactions that occur as responses to the traffic patterns traversing it. These interactions could span a small region of the network or the entire network in some cases. Moreover, in many NoC designs, local control decisions made within individual routers are often based not only on local router conditions but also on network-level information. Lastly, the correctness of some aspects of the NoC design, such as the routing and communication protocols and their implementations can only be observed by considering the network as whole, with some types of errors only manifesting at the network-level. Thus, for the complete functional verification of the interconnect, modeling the entire NoC design is inevitable and necessary. As a consequence, emulation platforms are commonly used in the pre-silicon stage, as they provide faster and more efficient capabilities to validate the interconnect under various traffic scenarios. In a typical emulation-based verification environment, the interconnect being tested along with some verification features, such as checkers and debug monitors, are mapped to the hardware emulation platform. This platform interfaces with a testbench environment that is responsible for driving the simulation. In this framework, the interconnect can be tested under different traffic scenarios that range from statistical traffic patterns and directed-random tests, which can stress the network and exercise a diverse set of its operations, as well as application traffic, which simulate the interconnect under more realistic runtime conditions.

1.2.2 Post-Silicon Validation

Once the first few silicon prototypes of the design are manufactured, post-silicon validation commences. Tests run directly on the chip, and at chip speed, providing a great performance advantage that allows validating the design under various operating scenarios. During this stage, it is possible to run entire applications and to boot up operating systems, in addition to running randomly generated test cases. Test outcomes are typically checked using hardware checkers or assertions or by comparing against a golden model for the execution. Although functional verification remains a leading goal at this stage, it is not the only concern, as bugs discovered in the silicon prototype can also be due to electrical errors and manufacturing defects.

During the post-silicon validation stage, checking the functional correctness of the interconnect can no longer be done independently from the other on-chip components, as the interconnect is fully integrated within the whole system. As a result, it is challenging to identify the sources of test failures and to localize functional bugs in the interconnect design. Moreover, the post-silicon validation platform offers very limited observability of the internal behaviour of the design, as only a small number of inputs and outputs can be

observed. Traditional solutions to this observability problem rely on the use of scan chains and debug buffers to monitor a small number of signals or state elements. Test failures are then debugged by extracting the debug information and attempting to diagnose the source of the error. However, in the context of validating an NoC-based interconnect, the observability gained from these techniques is still minimal, not to mention that reading the data requires regularly stopping execution to scan it out. Network-on-chip designs are large subsystems that often include complex features and hardware modules, which provide an extensive amount of state elements and control signals that can potentially be monitored to gain observability. In practice, it would be prohibitive to monitor all this data, and it is necessary to identify the critical set of states and signals that would be most valuable in capturing the functional operations of the network and that would also be practically feasible to monitor without slowing down or significantly perturbing execution. Another aspect of network-on-chip designs that exacerbates the observability challenge is that it is a distributed and parallel communication infrastructure. Thus, the full functional behavior of the interconnect cannot be observed by monitoring operations at a centralized location or by observing a smaller portion of the network. Moreover, the high performance of the post-silicon validation environment permits running long test cases with execution times exceeding millions of cycles. As such, it would be prohibitive to monitor or log the cycle-by cycle execution of the network, and monitored data must be efficiently collected and stored. Lastly, with only a subset of states and signals being monitored and without a complete and contiguous view of the network's execution, it becomes a challenge to make sense of the collected debug data and to effectively utilize it for detecting and debugging design bugs.

1.2.3 Runtime Verification

Although the majority of functional bugs are identified and fixed during design-time verification, the possibility of a small number of undiscovered bugs escaping into the released product is a persistent concern. The lack of scalability of formal tools and software-based pre-silicon verification methodologies, especially in the face of growing design complexity, renders these verification techniques inherently incomplete. Moreover, verification on acceleration and post-silicon platforms is limited by low observability and low error detection and debugging capabilities. As such, these techniques are generally used to ensure the functional correctness of the design's most common operations. However, functional errors remain latent in corner case operations, as well as any execution scenario that was not exhaustively verified. Over the years, several semiconductor companies and research

works have explored approaches that can guarantee correct operation at runtime, despite incomplete design-time verification. These solutions are intended to be active throughout the design's lifetime to allow the system to overcome any bug that has escaped design-time verification efforts. This advantage can be critical for hardware manufacturers, as bugs discoverd after a product's release can be sidestepped without causing significant losses.

Implementing an effective runtime verification solution for NoC interconnects poses its unique set of challenges. In contrast to design-time verification, where the goals are to detect, localize and debug functional bugs, runtime verification techniques are primarily concerned with detecting the occurrence of an error and bypassing it or recovering from it. In this context, it may be enough for a runtime detection mechanism to identify the occurrence of a communication error, without needing to localize the exact source of the bug. However, despite having more flexibility in terms of bug detection, runtime verification approaches have stringent constraints with respect to their area and performance overheads. Thus, it is necessary for such approaches to detect communication errors on-the-fly, using only light-weight mechanisms that incur minimal perturbation to the original execution. In addition, in many network-on-chip interconnect designs, once messages are injected into the network and while they are in-flight, no duplicate copies of them are stored elsewhere. Therefore, communication errors affecting in-flight traffic cannot always be easily overcome, since the communicated data is not always recoverable.

1.3 Overview of The Dissertation

In this thesis, we address the various challenges encountered throughout the verification lifecycle of network-on-chip interconnects, to enable the development and deployment of functionally correct communication infrastructures.

Starting out in the pre-silicon verifcation stage, we focus primarily on emulation-based verification, as we recognize that not only does it constitute a growing fraction of the pre-silicon verification efforts, but it is also indespensible when simulating the entire NoC design. As such, the challenges encountered during the pre-silicon stage become similar to those in post-silicon validation, with the lack of observability being a major obstacle to efficient functional error detection and debugging. To this end, we developed solutions to enhance the observability of the network's internal operations during a test execution.

The first step in achieving greater observability is identifying the critical set of states and signals that should be observed to construct a detailed-enough overview of the functional behaviour of the network during a particular execution. Therefore, we determine that

by logging specific state elements from each router along with debug data pertaining to the in-flight packets, as well as timestamp information of packet arrivals and departures, we can recreate a spatial and temporal overview of the execution. However, with a large number of in-flight packets traversing routers and many simultaneous execution events occuring within routers and the network, it is challenging to efficiently extract this debug data at the rate at which is generated. Therefore, we develop two approaches to monitor and collect the target debug information. In the first approach, in-flight packets are monitored and the debug data logged pertains to the progress of these packets through the network.Whereas, in the second approach, the network's routers are monitored and the debug data is logged captures the execution events observed at the routers. These approaches are complimentary and facilitate the verification of the interconnect under different test scenarios.

In our traffic monitoring solution, debug data is collected at every hop along a packet's path and is logged by systematically replacing parts of the packet's contents. At the end of simulation, each packet holds a functional and timing overview of its own path through the network, and the aggregation of the debug data from multiple packets recreates a subset of the interactions and operations that occurred in the network. Using this technique, the interconnect can be efficiently validated using directed-random traffic test-cases. In such tests, the traffic injected into the network is is usually in the form of statistical or random traffic patterns,where the data contents of in-flight packets are irrelevant to the network execution. Therefore, packet contents can be repurposed to store data that is being monitored and collected across the network, ultimately providing better observability and debugging support.

However, it may not always be possible to alter in-flight messages, such as in the cases when real applications are running on the verification platform. In this context, the traffic injected by each node is a response to messages it has received from other nodes, such that different input data generate different traffic patterns and hence a different network execution. Therefore, we develop a second approach that instead relies on monitoring the local operations of network routers. Debug data is then periodically logged from within each router and stored in the local cache or memory corresponding to that node or in designated trace buffers. As such, the debug data collected throughout a test's execution is effectively a set of sample points of execution that can be aggregated to recreate a functional behavioral overview of the network.

Beyond the design-time verification stages, we also develop runtime solution that equip the network with mechanisms to detect and recover from erroneous functional behavior that might be encountered throughout its lifetime as part of the final product. Among the simplest techniques to ensure correct communication is to hold on to a copy of every data

packet before it is injected into the network. This copy is maintained as long as the packet is in-flight and any error in transfer can be resolved by simply transmitting another copy of the packet. In this thesis, we show that this approach can be prohibitively costly in terms of performance as well as area. Moreover, this technique is often overcompensating, as design bugs manifesting at runtime are relatively rare, especially since the interconnect undergoes extensive verification before the product's release. As an alternative to such a solution, we propose an approach that avoids storing any redundant copies of in-flight packets. Instead, it relies on augmenting the original network with a simple and light-weight checker network that is formally verified and guaranteed to be functionally correct. This checker network is then used to send small look-ahead packet signatures that are used to identify communication errors. Such errors are then overcome by collecting all data that is in-flight at the time of the error occurrence, and then using the look-ahead signatures to reconstruct the original packets. On the other hand, we propose a second solution, where we relax the goal of not storing any redundant packet copies. Thus, we rely on identifying the subset of packets that are prone to errors and only creating copies of those packets. We dynamically identify network regions where the execution is more likely to expose an escaped design bug, and we create copies of only the packets that are traversing these target regions. Packet copies are then utilized to overcome the detected bug.

Figure 1.4 presents an overview of this thesis. With the goal of enabling the development and release of functionally correct interconnect designs, we group verification efforts into two phases: development-time and runtime. During the development phase, we focus primarily on emulation-based verification and post-silicon validation, as they constitute a major and growing segment of the functional verification process of interconnect designs. As shown in the top half of the figure, we identify three main steps in achieving better verification capabilities on emulation and post-silicon platforms. First, we determine the critical set of data that needs to be monitored from within the interconnect's sub-components. Second, we develop methodologies to efficiently extract and collect this data, generating debug logs of the execution. Third, we utilize the collected logs to provide valuable debug information and insights into the operations of the interconnect, which in turn enhances error detection and debugging. Beyond the development phase, we aim to provide functional correctness guarantees during the runtime operation of the interconnect. With the potential of design bugs escaping into the interconnect subsystem of a released product, we require error detection and recovery mechanisms to monitor execution, flag incorrect communication, and recover from the detected errors by ensuring that messages are delivered correctly to their intended destinations.

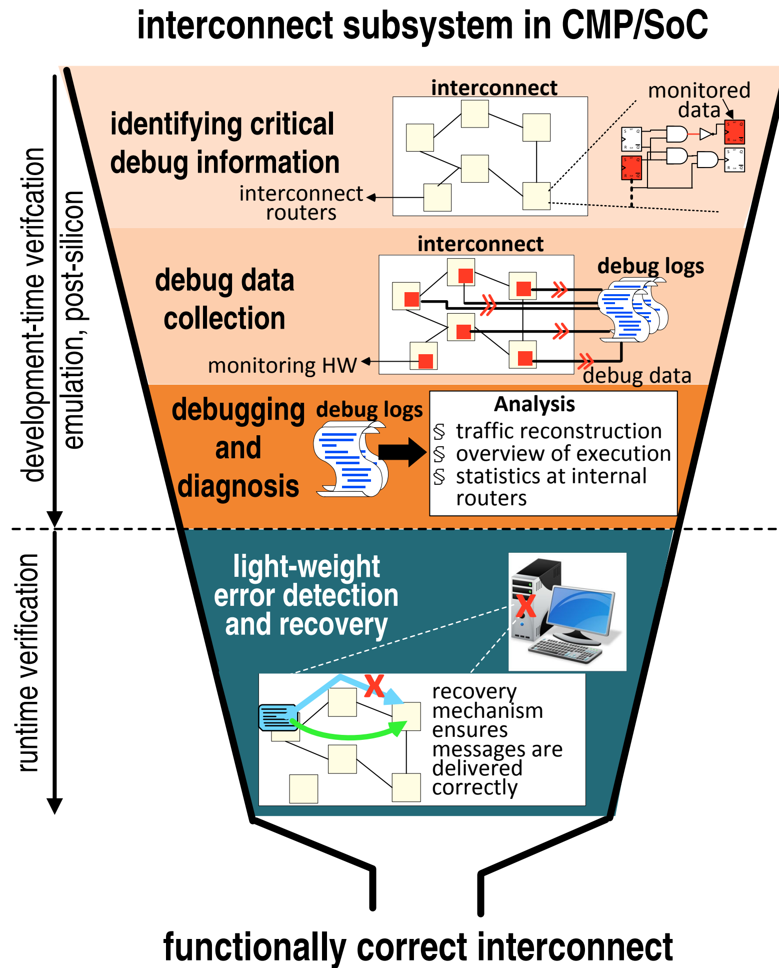

Figure 1.4 Thesis overview.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 overviews functional verification on emulation and post-silicon validation platforms, including common verification practices and recent research works that aim to improve this process. In addition, this chapter discusses common error detection techniques that can aid in localizing a test's failure to the occurrence of errors in the interconnect subsystem. Chapter 3 discusses approaches to enhancing observability during emulation-based verification and post-silicon validation to achieve better error detection and debugging capabilities. In this chapter, we first identify the set of critical state elements that needs to be monitored during execution to attain the needed observability. Then, we explore packet monitoring and router monitoring as two complementary approaches to collect and extract the debug data. In chapter 4, we present a complete packet monitoring solution, detailing the debug data that must be logged for each packet and presenting a novel solution that repurposes the contents of

packets to store the collected debug data. To futher enhance error detection and debugging capabilities, this solution also integrates localized hardware checkers for flagging communication errors and triggering the debug process. On the other hand, Chapter 5 describes and implements an alternative router monitoring solution, where network observability is enhanced through the periodic sampling of execution at each network router. This chapter describes the data that must be monitored at each router, and the mechanism by which the data is collected, stored and analyze. In Chapter 6, we introduce runtime verification and describe our two solutions that target the runtime correctness of the interconnect subsytem. These solutions explore runtime error detection and recovery mechanisms that trade-off the need to store duplicate backup copies of in-flight packets. Lastly, Chapter 7 presents the conclusions of this dissertation and future research directions.

Chapter 2

Design-time Verification of the Interconnect subsystem

The interconnect is a critical component in SoC and CMP designs, as it constitutes the physical fabric upon which the entire communication subsystem is implemented. Thus, extensive effort is spent on ensuring the functional correctness of the interconnect design and its implementation. However, with the adoption of network-on-chip interconnects and the growing size and complexity of these designs, the interconnect's verification has shifted to a heavy reliance on the use of emulation and post-silicon validation platforms. While these platforms provide powerful verification capabilities, they present their own set of verification challenges. In this chapter, we describe the characteristics of emulation and post-silicon frameworks and the unique trade-offs they introduce to the functional verification process. We also survey general approaches utilized in industry and proposed by the research community to address some of the verification challenges encountered with such platforms, including works that target specifically the verification of network-on-chip interconnects. Additionally, in this chapter, we examine error detection on these platforms, focusing on the different techniques that can be used to detect functional errors in the interconnect during its verification.

2.1 Design-time Verification Platforms

Simulation is the primary method by which today's hardware designs are functionally verified. Early-on in the design process, when the RTL model of the design is being developed, software-based simulations are heavily utilized. However, even with dedicating large arrays of servers to perform these simulations, software simulators applied to a full-chip design are still slow, achieving only 10-100 simulation cycles per second. As such, running full-chip simulations that are longer than a few hundreds of thousands of cycles is

practically infeasible in a software-only environment. Faced with the low speeds of software tools coupled with growing size and complexity of designs, semiconductor and CAD companies introduced emulation platforms into the pre-silicon verification process. By mapping the design under verification onto a hardware platform, the emulation verification environment runs simulations at speeds that are orders of magnitude faster than a software verifcation framework. A recent 2014 study on the functional verification practices in the semiconductor industry indicates that 35% of today's industry has already adopted hardware emulation as a way to extend the verification performance of large designs. In fact, the adoption of emulation is as high as 57% for designs consisting of more than 80 million gates [28]. Beyond the pre-silicon phase, once the RTL has been developed and sufficient verification coverage has been achieved, silicon prototypes of the chip are manufactured and subjected to extensive functional validation. Tests are run directly on the prototypes, with speeds close to that of the actual chip, in the hopes of uncovering any latent bugs before the product is released into the market.

2.1.1 Emulation Platforms

Emulation platforms typically consist of a number of FPGAs (field programmable gate arrays) that provide programmable logic blocks, onto which the design under verification (DUV) is synthesized. A host machine closely interfaces with the FPGA-based platform, driving the simulations running on it and capturing data from the execution. The exact distribution between the parts of the design and the verification features that must be mapped onto the emulation platform versus simulated on the host machine depends on many factors. Earlier in the development phase, when verification efforts are focused on a specific component or on a smaller portion of the design, it is possible to synthesize only that single component or that portion onto the FPGA platform. The rest of the system can then be functionally modeled on the host machine or possibly on the emulation platform itself, if enough hardware resources are available. In such a setup, the stand-alone interconnect design can be mapped to the emulation platform along with traffic generators and receptors to drive the execution of test cases. With bigger and more powerful emulators, the entire SoC or CMP design may be mapped onto the platform, emulating the full system's functionality. Whereas, non-synthesizable testbenches and verification modules run on the host. In additon to synthesizing the DUV, it is also common to include additional verification functionality, such as functional checkers and test generators, on the emulation platform itself. Embedded alongside the DUV, these verification modules interact more seamlessly with the DUV, reducing the need to communicate back and forth with the host machine. In

practice, the frequent transfer of large amounts of data between the host and the emulation platform can become a bottleneck that degrades the overall performance of this framework. Besides the engineering effort needed to map the design onto the emulation platform, this setup also presents major challenges to the debugging process due to the limited observability it offers into the DUV's internal execution. This problem is commonly addressed by instrumenting the design with special purpose logic to monitor a subset of the its signals and operations. The monitored data is then stored in trace buffers on the platform or off-loaded to an external trace memory, to be analyzed by debugging tools.

2.1.2 Silicon Prototypes

During the post-silicon stage, silicon prototypes of the chip are validated for correctness in actual application environments and over specific operating conditions. At this stage in the verification process, the interconnect is fully integrated as part of the whole system, and tests running on these prototypes range from random instruction sequences to end-user applications, including operating systems, scientific workloads, and typical user applications. One of the main challenges encountered during the validation of these prototypes is the restricted observability of the design's signals. A common method for gaining observability into the internal behavior of the system consists of augmenting the design with boundary scan registers [2] to serially shift output or internal state data values from the design. Moreover, it is typical to also include various design-for-debug features (DFD), such as embedded logic analyzers (ELAs) and trace buffers. ELAs primarily consist of programmable trigger units and data capturing units. The trigger units specify a set of events or execution conditions to trigger the monitoring of internal design signals, whose values are then logged by the data capturing units and stored in on-chip trace buffers. Once these buffers are full, their contents can be off-loaded for further debugging and analysis. It is also possible to leverage the on-chip memory and cache modules of the CMP and SoC to store debug data collected during execution [46, 27]. In this setup, the design's on-chip memory is logically divided into two portions, with one reserved for storing the instructions and data of the applications running on it and the other for storing debug data. With such an approach, longer traces of execution can be logged, without the need to implement large trace buffers. The reserved portions of the cache are also released after post-silicon validation is complete. Although DFD features are extremely useful for verification, they provide little benefit to the released product from the perspective of the end-user. Therefore, design houses can only invest a limited amount of the chip's hardware resources to implement DFD functionality.

2.2 Prior Solutions Addressing Emulation and Post-silicon Validation

There are a multitude of solutions proposed in academic research to improve emulation and post-silicon validation. These solutions range from techniques to increase observability, to the design of DFD features, the design and implementation of checkers, test generation, as well as error detection and debugging. For example, to enhance observability into a design's operations, many works focus on identifying which of the design's signals must be traced to allow the reconstruction of unobserved state elements of the design. Ko and Nicolici [37] were first to propose an automated trace signal selection method that tries to maximize the number of non-traced states restored from a given number of traced state elements. Other research followed proposing automated trace signal selection algorithms that rely on different heuristics to estimate the state restoration capabilities of a chosen set of signals [56, 15, 61, 43, 21]. On the other hand, other solutions have focused on designing efficient DFD infrastructures, including scan architectures [69], distributed embedded analyzers and trigger units [38, 36, 9], among many others.

In addition to these solutions, several research works have targeted the post-silicon validation of processor designs and memory systems. Whereas, a few others focus primarily on the post-silicon validation of NoC interconnects [68, 23, 22]. In the context of NoC designs, Vermeulen, *et al.* describe a general transaction-based NoC monitoring framework for systems-on-chip, where monitors are attached to master/slave interfaces or to routers. These monitors can filter traffic to observe transactions of interest as well as analyze network performance. [22] proposes adding configurable monitors to NoC routers to observe router signals and generate timestamped events. The events are then transferred through the NoC for analysis off-chip or at dedicated processing nodes. This work was later extended in [23] by replacing the event generator with a transaction monitor, which can be configured to monitor the raw data of the observed signals or to abstract the data to the connection or transaction level. However, the authors of [23, 22] report a high area overhead (17%-24%) for the monitors, which are not required for the network to operate after post-silicon validation.

Other debugging solutions for NoC-based multi-cores and systems-on-chip (SoCs) propose platforms that make use of the NoC-based interconnect to debug the cores and communication protocols in the systems, without addressing the debugging of functional errors in the interconnect design itself. In the work proposed by [66], debug probes are added between each core under debug (CUD) and its network-interface. The probes monitor communication transactions, generate signals to control the CUD, and read the CUD's

trace buffers. Control and debug data are transferred between the probes and an off-chip debug controller through the NoC. Similarly in [76, 75], probes are added to monitor incoming and outgoing packets of master IPs in an SoC, which are then used to analyze the initiation and completion of transactions.

A number of works have targeted emulation of networks-on-chip [74, 55, 30, 29, 39], where authors proposed different ways of implementing an emulation platform that allows the modeling and exploration of various NoC designs. These emulation platforms rely on traffic generators that can be configured to inject different traffic patterns. At the same time, traffic receptors are utilized to analyze received packets and other end-to-end correctness and performance metrics.

2.3 Interconnect-targeted Error Detection Capabilities

A critical step in the verification of the interconnect subsystem is determining when a functional design bug manifested in the interconnect during a test's execution. Generally, functional errors can be detected by checking the design's end-to-end execution at the system-level, where the DUV's outputs and the state of the system at the end of a test's execution are compared against the outcomes expected to have been produced by that test case. The expected results are obtained from running the test on a golden reference model, written in a high-level language, or are generated based on an understanding of the DUV's functionality and specifications. However, beyond determining the failure of a test case, it is important to localize the source of the failure to an error in a specific component or a set of components in the design. Without this capability, and especially when considering large and highly integrated systems, there may be countless potential causes for the test's failure and it is infeasible to consider each and every one. In this section, we discuss functional error detection, focusing primarily on the interconnect subsystem and the different approaches that can be adopted to identify the occurrence of communication errors and to localize these errors to a specific subset of components within the interconnect design.

2.3.1 Functional Errors in the Interconnect

Independently of the exact design bug and its location within the interconnect hardware, the functional correctness of communication can only be compromised in a limited number of ways. Previous works have identified four main conditions that are needed to ensure the functional correctness of a network: 1) No packets are dropped in-transfer. 2) Packet deliv-

ery time is bounded. 3) No packet data corruption occurs. 4) No new packets are generated within the network [54, 17]. We further elaborate on the second condition to distinguish between the following situations: when a packet is delivered to the wrong destination node, when it is not delivered at all, or if its delivery is delayed. This latter issue of delayed packet delivery may not always manifest as a functional correctness issue, and is often categorized as a performance bug. However, in NoC designs with strict quality-of-service (QoS) guarantees, if the network's performance does not meet the latency and threshold specifications, then the performance degradation is in fact a functional correctness concern. Even in NoC designs that do not support stringent QoS requirements, functional bugs affecting performance can have crippling consequences on the operation of the entire system, and it is equally important to catch these bugs during verification. In this chapter, and based on the above four conditions, we classify the communication errors that could arise into seven types. The first two errors fall under the data corruption condition, the third error combines packet drops and creations, and lastly, the remaining errors fall under the condition of timely delivery. Therefore, the process of detecting the occurence of a design bug in a NoC ultimately requires identifying the occurrence of these seven types of communication errors.

1. **Packet data corruption:** The data contents of a packet's flits are corrupted in transfer.
2. **Flit drop or replication:** A flit within a packet is dropped or replicated.
3. **Packet drop or replication:** A packet is dropped from the network or is replicated, creating multiple copies.
4. **Network deadlock:** Packets are blocked waiting on each other to free resources in a way that none of them can advance forward. Unless the deadlock is resolved, packets involved in a deadlock are permanently blocked.
5. **Packet livelock:** A packet continues to move through the network, but it never reaches its destination.
6. **Starvation:** A packet is temporarily blocked waiting to acquire resources that are continuously given to other packets.
7. **Misrouting:** A packet is delivered to the wrong destination or it is sent along a path that violates the network's routing protocol.

2.3.2 Network-level End-to-End Error Detection

Communication errors during a test execution can be detected by monitoring the inputs and outputs of the network and comparing the input/output data against a golden model of execution, if it is available. Additionally, errors can also be detected by checking that the network's end-to-end input/output data meets certain correctness invariants. An example

of a basic correctness invariant of a NoC design states that the number of packets delivered by the network must match the number of packets injected into the network. Another possible end-to-end checker consists of ensuring that for every packet delivered, its destination matches the ID of the node it was delivered to. Such end-to-end checkers can be implemented as hardware modules or assertions [18], so that they run along with the DUV, performing the checking functionality upon the delivery of every packet. They can also be implemented as software algorithms, running on the host machines that interface with the verification platform. In the latter approach the checking functionality may either be invoked at the end of execution or periodically as the relevant input/output data that must be checked is transferred between the platform and the host.

2.3.3 Error Detection through Localized Checkers

While network-level end-to-end checks may succeed in flagging a communication error, they do not always provide enough information to help localize and debug the source of that error. Additionally, in some cases, end-to-end checks may have long error detection latencies, especially if the error is only caught at the end of execution or after it has caused the test to fail. A long detection latency can obfuscate the source of the error, and it leaves a long window of execution that must be analyzed for debugging. Better error detection capabilities can be achieved by equipping the verification framework with localized checkers to monitor the internal operations of the network and flag potential bugs close to the time and source of their occurrence. In this section, we discuss the use of localized checkers deployed at the level of routers as a way to achieve fine-grain functional error detection in NoCs. We explore the implementation of these checkers as hardware modules, synthesized along side the DUV, or purely software models running within the verification environment.

Hardware Checkers

One approach to detecting communication errors in a network-on-chip interconnect relies on using hardware-based checkers, where checker modules are added to routers to monitor their execution and flag any suspicious behavior on-the-fly. In the literature, there have been several works that have explored the use of such hardware checkers, but many of these works primarily targeted the area of NoC runtime reliability in the face of faults [57] [31]. For example, in [57], authors propose 32 different correctness invariants to cover the control-logic of a typical router architecture. These invariants are then implemented as hardware-assertions, which are added to the router's design in order to provide near-

instantaneous detection of anomalies in the router's operations during runtime execution. On the other hand, other works have explored hardware-based error-detection, specifically to detect and recover from deadlock errors in routing algorithms [12, 59, 10]. In this section, we survey some of these proposed hardware-based checking approaches, with a focus on those techniques that have been utilized to detect the type of errors highlighted at the beginning of this chapter.

Packet data corruptions: A typical approach to detecting data corruptions in-transfer uses error-detection codes (EDC) added to each packet upon its injection into the network. At the destination node, the packet is checked against its EDC and any mismatch flags a data corruption error. In the area of runtime fault-tolerance, this method is typically used in conjunction with a retransmission scheme that allows the erroneous packet to be retransmitted form the source-node, upon the detection of the error. The same type of checking can also be applied at a flit-granularity, where individual flits are augmented with error-detection code, as well as router-granularity, where the checking is performed at every router along a packet's path. This latter technique is known as switch-to-switch error-detection [51].

Dropped and duplicated flits: Generally, an error causing the loss of a data flits or the creation of a new flit within a packet can be considered a type of data corruption and can be detected using the same approach of adding error-detection code to the packet. However, to particularly flag a dropped/replicated flit error, a router can keep track of the number of flits it observes for each packet and then compare that number against the packet's size, which is determined upon its injection into the network. In practice, this is often implemented by adding a *size* field to the header flit of every packet. A simple counter and comparator added to every input buffer are then used to keep track of the number of flits observed. If the tail flit is reached and the counter does not match the size field, then a flit must have been duplicated, created or dropped.

Dropped and duplicated packets: Dropped and duplicated packets can be detected by maintaining a packet counter per router to track incoming and outgoing packets [31]. The counter is incremented upon receiving a tail flit and decremented upon sending one. If packets are not dropped within the router, then the counter should reach a value of zero at some point within a checking window. Similarly, a packet counter reaching a negative value can be used to identify packet duplication or spurious packet creation. In the case of dropped packets, it is possible for this approach to exhibit false positives, particularly under high congestion traffic. High congestion can also mask packet duplications, caus-

ing this technique to exhibit false negatives. However, [31] shows that choosing a suitable checking window size can practically eliminate false positives. Moreover, false negatives are rare and duplications will eventually be detected.

Deadlock: Network deadlock is a widely explored topic in network-on-chip interconnects, especially in the context of runtime deadlock detection and recovery. One of the simpler techniques to flag potential deadlocks uses the inactivity period of a blocked packet as an indication of a deadlock [12, 13]. [12] implements this technique using counters, added to each router, such that one counter is associated with each input buffer. After a header flit, marking the beginning of a new packet, reaches the head of an input buffer, the corresponding counter is incremented at every cycle. The counter is reset when the tail flit is observed. If the counter exceeds a user-defined threshold, then the packet is assumed to be blocked and a deadlock is flagged. Similarly, [48, 49] proposes to monitor the inactivity period of a physical channel, where long periods of inactivity that exceed an acceptable threshold value are identified as deadlocks. However, one of the main problems encountered in these threshold-based approaches is that they are susceptible to flagging false positives, due to their inability to distinguish a true deadlock scenario. This problem particularly arises when network channels are occupied by long packets, when the network is heavily congested, or when the threshold value is not well-calibrated. In fact, a study presented in [67], shows that a single static threshold value cannot satisfy all types of traffic patterns. In the context of implementing adaptive routing algorithms, false positives lead to unnecessary deadlock recoveries and a consequent performance degradation. As a result, alternative approaches that rely on more sophisticated mechanisms for deadlock detection have been proposed. The authors in [44, 59] consider the sequence of packets involved in a deadlock as a tree whose root is the packet that is advancing. When the packet at the root of the tree later becomes blocked, then their solutions marks the packet that was blocked due to the root as deadlocked, if all of the packet's potential output channels have been inactive for a given time-out value. Other techniques such as [42, 62] rely on forwarding control packets along inactive networks channels to identify cycles. Whereas, the work proposed by [10] accurately identifies deadlocks at runtime by employing transitive closure computation to discover deadlock-equivalence sets, which imply the existence of request loops in the network.

Starvation: Design bugs in the arbitration and allocation modules of routers can lead to starvation errors, where a packet's requested resources are continuously granted to other in-flight packets. From an end-to-end perspective, if starvation errors occur frequently,

then these bugs will also manifest as a degradation in the network's end-to-end latency and throughput. However, the network can also be instrumented to identify potentially starved packets by equipping routers with simple checkers to track the progress of packets through them. Similar to the counter-based approach proposed for deadlock detection, the work presented in [31] adds a time-out counter to each router's input port to count the number of cycles the input port was not granted an output channel. As in the case of deadlocks, using only a time-out counter has the potential of flagging false positives, not to mention that it cannot distinguish a starvation error from a true deadlock scenario.

Livelock: The ability to detect and recover from livelocks is important in enabling highly adaptive routing algorithms in networks-on-chip, and there are numerous solutions proposed to avoid the occurrence of livelock scenarios when using such routing protocols. For example, Time-to-live (TTL) counters are commonly added to packets to keep track of how long they stay in the network [58]. When a counter times-out, a mechanism is then provided to ensure that the packet will be delivered or the packet is dropped and retransmitted. In line with this solution, a hop counter can also be utilized instead of TTL. A counter is added to the header flit of every packet and it is incremented at every hop. If the packet traverses more hops than a preset threshold, then a possible livelock is flagged [24]. Other approaches to detect and recover form livelocks provide each packet with a priority, which is based on the packet's age, such that the oldest in-flight packets are eventually given the highest priority and are guaranteed forward progress [58]. In the context of functional verification, similar techniques can be used to identify the occurrence of livelocks in the network.

Misrouting: One way for a packet to be misrouted is if it is sent to the wrong destination. Such an error can be detected by adding a comparator to each router in the network. Upon each packet's delivery to its final destination, a simple check ensures that the destination field in the header flit matches the destination node ID [57]. Misrouting could also occur if a packet is delivered to the correct destination, but takes incorrect routes along its path. In this case, whether the misroute can be classified as an error depends on the network's routing protocol. Deterministic routing protocols assign a single unique path between each pair of source and destination nodes, and any packet that deviates from the set path is erroneous. Fully adaptive algorithms adapt routing decisions based on the state of the network, such that there are no constraints on the paths taken and all paths are considered correct. However, fully adaptive routing algorithms are often implemented with deadlock and livelock avoidance measures that may restrict the paths that can be taken by packets.

For example, one approach to avoid deadlocks in fully adaptive routing algorithms relies on restricting the directions in which packets can turn in the network to prevent the occurrence of cyclic dependencies [32]. Other non-deterministic routing algorithms require that packets are routed only along the shortest (minimal) paths between a pair of source and destination nodes. In a network implementing a minimal routing routing protocol, any time a packet traverses a non-minimal path, then that packet is erroneous. Therefore, in many cases, routers can be instrumented with simple checkers to identify misroutes, but the exact checker implementation is dependent on the routing protocol and the network architecture. For example, for deterministic routing, a look-up table or a hardware assertion can flag a misroute error when a packet arrives at a router that should not be on that packet's source-destination path [31, 57]. Similarly for networks using minimal routing protocols, a router can check, upon receiving a new packet, whether the router falls within the minimal quadrant between that packet's source and destination nodes. If the router is outside the minimal quadrant, then the packet is following a non-minimal path and an error is flagged.

Software-based Checkers

Functional errors during verification on emulation platforms and during post-silicon validation can also be detected using software-only checkers, implemented a part of the testbench environment and running on the host computers that interface with the hardware platforms. However, this requires the framework to record the relevant signals, events, and/or state elements that are needed to perform the checking and then to periodically transfer this information to the host machines. In the previous section, we introduced the use of distributed hardware checkers, localized at individual routers and responsible for monitoring each router's execution to identify errors. In this section, we explore an equivalent software-based implementation of these checkers. We assume an ideal framework that observes and logs the sequence of packets entering, residing in and existing each router, along with the timing of these events. At the end of execution, an activity log for each router is available, and this data is off-loaded to the host for analysis. Based on this setup, we describe the erroneous behavior that a software-based checker must identify to be able to detect the error categories highlighted at the beginning of this chapter. Note that in practice, it may not be possible for the verification platform to record as much details from the network's execution. This limitation affects the detection capabilities of these software checkers, and we further elaborate on this trade-off in Section 2.3.3.

Packet data corruption and dropped or duplicated flits: For the purposes of implementing a software-based checker, we group packet data corruptions and dropped/duplicated flits

together, as they are both errors that compromise the data integrity of packets. Such errors can be detected using error-detection codes (EDCs) computed for each packet upon its injection. Then, the packet's contents are verified by recomputing the error-detection code and matching it to the original EDC value. This verification can be done once the packet is delivered at its final destination node or at every router it traverses. It can also be easily performed through a software function, provided that the verification framework records all injected packets and their associated EDCs.

Dropped or Duplicated Packets: At the network-level, the dropping of a packet or the creation of a new packet manifests as a mismatch between the number of packets injected into the network and the number of packets delivered. This mismatch can be checked using a simple software assertion. Additionally, by recording the statistics of packet arrivals and departures from each router, a mismatch can easily be detected at internal network routers.

Livelock: A network livelock exists if a packet is being transferred through routers but not advancing to its destination. Since the network has a finite number of nodes, a livelocked packet will eventually traverse the same router twice. Given a log of the packets traversing each router, a software-based checker can detect a livelock by identifying a packet being present at the same router at different non-consecutive times. Moreover, given an overview of packet traversals in all routers, then the path of each packet can be reconstructed to identify livelock cycles.

Deadlock: Deadlocked packets are blocked at routers, unable to progress forward, due to the existence of a cyclic dependency among the resources requested by these packets. A blocked packet can be identified if it arrives at a router, is stored in one of its buffers, but never leaves. By independently analyzing the packet traversals through each router, a software checking function flags these blocked packets. Then, based on the network channels that these blocked packets are requesting, a software-based checker can then proceed to analyze the logs of subsequent routers to build a channel dependency graph and determine the existence of a cycle. Channel dependency graphs are commonly utilized to theoretically ensure deadlock-freedom in some routing protocols [25].

Starvation: Similar to deadlock errors, a packet that is blocked for a period exceeding a pre-defined threshold can be flagged as a potentially starved packet. Further analysis of the resources requested by that packet can be performed to determine whether these resource are inactive (also blocked) or were granted to other packets.

Misrouting: Misrouting errors can be detected by having a software checker iterate through the log of packets that traversed each router. Depending on the network's routing algorithm, it is then straightforward to check whether a packet traversed an incorrect path between its source and destination nodes. Alternatively, the logs from all routers can be aggregated to obtain the path traversed by each packet, and then an analysis of these paths would verify whether the network's execution conforms to the routing protocol.

Trade-offs of Hardware and Software Checkers

The main advantage of instrumenting network routers with localized checkers is that these checkers operate along-side the design and flag erroneous behaviors as soon as they occur. However, adding hardware for the purposes of verification incurs an area overhead that may not always be desirable. In an emulation-based verification environment, it is typical to map some verification modules, along with the design, onto the configurable hardware platform. Nevertheless, the emulation platform has limited hardware resources and it is not feasible to implement checkers with a large area footprint. This problem is even more significant in the context of post-silicon validation, where the added hardware verification features become a permanent part of the released chip. Besides area constraints, there is often a trade-off between the complexity and size of the checkers and their detection accuracy. For example, some types of errors, such as deadlocks, are distributed in nature, spanning multiple routers and involving multiple packets. Such errors are not always accurately detected by a local checker that monitors the operations of a single router, independently of other routers in the network. However, in some cases, design-time verification efforts afford to trade-off some detection accuracy for smaller and simpler checker implementations, particularly if the loss of accuracy is in the form of false positives, which lead to unnecessary debugging and analysis to eliminate them. Therefore, the choice of checkers is ultimately a verification decision that tries to balance the trade-off between the desired detection capabilities and the feasibility of implementation.

On the other hand, relying on software-based checkers avoids incurring an area cost for error detection. It also allows flexible checker implementations, which can be written in high-level languages such as C/C++ and SystemVerilog. Moreover, since the checking functionality is performed offline, on the host, the complexity of the checker's implementation is not a limiting factor. Thus, checkers can be designed to analyze debug data from multiple routers and check for correctness invariants at the network-level as well as the router-level. However, the main drawback of a software-approach to error detection is that it requires the verification framework to monitor the network and collect detailed logs of

execution. An added advantage is that these logs can later be leveraged for debugging. In practice, all emulation and post-silicon platforms allow for the collection and transfer of debug data. However, recording large amounts of data requires dedicating large on-chip trace buffers to store it. Moreover, having very frequent data transfers between the verification platform and the host machines slows down execution, which must be halted until the transfer is complete. These limitations are typically countered by sacrificing some observability and monitoring a smaller subset of events and signals. As a result, in most cases, the execution logs that can be collected are incomplete, which in turn reduces the error-detection capabilities of software-based checkers. As an example,if records of a packet's traversal is missing from some routers' execution logs, then an error affecting that packet can go unnoticed by a software checker that depends on these logs to perform the analysis. Nevertheless, using a software-based approach to perform error detection remains a viable alternative, especially when introducing an area overhead to the emulation and post-silicon platforms is a concern.

2.4 Summary

This chapter provides an overview of common practices in design-time verification, focusing on emulation and post-silicon validation platforms as they constitute a major and growing fraction of design-time verification efforts. We also describe recent research works that aim to improve the emulation and post-silicon validation process, including approaches that target the verification of network-on-chip interconnects. Lastly, we discuss the need for equipping emulation and post-silicon validation frameworks with error detection capabilities that can flag functional errors in the interconnect's execution. In this context, we first highlighted the conditions that must be met to ensure a network's functional correctness, from which we identified seven types of communication errors that could arise. We then discussed different approaches to detect these communication errors, including end-to-end network checkers and localized hardware and software checkers.

Chapter 3

Enhancing Observability of Emulation and Post-Silicon Verification

The verification of the interconnect subsystem on emulation platforms and during post-silicon validation is hindered by the limited observability into the design's internal behavior. Basic observability is attainable by monitoring the NoC's input and output data, which consists of the packet's injected into the network and ejected from it. However, network-on-chip interconnects are distributed and parallel communication fabrics, and observing a network's end-to-end inputs and outputs provides a very restricted view of its execution. Ultimately, for efficient detection and debugging of functional bugs in the interconnect design, it is critical to instrument the network with a monitoring solution that provides more comprehensive and detailed insights into the interconnect internal behavior throughout a test's execution.

In this chapter, we first identify the critical set of internal signals, events, and/or data element that encapsulate as much of the network's functionality as possible. Then, we explore how this identified data can be monitored and logged during execution to provide debugging support. To this end, we distinguish between two main approaches. In the first approach, in-flight packets are monitored and the debug data that is logged pertains to the progress of these packets through the network. In the second approach, the network's routers are monitored and the debug data that is logged corresponds to execution events observed at the routers.

3.1 Identifying Critical Debug Information

A deeper look into the design paradigm of network-on-chip interconnects demonstrates that the overall execution flow of the network is almost entirely dependent on the traffic traversing the network at the time, such that different traffic patterns and injection times may result in completely different executions. Thus, if the verification framework can

track how packets traverse the network and the timing of their arrivals and departures at all routers along their paths, then we can gain significant insight about the router-level and network-level operations that occurred. The first step in accomplishing such a tracking scheme is for the verification framework to have the ability to distinguish and identify different packets throughout execution. This is commonly implemented by tagging each packet with a unique identification number that is added to the packet upon its injection into the network. In addition, the verification framework must also log the sequence of routers each packet traverses, starting from the packet's injection at the source node to its arrival to its destination node. To gain further observability of the router-level operations and control decisions, it is also necessary to identify and log the timing of packet traversals through routers along their paths. This includes logging packet arrival and departure events to and from each router. Using these events, we can start to reconstruct an overview of the interactions among packets within routers and the corresponding contestation for network resources. Although packet arrival and departure times aid in understanding a router's control decisions, they may not be enough to facilitate the localization and debugging of design bugs within the router architecture. Thus, it may also be benificial to monitor a router's main logic components, particularly the arbitration and allocation modules that determine the management of resources within the router and the network, as well as the order by which packets are allocated to these resources.

3.2 Monitoring and Logging Debug Data

A common practice in emulation-based verification and post-silicon validation is to map verification features along with the design onto the verification platform. In the context of emulation, these features include functional checkers, monitoring logic, as well as trace buffers to store any debug data collected from the execution. Similarly, post-silicon validation utilizes various design-for-debug features that can be configured to monitor a specific set of events and extract the relevant debug information. In a network-on-chip interconnect, execution is driven by the control logic within routers that dictate the allocation and management of network resources to in-flight packet. Therefore, gaining observability of the network's internal behavior is best achieved using a distributed monitoring framework, where each router is instrumented with a monitoring and logging unit (MLU). The MLU of each router identifies and logs events-of-interest occurring within that router. The exact design of the MLU, including the type events it logs and how it logs them, ultimately depends on the verification platform in-use and the debugging resources available on it.

Similarly, depending on the verification setup, the debug data extracted from each router's MLU can be stored in trace-buffers, the design's memory and cache modules, or extracted off-chip to an external memory, as discussed in Chapter 2. In this chapter, we explore two approaches to monitoring and logging debug data from routers. In the first approach, upon the arrival of a pakcet to a router, the router's MLU logs debug data outlining the progress of that packet within the router. Thus, the debug data collected at each MLU is grouped per packet, and since there may be multiple packets traversing a router at the same time, multiple such groups may need to be logged simultaneously. We refer to this approach as packet monitoring. On the other hand, in the second approach, an MLU views its corresponding router's execution as a series of events occurring over time. At every cycle, the data that is logged captures the state of that router, including events pertaining to all packets traversing it at that time. We refer to this approach as router monitoring. Both the packet monitoring and router monitoring approach can be used to monitor the same set of events. However, in the first, the data that is collected at each router is associated with the packets traversing it, and is generated at the rate at which packets progress. Whereas, in the second approach, the data collected at each router is associated with the router itself and is generated at every cycle. In practice, these differences affect how each approach is implemented and the overall gain in observability that each can provide.

3.2.1 Packet Monitoring

One way to view the interconnect's execution is as a series of packets traversing the interconnect over time, with each packet arbitrating for and acquiring network resources at every hop along its path. From this perspective, a verification framework can attain observability of the interconnect's internal execution, by monitoring the traversal of every packet at every hop along its path. This can be accomplished by configuring each router's MLUs to log compact pieces of debug information for every packet that traverses the router. In such a scheme, the MLU first identifies when a new packet is delivered to its associated router. Then, as the packet progresses through the different router stages, various events pertaining to that packet are incrementally collected. Based on the critical debug information identified in Section 3.1, these events can range from the time of the packet's arrival and departure, to its input and output directions, the resources it occupies, among many others. Moreover, in the cases when multiple packets are traversing the same router at the same time, that router's MLU logs one debug data entry for each packet. Each entry describes its corresponding packet's progress through the router, independently of the other packets that are present at the time. The observability achieved from relying on such

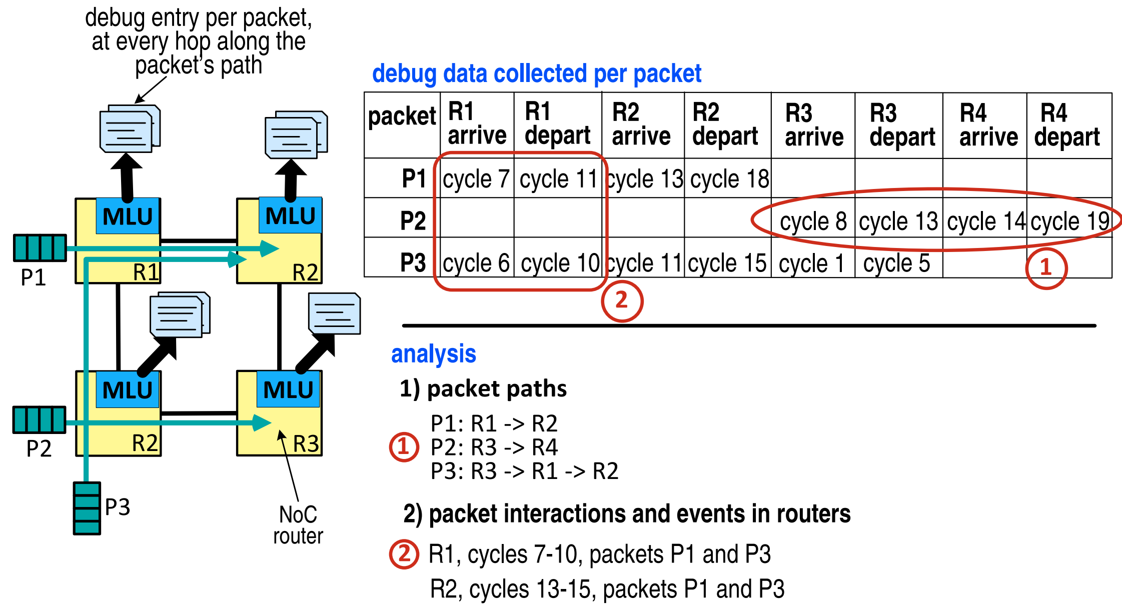

Figure 3.1 Packet monitoring. Using a packet monitoring approach, each router's MLU logs a debug entry for every packet that traverses the router. By the end of execution, the debug data that was collected is associated with the packets that traversed the networks. An example of the debug data log, showing only arrival and departure times of packets, is shown in the figure. This information is then analyzed to reconstruct the paths of packets as well as the interactions of packets in the network.

a packet monitoring approach is two-folds. First, by examining the debug data of each packet, we can generate a complete view of the path it followed. Second, by examining the debug entries generated by each MLU, we can reconstruct the sequence of operations observed at each router, and the interactions of packets within the router. Together, the internal traffic flow and the router-level operations and interactions, provide valuable insights into the network's execution, thereby facilitating the detection and debugging of functional bugs in the interconnect.

Figure 3.1 illustrates an example showing our packet monitoring approach. There, packets P1, P2, and P3 traverse a set of routers, and debug data is collected for each packet at every hop along its path. The right side of the figure highlights the collected debug data, which includes the arrival and departure events of these three packets to and from each router. An analysis of this data reveals the path traversed by each packet, as well as the packet interactions within routers. Note that, in this example and for simplicity, the debug data only includes packet arrival and departure events at routers. However, in practice, our packet monitoring approach can be configured to log other types of details, such as a packet's input port and input virtual channel, the output resources it requests, *etc.*

In a standard emulation or post-silicon verification platform, the debug data collected

at each MLU must be extracted from the MLU to be stored in designated trace buffers or other memory modules. One implementation option consists of transferring the debug data from a router's MLUs to the trace buffers upon every change in a packet's state within the router. However, with multiple packets traversing each router and successive events being observed for every packet, this option can easily create a bottleneck between the MLUs and the designated storage locations. This bottleneck is alleviated by equipping each MLU with a small buffering space to store the debug entry of each packet. Therefore, a packet's debug information is logged and consolidated in its corresponding debug entry, after which the MLU transfers the debug entry out of the router only once, at the time the packet leaves the router. However, matching the rate of debug data transfers to the rate at which packets are sent out of a router is not always sufficient to eliminate this bottleneck. For example, considering a general virtual channel router design in a mesh-type interconnect, routers that are not on the boundary of the network have five active input and output ports, with at least 2 virtual channels per port. These routers are theoretically capable of sending out at least 10 packets at the same time, which requires a prohibitively high bandwidth link between each MLU and the on-chip trace buffers to transfer the debug data associated with these packets. A lower bandwidth link can be sustained if the MLUs are equipped with larger storage buffers or if the execution stalls to allow enough time to transfer the debug entries. This problem may also be addressed by monitoring a smaller subset of in-flight packets, which inherently leads to less debug data being logged. However, as we evaluate in the rest of this Section, monitoring fewer packets greatly reduces the observability that can be attained from this technique and lowers the verification framework's debugging capabilities.

Downsizing the Debug Data

Monitoring a uniformly distributed sample of packets: A simple approach to reduce the amount of debug entries generated by each router's MLU is by monitoring a fewer number of packets. Once a packet is chosen to be monitored, then it is tracked through every hop along its path. Thus, the monitored packets effectively constitute sample points of the network's execution, and uniformly distributing the chosen packets over time would provide the most consistent view of execution. To implement this down-sampling, each source node maintains a running counter of the number of packets it injects into the network and marks every *nth* packet as a packet-to-be-monitored. Evidently, there is a trade-off between the sampling period, *n*, and the observability that can be gained from this solution. With a low sampling rate, it is more probable that a packet affected by a functional bug is one that

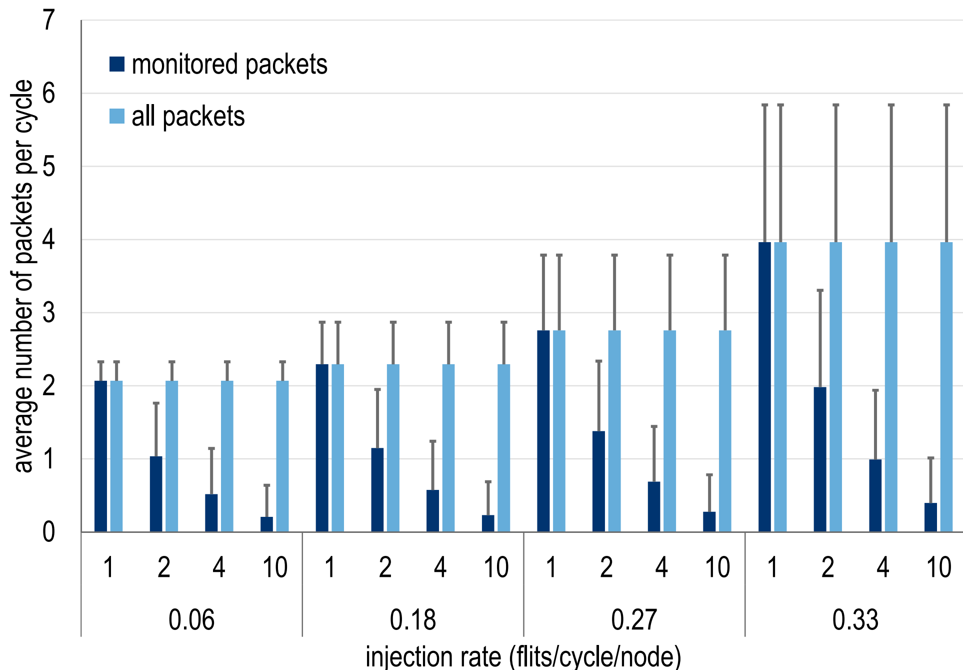

Figure 3.2 Monitored packets observed at routers when using a packet monitoring approach with uniform sampling. The x-axis shows varying monitoring periods (1,2,5,10 cycles) and varying injection rates, from low injection (0.06 flits/node/cycle) to high injection (0.33 flits/node/cycle). When the sampling period increases, only 1-2 monitored packets, on average, can be observed in routers at each cycle. The remaining set of packets traversing the router and their assosciated events are not observable.

is not being monitored. Hence, there would be no debug data associated with this packet to help localize and debug the error. However, even when the packet affected by the bug is a monitored packet, its debug data only reveals its state at different routers, but provides no information about the other events occurring at those routers. Whereas, examining the debug data of other packets that traversed the same set of routers around the same time, allows us to reconstruct the execution conditions at these routers. A low sampling rate reduces the probability of interactions between monitored packets, which in turn diminishes the debugging value of this technique.

To evaluate the effects of uniform sampling on our packet monitoring approach, we modeled an 8x8 mesh network using the cycle-accurate Booksim simulator [24]. We ran uniform traffic, while varying the injection rate from low injection (0.06 flits/cycle/node) to high injection (0.33 flits/cycle/node). In each run, we also varied the sampling period, n, from 1 to 10. A sampling period of 1 means that 100% of all packets injected are being monitored. On the other hand, when n is set to 10, each source node marks every 10th packet it injects as a packet-to-be-monitored, such that approximately 10% of the overall injected packets are being monitored. Based on this setup, we evaluate the number of

monitored packets (i.e packets marked as monitored) observed at a router for every cycle when that router has at least two packets traversing it. Figure 3.2 shows our results, averaged over all routers in the network and over all execution cycles. For comparison, the graph also shows the average number of packets traversing routers at each cycle. When the sampling period is set to 1, all packets are being monitored and all interctions between these packets are observable. However, a sampling period of 2 (50% of all injected packets are being monitored), shows that in most cases, only 1-2 monitored packets are observed in a router at a given time. Therefore, in these cases, only the interactions between the monitored packets are observable. Whereas, the remaining packets present at the router and the execution events associated with them are not visible. This problem is amplified at higher injection rates and larger sampling periods For example, at an injection rate of 0.33 flits/cycle/node, routers are traversed by as many as 4-6 packets, on average. However, with a sampling period of 10, only 1 of these packets is observed, on average, and no debug information is available for the remaining majority of packets. Thus, relying on a packet monitoring technique along with uniform sampling does not provide enough observability of each router's internal operations.

Monitoring bursts of packets: The likelihood of capturing interactions among monitored packets can by increased by monitoring bursts of packets. At periodic intervals, each source node marks a certain number of successively injected packets as packets-to-be-monitored. Marked packets are then observed throughout their network traversal, with debug data collected at every hop along their path. By monitoring packets in bursts, then there is a larger number of monitored packet traversing the network in a given interval. These packets are more likely to encounter one another at different routers, and the debug information obtained from these packets provide greater insights into the routers' internal operations. However, as with uniform sampling, if the frequency of sampling is low and the burst size is small, the few packets being monitored can significantly reduce observability and debugging capabilities.

3.2.2 Router Monitoring

The interconnect's execution can be also be considered from the perspective of the routers and how they view and respond to in-flight packets over time. With such an approach, the monitoring and logging unit (MLU) of each router observes the router's execution at every cycle and logs events-of-interest, such as the arrival or departures of packets, the allocation of virtual channels and crossbar connection, among many others. In this context, the debug data that is logged at every instance holds information about all the packets that are travers-

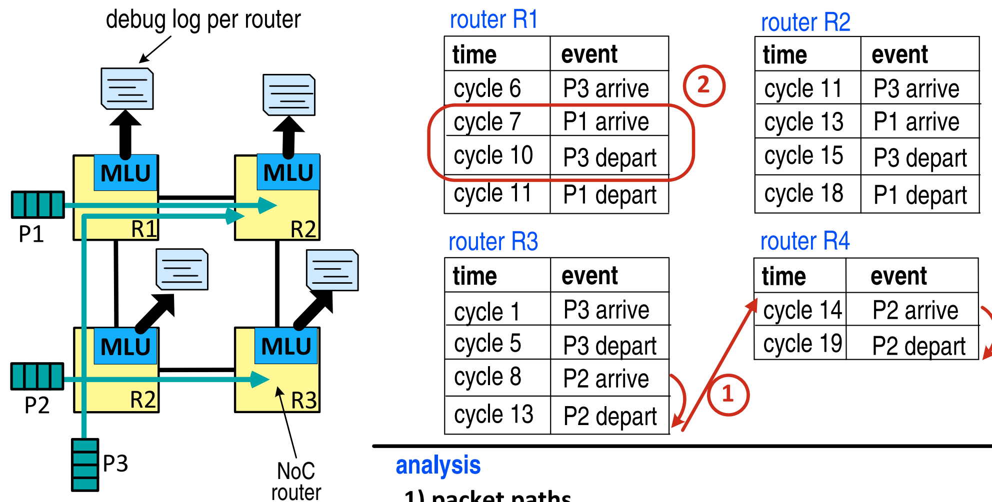

Figure 3.3 Router monitoring. At every cycle, each network router logs the events it observes. An example of the debug logs is depicted in this figure, showing only packet arrival and departure events. Grouping events from the logs of multiple routers permits the reconstruction of the traffic flow through the network. In addition, each router's debug log also reveals the packet interactions occurring in the router throughout the test's execution.

ing that router at that time. In contrast to packet monitoring, the data collected from each router inherently captures the packet interactions within the router, as well as the router's operations. Even further observability can be gained by aggregating the debug data from across routers, which allows the reconstruction of the traffic flow through the network.

Figure 3.3 depicts an example of our router monitoring approach. Each router's MLU logs a cycle-by-cycle trace of execution consisting of all packets and events observed at the router. Note that, for simplicity, the debug logs in our example only show events related to the arrival and departure of packets to and from the network routers. At the end of execution, the collected debug data from each router is analyzed to determine the router's internal operations and packet interactions. Additionally, examining the logs from multiple routers allows reconstructing the path followed by each packet.

Downsizing the Debug Data

Full observability of the network's execution is achieved when debug data is collected from each router cycle-by-cycle, revealing every packet that traversed it and the router's control

decisions made as a response to these packets. However, test executions can cover millions of cycles, and in each cycle, multiple packets are traversing a router. The result is that a large amount of debug data is generated by each MLU, and this data must be transferred from each router to the designated storage memories. In practice, this rate of debug data generation is not sustainable, and we consider two approaches to downsize the debug data and make it more manageable.

Monitoring periodic intervals: One downsizing approach consists of periodically monitoring short intervals of execution. During each monitored interval, every router logs information about the packets traversing it and the resources being allocated to them. Although this scheme reduces the amount of debug data to be logged overall, its implementation faces two main challenge. The first problem is that there are windows of execution where there is a complete lack of observability of the network's internal operations, and any functional bugs manifesting within this interval cannot be debugged. The second problem is that debug data must still be extracted from each router's MLU, at a cycle-by-cycle rate, throughout the monitored intervals. As in the case of packet monitoring, this requires a high bandwidth interface between every MLU and the designated debugging storage units. If the transfer rate cannot match the debug data generation rate, then the debug data must be dropped or the MLU must be equipped with its own buffering storage to accumulate the data that cannot be transferred on-the-fly.

Monitoring periodic snapshots: Although, a cycle-by-cycle trace of a router's execution provides complete observability of that router's internal operations, it may be overly detailed and in some cases unnecessary. Therefore, we consider a new technique to downsize the debug data that relies on taking snapshots of each router's execution at periodic intervals. Each snapshot captures the corresponding router's packet contents and control decisions at the time the snapshot is taken, providing uniformly distributed samples of the router's operations. The advantage of this approach is that the time between snapshots provides a window during which the debug data can be collected from each router, without requiring this data to be dropped or accumulated within routers. In this setup, when snapshots are taken relatively frequently, we can still achieve a high degree of observability throughout execution. Additionally, depending on the snapshot frequency, the execution over the time intervals between consecutive snapshots can be partially, or in some cases fully reconstructed, based on the adjacent snapshot samples.

Chapter 4
Packet Monitoring

Chapter 3 introduced packet monitoring as an approach for collecting debug data from network-on-chip interconnects during functional verification on emulation and post-silicon platforms. In this approch, a debug entry is collected for each packet at every hop along its path. When the debug entries of a packet are aggregated, the path followed by that packet through the network can be reconstructed. Moreover, the debug entries collected by a single router constitute a trace of events observed at that router. The traffic flow and router-level operations that can be observed using this technique provide valuable insights into the network's execution and facilitate error detection and debugging on these verification platforms. However, as discussed in Chapter 3, using packet monitoring to boost observability and debugging capbailities ultimetely requires monitoring most, if not all, of the packets injected into the network. In practice, this translates to a large number of debug entries that must be collected from the network's routers to be stored in the verification platform's designated trace buffers and memory units.

In this chapter, we introduce DiAMOND [5], a novel solution that utilizes packet monitoring to enhance observability by relying on the in-flight packets themselves to collect the debug data from network routers. DiAMOND is based on the observation that an important phase in the functional verification process consists of running randomly generated test cases that are aimed at exercising as much of the network's functionality as possible. In these randomly generated tests, network traffic is generated using constrained random methods or according to established statistical models. In this context, the traffic flow through the network is entirely independent of the data contents of the generated packets. Moreover, the network's execution is only driven by the spatial and temporal distribution of traffic, and not the data contents of packets. Therefore, DiAMOND proposes to replace packets' data contents with the debug information collected during the network's execution. At every router along a packet's path, we gather debug data that encapsulate the packet's current state and we systematically substitute the data flits of that packet with this information. Once packets arrive to their final destination nodes, they are transferred to the

designated verification trace buffers, such that at the end of execution all the debug data that was collected resides in the contents of the delivered packets.

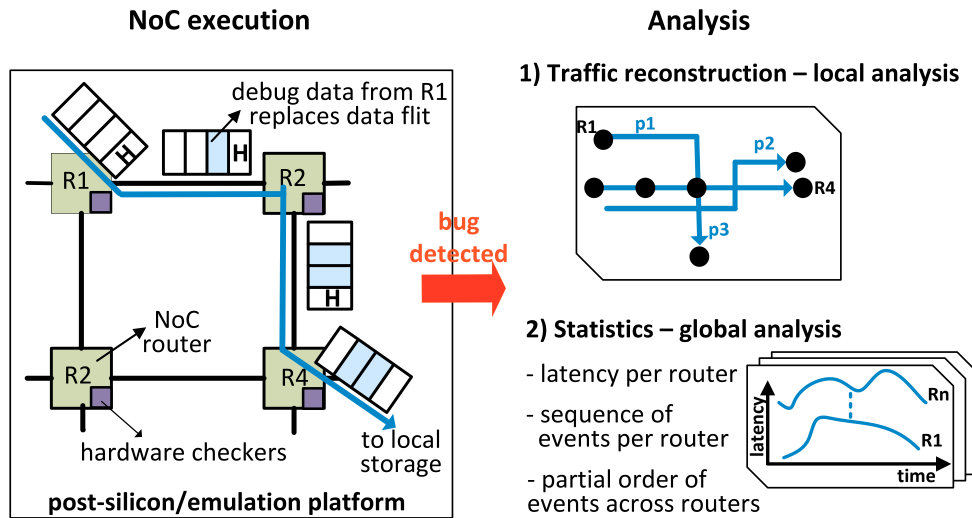

Figure 4.1 Overview of our solution. During NoC execution, debug data is collected at every hop and stored in the packet, overwriting data flits. Routers are instrumented with hardware checkers that monitor execution and flag functional errors. Upon error detection, the debug data is analyzed to reconstruct traffic, as well as provide a number of relevant statistics.

4.1 DiAMOND Overview

DiAMOND (**D**istributed **A**lteration of **M**essages for **O**n-Chip **N**etwork **D**ebug) provides enhanced observability of the NoC's internal operations by leveraging the data contents of packets to incrementally store debug information. For more efficient verification, DiAMOND also integrates a fine-grain hardware-based error detection solution to flag errors close to the time of their occurrence. DiAMOND partitions a test's execution into epochs, during which the network is instrumented for bug detection, as well as debug data collection. Figure 4.1 illustrates a high-level of our solution. As packets traverse the network, their data content is substituted with debug information collected at every hop. In parallel with debug data collection, small hardware checkers, added to each router, monitor the network's execution and detect functional bugs. Upon flagging an error, execution is halted and the debug data that has been collected is analyzed. On the other hand, if the epoch ends without the detection of any bugs, the debug information collected is simply overwritten in the following epoch. Furthermore, DiAMOND's debug data analysis is a two-phase process. In the first phase, analysis is carried out locally, such that each packet's

data is independently examined to reconstruct a detailed view of the packet's path through the network. Whereas, the second phase is a global analysis phase and debug data from all nodes are aggregated to obtain a comprehensive overview of the network's behavior.

In this chapter, we consider that the baseline design-under-verification is a general chip-multiprocessor, where each node consists of a processing element and a local cache. Therefore, once a packet is delivered to its destination node, DiAMOND stores it in the local cache associated with that node. Additionally, debug data analysis is performed using software-based algorithms running on the CMP cores themselves. By leveraging the CMP's processors and caches in the verification process, DiAMOND avoids the use of additional trace buffers as well as the periodic transfers of debug data off-chip for analysis. However, when the DUV is a SoC design, where the system consists of general purpose processors, as well as other IP modules, DiAMOND's validation flow can be adapted accordingly. For example, in an SoC design, only nodes with memory/cache modules store debug data, whereas other nodes can be configured to transfer this data to on-chip trace buffers. The debug data analysis process can also run on the available on-chip cores or performed off-chip. Similarly, DiAMOND must also be adapted when an emulation platform is used to validate the interconnect as a stand-alone component, or when the design's processor cores and caches are not being fully modeled. Figure 4.2 shows the complete validation flow of our solution and its integration within an emulation and post-silicon validation platform.

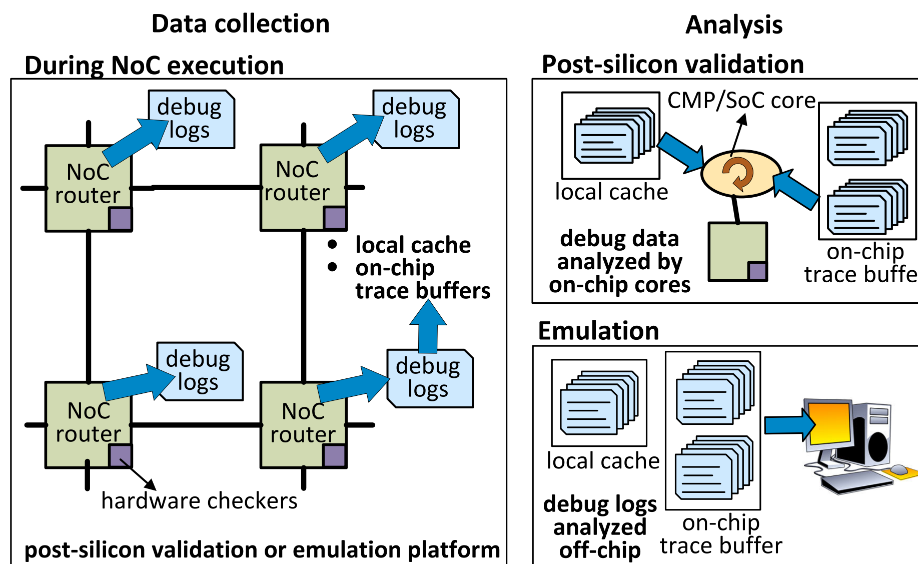

Figure 4.2 DiAMOND's validation flow. During the emulation or post-silicon validation of NoC designs, debug data is collected during testbench execution. The debug logs can be stored in the local cache or memory associated with each node or in on-chip trace buffers. Upon the detection of an error, execution is halted and the collected debug data is analyzed by the on-chip processor cores or can be transferred off-chip and analyzed separately.

4.1.1 Debug Data Collection

Debug data is collected for every packet injected into the network and at every hop during its flight. At each hop, the debug data associated with each packet is stored in one of the packet's flits replacing the original data content.

For every input buffer within the router, we add a register, called *log_buffer*, to store the debug information associated with the packet at the head of the input buffer. This *log_buffer* is updated everytime a new packet is at the head of its corresponding input buffer. In addition, we require each router to include a packet counter (*pckt_cntr*) that is incremented upon receiving a new packet. The information collected and stored in the *log_buffer* consists of:

1. The router ID
2. Arrival timestamp (*timestampA*) that indicates the value of the *pckt_cntr* when the header flit of the packet was received by the router.
3. Departure timestamp (*timestampD*) that indicates the value of *pckt_cntr* when the header flit was sent from the router. Logging timestampA and timestampD allows us to order packets passing through each router, as well as reason about packet interactions within a router.
4. A third timestamp (*pckt_latency*) that indicates the amount of time (in cycles) the packet's header flit remained in the router. This timestamp allow us to analyze packet latencies observed at interval routers.
5. The packet's input port and input virtual channel.
6. The output port and virtual channel the packet requests.

Once the *log_buffer* is complete, the debug data is written to one of the packet's body flits. The index of the flit to be written is maintained by a counter that is added to the packet's header flit. When a packet arrives to a router and reaches the head of one of its input buffers, we first extract the flit index where the debug data will be written. Then, *timestampA*, the input port and the input virtual channel are logged in the *log_buffer*. When the header flit completes its route computation and virtual channel allocation, the requested output port and requested virtual channel are logged. Finally, when the header flit is sent to the next router (or ejected if it is at a destination router), *timestampD* and *pckt_latency* are logged. Once the packet's header has been routed to the next hop, the packet's body flits follow. Based on the flit's write index, the *log_buffer* is simply written in the appropriate flit.

A typical flit width in NoCs is between 128 and 256 bits [45]. In our evaluation, we assume a flit width of 128 bits and a *log_buffer* size of 64 bits. Therefore, the debug data collected at every hop occupies only half a flit, with the remaining half written at the next hop. In order to implement this functionality, the flit write-index field in packet headers is extended by 1 bit, which indicates whether the debug data will replace the first or second half of a body flit. Moreover, the flit write-index field is incremented once every two hops.

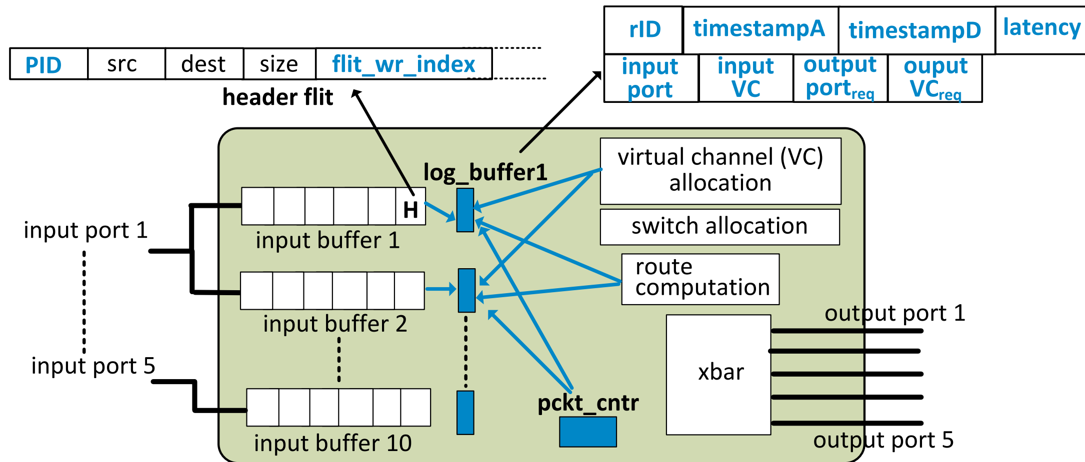

Figure 4.3 DiAMOND's debug data collection at NoC routers. A log_buffer register is added per input buffer. When a packet reaches the head of an input buffer, debug data is collected for that packet and stored in the associated log_buffer. A packet counter is also added per router to provide the *timestampA* and *timestampD* values. Additional fields are included in each packet's header flit to identify the flit index where the collected debug data must be written at every hop.

Figure 4.3 shows the architecture of a typical virtual channel router, as well as the hardware additions required for implementing DiAMOND's debug data collection mechanism along with the fields that must be added to each packet's header flit.

In the case of packets that do not have enough flits to store the collected debug information, we provide three solutions for our approach, depending on the needs of the verification methodology in use: *drop remaining*, *drop at alternate hops*, and *append*. Figure 4.4 illustrates the behavior of each mode. In these three modes, the verification process can be tuned to trade-off debug capabilities with the degree of perturbation introduced to the original system.

Drop Remaining

During drop remaining, when the number of hops in a packet's path exceeds the available flits in the packet, additional debug data is dropped. This mode of operation is simple to implement at the expense of low observability for packets with long routing paths.

Drop at Alternate Hops

In this mode of operation, debug data collection is implemented as before. However, when the space in the packet is exhausted, new debug data overwrites older debug data, creating an every-other-hop scheme. For example, as shown in Figure 4.4, the debug data collected

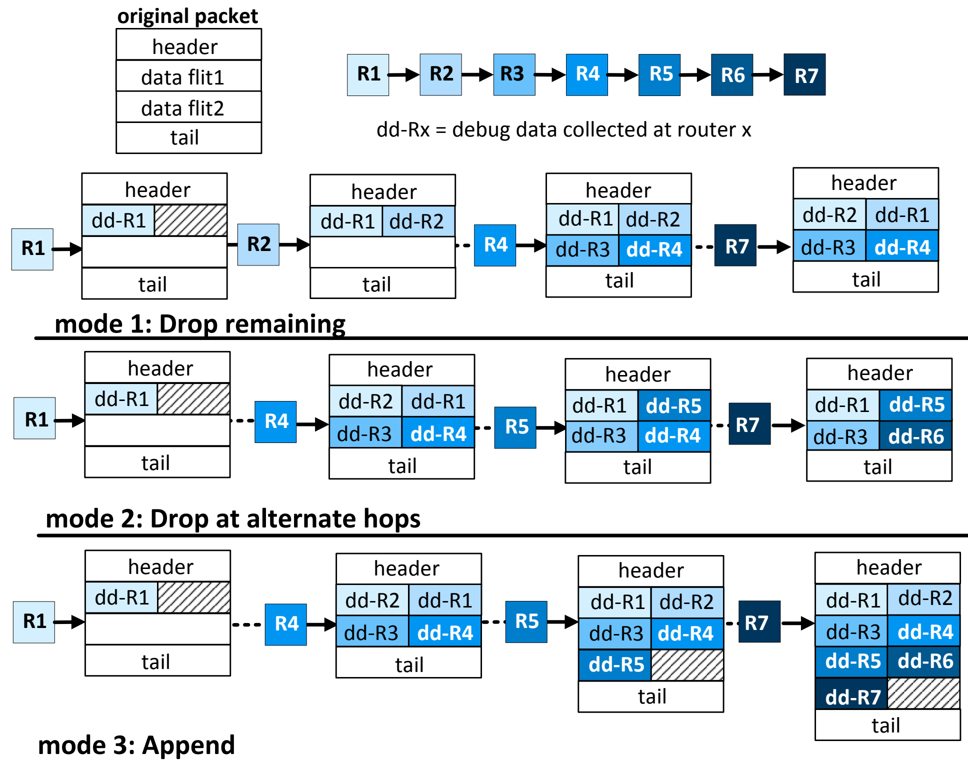

Figure 4.4 Modes of operation. In *drop remaining*, if the number of hops exceeds the available data flits, additional debug data is simply dropped. During *drop at alternate hops*, additional debug data replaces alternating entries providing a more uniform overview of the packet's path. The last mode, *append*, creates new flits as needed and appends them to the packet.

at router R5 replaces older debug data collected at router R2. As opposed to drop remaining, this mode provides a longer and more uniform overview even of long routing paths. Moreover, data belonging to the alternating missing hops can be partially reconstructed or extrapolated from the debug data that remains. For example, the output port requested along with the router ID can be used to determine the downstream router, for which we have no log. Similarly, the input port and router ID can be used to determine the missing upstream router.

Append

We also provide a mode of operation that allows routers to append new flits to the packet. New flits are appended before the tail flit, as illustrated in Figure 4.4. While this mode provides the complete path of a packet, it requires additional hardware to add the new flits. It also alters the network's original execution by creating longer packets, potentially masking bugs. It is also possible for the perturbation created by increasing the length of some packets, to expose bugs that would have not been observed otherwise.

4.1.2 Error Detection

DiAMOND's debug data collection methodology is orthogonal to the mechanism by which functional bugs are detected in the NoC. In general, and in the absence of an NoC-specific detection mechanism, functional errors in the interconnect can be detected if they cause the test or the entire system to fail. However, this failure can come millions of cycles after the bug occurrence. Moreover, it could be difficult to distinguish whether this failure is the result of a bug in the interconnect itself or in other modules in the system. As a result, deploying such a general solution along with our debug data collection mechanism might not be effective when running long test cases. The high detection latency results in large debug logs that can be prohibitive to store and analyze. As a result, we propose the use of a fine-grain detection approach that relies on adding small checkers to the NoC routers to monitor the network's execution for signs of erroneous behavior. Execution is also partitioned into epochs, and, by the end of an epoch, if no error has been flagged, then the debug data collected during that epoch is overwritten in the next one. On the other hand, if an error is detected, execution is halted and all in-flight packets are drained. The data collected during the epoch is then analyzed to debug the error and identify its source.

Chapter 2 discusses different types of localized hardware checkers that can be utilized to detect a wide range of functional errors. DiAMOND borrows these checker implementations and integrates them with its debug data collection mechanism. However, to detect deadlock errors, DiAMOND opts for the counter-based approach that is discussed in Section 2.3.3. A counter is associated with each input buffer to measure the number of cycles a packet is blocked at the router. Eventhough, there have been more accurate deadlock detection checkers proposed, we choose the counter-based approach, as it is simple and it introduces minimal area overhead. Moreover, during validation, the occurrence of false positives, a possible consequence of using simple counters, is not as critical as during the runtime operation of the design. The rate of false positives can also be decreased by configuring the time-out threshold to different values depending on the tests being run. In addition, DiAMOND makes use of the same counter to also flag starvation errors. However, deadlocks and starvation errors are distinguished by allowing the network to drain after an error is flagged. During the draining process, new packets are not injected into the network. In contrast to deadlock, as the contention for network resources reduces, starved packets will eventually acquire the resources they need to move forward and their corresponding counters reset to zero. Lastly, DiAMOND leverages these deadlock/starvation counters to provide a packet's latency timestamp, *pckt_latency*, which is logged during debug data collection (4.1.1).

4.1.3 Debug Data Analysis

Once an error has been flagged, execution is halted and the network is allowed to drain. Packets blocked, due to deadlocks or livelocks, are permitted to drain to the closest node. At this point, all the debug data that was collected during execution is residing in the content of packets, which are stored in the local caches or trace buffers across the network. This data is processed in two steps: local and global, each providing a different overview of the network's execution.

Local Processing

During this phase, the content of each local cache is individually analyzed by a software application running on the corresponding core. The data can also be loaded off-chip for a similar analysis. By examining the contents of every packet, its path through the network can be reconstructed. The path overview allows the identification of any livelock cycles, as well as any misrouted segments along its route. In the case of adaptive routing algorithms, the reconstructed paths provide insights regarding the performance and effectiveness of the routing protocol. In addition, by examining the recorded *pckt_latency* timestamps, network performance can be analyzed. Periods of high packet latency can be identified along with the routers where this high latency was recorded. Finally, by comparing a packet's requested output port and output virtual channel within a router relative to the input port and input virtual channel of the downstream router, functional bugs in switch arbitration logic can be flagged.

Global Processing

Through the local processing step, execution intervals or routers of interest are identified. Then, data from all local caches are aggregated at a central location, where another software algorithm, running on one of the cores or running off-chip, groups this data on a per router basis. Then, using the *timestampA* and *timestampD* counters, each router's data is sorted by increasing time. The sorted information basically encapsulates the series of packets and events witnessed by each router during execution. This, in turn, gives insights regarding packet interactions within routers, allowing us to reason about the source of the error observed. Since each router's *timestampA* and *timestampD* represent the value of the router's packet counter, these timestamps do not have a notion of physical time. Therefore, the arrival and departure of packets from different routers can not be correlated. However, by leveraging techniques similar to those used in ordering events for distributed systems [41],

we can still construct a partial order of events by using packets as points of reference. A packet transferred from routerA to routerB serves as a synchronization point, where events observed in routerA before the packet was sent can be classified to have happened before the events occurring in routerB after the arrival of the packet.

4.2 Experimental Evaluation

We modeled an 8x8 mesh network using the cycle-accurate Booksim simulator [24, 52], and modified it to implement the three modes of DiAMOND's data collection mechanism. Our baseline router architecture consisted of a general input-queued virtual channel router, with 5 input ports and 2 virtual channels per port. We also ran both random directed traffic, as well as network flow traces from the PARSEC benchmark suite [16]. For uniform random traffic we varied the packet size between 5 flits/packet (a header, a tail and 3 body flits) and 7 flits/packet. As for the PARSEC network flow, traffic consisted of both control packets and data packets. While data packets consisted of 5 flits, control packets were only 1-flit long.

4.2.1 Implementation and Area Overhead

To instrument network routers to collect debug data, we require some minor addition to the router architecture. First, the debug data that is collected for each packet is stored in a register, the *log_buffer*, before its written to the appropriate data flit. We require one *log_buffer* for every input buffer. The size of the *log_buffer* register depends on the network size and router architecture. In our experimental framework, we determined the length of the *log_buffers*, as shown in Table 4.1. The lengths of *timestampA* and *timestampD* (and hence the *pckt_cntr*) were chosen to be 15 bits, to ensure that the packet counter does not wrap around too frequently. In the event that a wrap-around occurs at any router, we force the epoch to end early, which permits clearing the previously collected debug data from the caches and restarting the counters. We chose a length of 10 bits for the *pckt_latency* field, limiting the maximum latency value that can be logged to 1,024 cycles. Finally, we assume a flit size of 128 bits, which is a common flit length [45]. Therefore, we are able to store the *log_buffers* collected along two hops in each flit, as explained in Section 4.1.1.

Our solution also requires adding several fields to each packet's header flit, which are listed in Table 4.2. A header flit commonly carries the router IDs of the packet's source and destination nodes. It also commonly has unused bits, which we can utilize for our solution.

log_buffer entries	number of bits
routerID	6 bits
timestampA	15 bits
timestampD	15 bits
pckt_latency	10 bits
input port	3 bits
input virtual channel	1 bit
output port requested	3 bits
output virtual channel requested	1 bit
total	**64 bits**

Table 4.1 Log_buffer.

fields	number of bits
PID	8 bits
flit_write_index	4 bits
size	3 bits
total additions	**15 bits**

Table 4.2 Additions to header flits.

Therefore, we include a small counter, which along with the source and destination, serves as an ID that can uniquely identify the packet in the network. Moreover, to keep track of the flit ID where the debug data will be stored at each router, we require an additional *flit_write_index* field. The *flit_write_index* is a counter that is incremented every two hops. It also has an extra bit, which indicates whether the debug data will replace the first or second half of the flit, as explained in Section 4.1.1. Lastly, we require the addition of a *size* field to be used in the detection of dropped and duplicated flits (Section 4.1.2). In our implementation, we chose the packet ID counter to be 8 bits, which along with the packet's source and destination node IDs forms a 20 bit unique identifier of each packet. The length of the *size* and *flit_write_index* fields is determined by the number of body flits in a packet and would typically be 3-4 bits. In our evaluation, we consider packets of size 5 and 7 flits, making the *size* field 3 bits long and the *flit_wr_index* 4 bits (including an extra bit to encode which half of the flit will be replaced by the debug data).

We implemented these additions and the detection checkers in the Verilog model of the baseline router architecture. Synthesis results show an area overhead of 2%. Moreover, the incurred power overhead is minor and is in itself not a significant concern during the emulation and post-silicon validation of the NoC. These hardware modifications are also decoupled from the router's functionality and can be disabled when the chip is released.

4.2.2 Evaluation of the Local Analysis Phase: Path Reconstruction

We examined the observability gained from utilizing our debug data collection solution by evaluating the fraction of the path that can be observed for each packet. In our validation platform, the path reconstruction process is completed during the local processing phase, where body flits of packets are examined and the sequence of routers, through which each

PARSEC network flow	drop remaining	drop at alternate hops	append
blackscholes	83.2%	96.3%	100%
bodytrack	85.0%	97.1%	100%
dedup	84.4%	96.8%	100%
ferret	84.5%	96.9%	100%
freqmine	83.8%	96.6%	100%
streamcluster	84.3%	96.8%	100%
swaptions	84.2%	96.8%	100%
vips	81.4%	95.4%	100%
x264	83.0%	96.2%	100%
average	**83.76%**	**96.54%**	**100%**
uniform traffic packet size = 5 flits	87.1%	97.8%	100%
uniform traffic packet size = 7 flits	98%	100%	100%

Table 4.3 Average path reconstruction

packet passed, is reconstructed. Table 4.3 shows the average percentage of each path that can be reconstructed under all three modes of operation. For the PARSEC network flow, data packets consist of 3 body flits and could carry complete debug data from 6 routers along their path. Therefore, under drop remaining, we are able to achieve full observability (100% path reconstruction) over packets whose path traverses 6 or fewer routers. Remaining packets have smaller path reconstruction fractions depending on their path length. For all PARSEC benchmarks, the percentage of path reconstruction is 83.76% on average. During the drop at alternate hops mode, when all body flits have been utilized, new debug data replaces older data by over-writing only the second half of each body flit, as illustrated in Figure 4.4. Moreover, routers pertaining to the alternate missing hops can be extrapolated. For the PARSEC network flow, in addition to the 6 routers that can be extracted directly from the debug data, the 3 alternate routers that were overwritten can be deduced from the recorded router IDs and input ports. Therefore, paths consisting of up to 9 routers can be fully observed. This mode provides a higher path reconstruction of 96.54%, on average. Note that, the PARSEC network flow also consists of 1-flit control packets that do not have any body flits. For such packets, we are not able to collect any debug data during these two modes. Finally, for the append mode, we achieve 100% path reconstruction for both control and data packet, as expected, since packets can append as many new flits as needed to store debug information. Similar results are also observed for uniform random traffic. Results are averaged over a sweeping injection rate from low injection, 0.04 flits/cycle/node, to high injection, 0.24 flits/cycle/node.

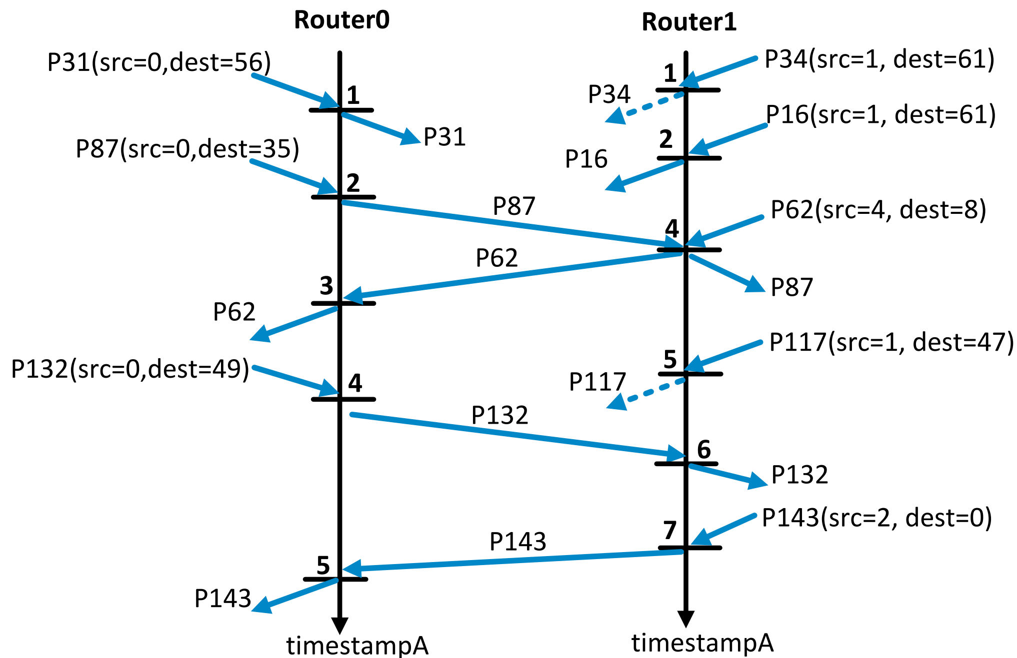

Figure 4.5 Example of reconstruction of packet interactions. TimestampA values are used to construct the sequence of events observed in each router. For example, the packet with ID 31 (P31) is received and sent from router 0 at timestampA=1. Dashed lines indicate packets sent from router0 or router1 to other routers in the network.

4.2.3 Evaluation of the Global Analysis Phase: Case Study

Packet interactions. Our solution also allows examining packet interactions within routers, as well as constructing a partial overview of global network behavior. During the global analysis phase, debug data from all nodes is aggregated and grouped per router. Each router's data is then sorted by increasing *timestampA* values, allowing us to order packet arrival and departure to and from each router. As an example, we ran uniform traffic over an 8x8 mesh at an injection rate of 0.19 flits/node/cycle. Figure 4.5 shows the packets traversing router0 and router1 in the first 100 cycles of the simulation, as obtained from our solution. Note that since each router's *pckt_cntr* operates independently, the *timestampA* values are not synchronized across routers, preventing the establishment of a complete global order of events across the network. However, by leveraging common packets as points of reference, we can establish a partial order. For example, in Figure 4.5, packet 132 is a common packet between routers 0 and 1. Based on that, events relating to packets 31, 87 and 62 in router0 (*i.e.*, the events that occurred before sending packet 132) happened before events associated with packet 143 in router1.

Latency at internal routers. Typically, during the performance validation of NoCs, the debug information that can be collected relies on end-to-end analysis of latencies and

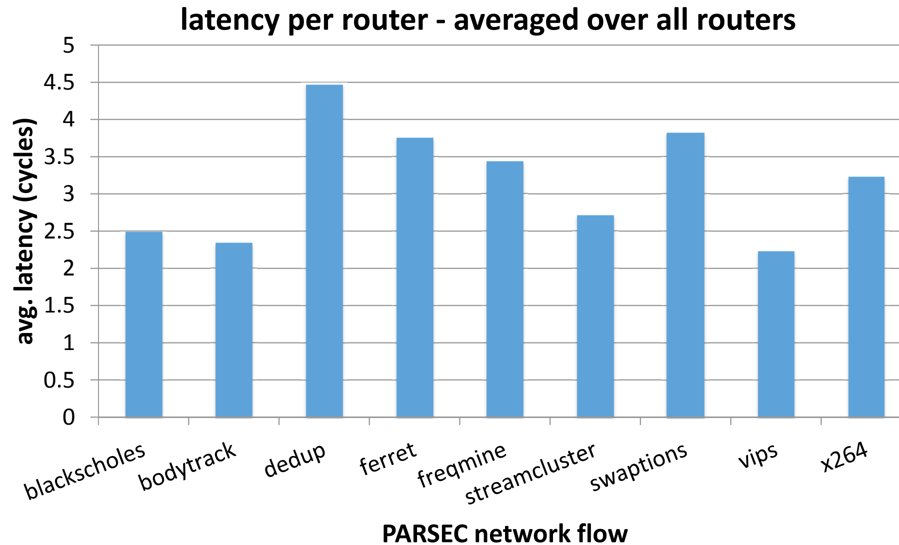

Figure 4.6 Average packet latency at internal routers. Results are shown for the PARSEC network flow, averaged over all routers

throughput. However, our solution has the benefit of providing performance statistics at internal routers, since debug data is collected at every hop along a packet's path. By examining the *pckt_latency* field recorded in the debug data, we can study the average and maximum packet latencies observed at every router and throughout the network's execution. This information facilitates debugging performance bugs, where the execution is functionally correct but does not meet the performance specifications of the design, such as in the case of starvation errors. As an example of the type of results that can be generated, Figure 4.6 shows packet latencies, averaged over all routers for the various PARSEC benchmarks. Some benchmarks, such as *dedup* and *ferret*, exhibit larger average packet latencies within routers, as compared to others.

We can also plot average packet latencies observed within each router during a specific testbench execution. For example, Figure 4.7 shows this information for the *dedup* benchmark, where we can observe that router 49 exhibits the highest average packet latency compared to other routers. Using such results, we can identify potential performance bottlenecks in the network. It is also possible to use our scheme to examine packet latency values over time and per router. For example, Figure 4.8 shows the variation in packet latencies observed at router 49 throughout the execution of *dedup*. Based on that, execution periods of interest can be identified for further analysis, such as the period highlighted in Figure 4.8, where we record the first significant increase in latency. By examining packet interactions and path reconstruction results for the *dedup* benchmark, we identify the packet associated with this latency and find that it is blocked in router 49 due to congestion in the downstream router 41. Router 41, in turn, has several packets that are also waiting for busy

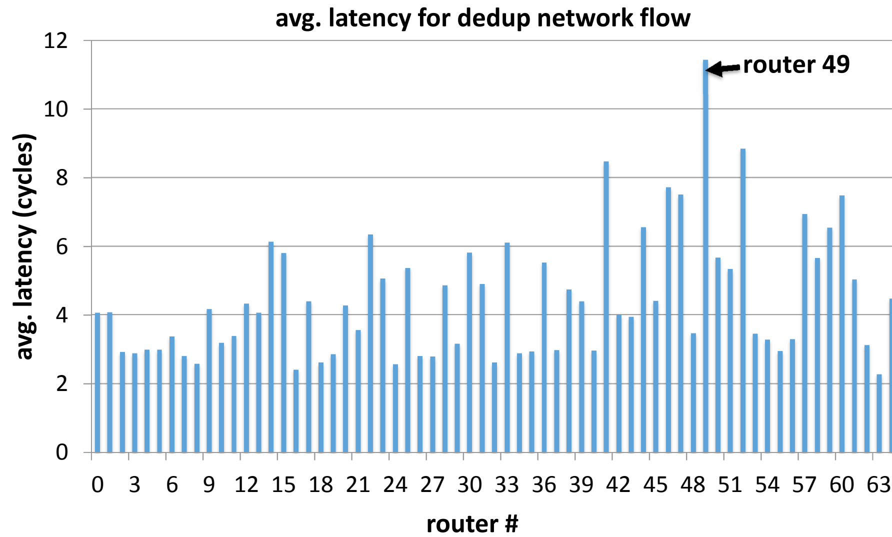

Figure 4.7 Average packet latency for a sample benchmark. For the dedup network flow, we show the average packet latency observed at each router.

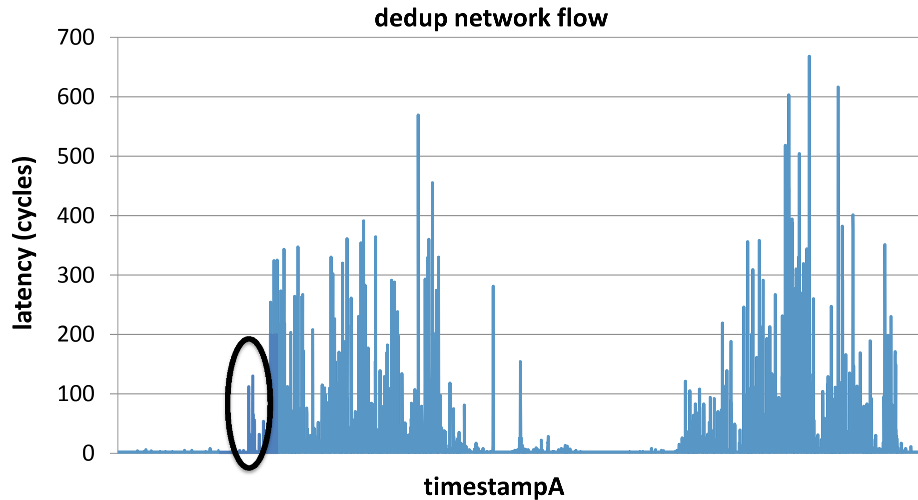

Figure 4.8 Packet latency at router 49 for dedup benchmark.

virtual channels. By means of inspection, we realized that the simulated router design was setup to utilize a basic credit-flow mechanism that releases the output virtual channel only after the entire packet is transferred, which amplifies the packet latencies in the presence of congestion. This example highlights how DiAMOND provides high quality diagnosis and traffic-inspection capabilities in post-silicon or emulation environments, inculding the ability to investigate performance flaws in the network.

4.3 Summary

In this chapter, we desribe a packet monitoring solution to boost observability of a network's internal operations during verification on emulation and post-silicon platforms. Debug information is collected for every packet, at each router along its path, and then systematically written in one of its body flits. In addition, simple hardware checkers are added to routers to monitor execution and flag functional bugs. Upon bug detection, the collected debug data provides increased observability of network traffic. The analysis process reconstructs the packets' paths, achieving, in most cases, over 80% reconstruction. The collected debug dat also provides several functional, as well as performance, statistics regarding the network's operation, including the sequence of events that occurred within routers and packet latencies observed per router and over time. Although our packet monitoring solution succeeds in tracking the majority of packets in the network, it alters the data contents of packets, making it only suitable for verifiying the network under synthetic traffic workloads or application traces. Nevertheless, directed random testing is both a common and crucial aspect of functional verification that allows exercising and validating a wide range of a design's operation, and thus applying our packet monitoring approach to these verification frameworks can significantly improve error detection and debugging efforts.

Chapter 5
Router Monitoring

In Chapter 4, we introduced and evaluated a novel packet monitoring solution that collects debug data for each packet at every hop along its path and stores the collected data by incrementally replacing the packet's data contents. As such, this approach is best-suited for tests consisting of randomly generating traffic patterns or application-traffic traces. However, complete functional verification of the NoC also requires running full applications to evaluate the correctness of the system under real-world operating conditions. In such situations, altering the data contents of in-flight traffic is no longer possible, as it affects the application's execution or prevents it from running all-together. Therefore, when running application-based tests, our packet monitoring solution is not applicable and a different methodology is needed to provide observability. In this chapter, we discuss router monitoring as an alternative solution to this problem, implemented through a novel solution, NoCDebug [4, 6]. In NoCDebug, routers are instrumented to periodically take snapshots of the events they are encountering at the time. These snapshots are then accumulated to provide an activity log of each router's execution, greatly improving observability into each router's operations as well as the overall network execution.

When using NoCDebug, a test's execution is partitioned into a series of epochs, each comprising a logging phase and a checking phase. During the logging phase, routers take snapshots of their internal state. These snapshots are taken at regular intervals and stored in the verification framework's designated storage units. When the available space is exhausted, the logging phase terminates and the network execution is stopped. Then, the snapshots collected at each router are analyzed to detect the occurrence of communication errors, and we refer to this phase as a local check phase. If an error is suspected, then the system initiates the global check phase to help determine the cause of the error. The global check phase aggregates the snapshot logs of all routers, and then it provides an overview of the network traffic at the time of the bug occurrence and the reconstruction of the paths followed by packets in flight.

In this chapter, we describe NoCDebug's approach while considering a baseline chip-

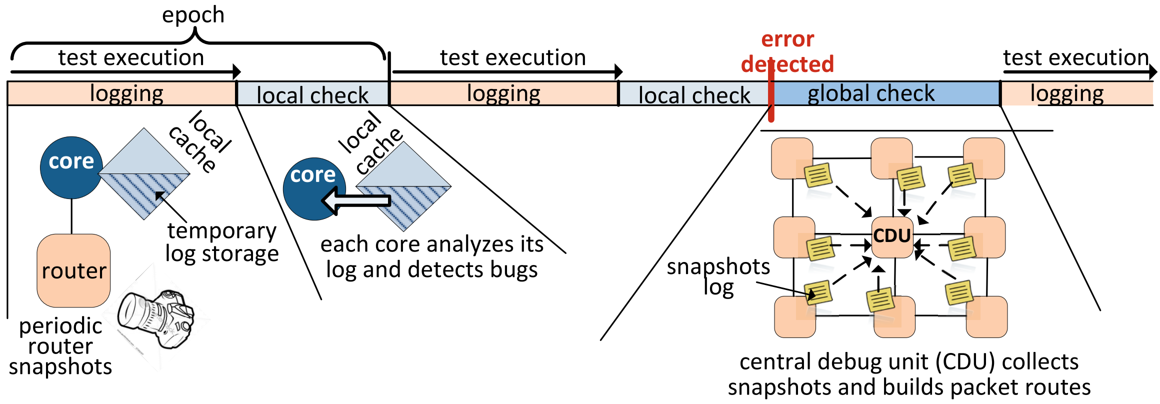

Figure 5.1 Execution flow of the NoC debug platform. Execution is partitioned into epochs, each consisting of a logging phase and a local check phase. During the logging phase, snapshots of each router's contents are periodically taken and logged in a reserved portion of the local cache. During the local check phase, each local log is analyzed by a software algorithm running on its corresponding processor core. If an error is detected, the global check phase collects the snapshot logs from all caches and reconstructs the route of packets that were in-flight during the logging phase.

multiprocessor design. Therefore, we propose to store the snapshots collected at each router in a designated portion of the local L2 cache corresponding to that CMP node. This space is temporarily reserved for verification purposes, but is released afterwards. Additionally, once the reserved cache space is full, the logs are analyzed through a series of checks implemented in software and running on the CMP's cores. Figure 5.1 illustrates this flow. However, our solution can also be adapted to designs other than CMPs and to different verification frameworks. The snapshots collected at each router can be stored in on-chip traces buffers and periodically transferred to the host computersthat interface with the verification platform. Similarly, the analysis process can be run on these hosts as well.

5.1 Logging

5.1.1 Logging in the Routers

During the logging phase, each router is instrumented to take snapshots of packets traversing it at the time, as illustrated in Figure 5.2. Snapshots are taken periodically at fixed time intervals. This rate is set by the user, based on the characteristics of the NoC and the traffic density. To take a snapshot, the physical clock of the router is used to track time. To identify the packets traversing the router at the time the snapshot is to be taken, a router looks for packet header flits stored in its input buffers. This is accomplished by

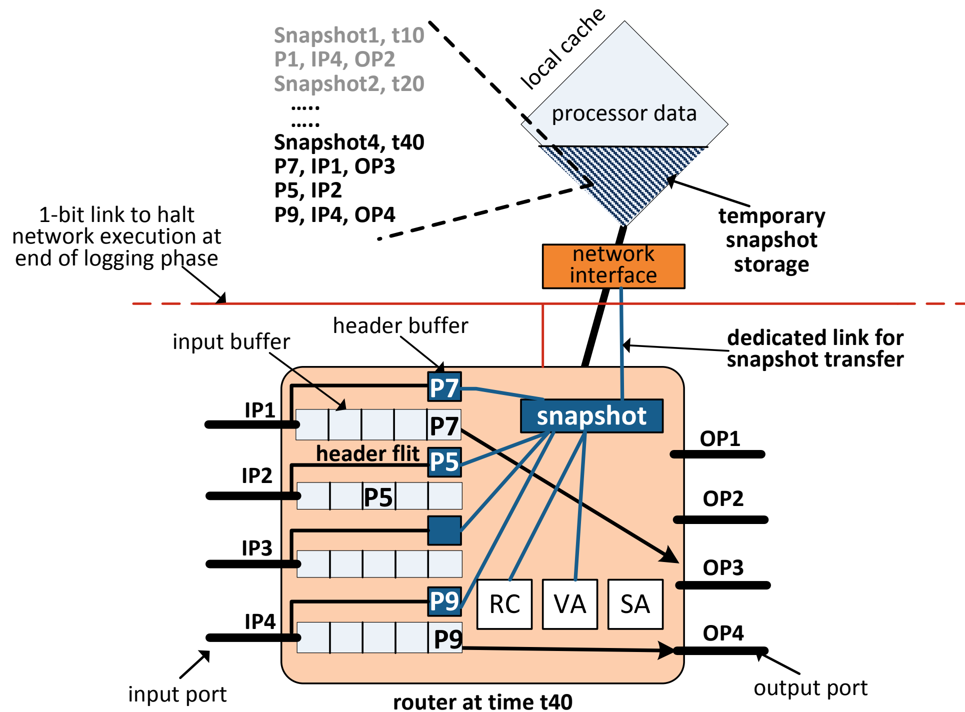

Figure 5.2 Logging. Routers periodically take snapshots of packets traversing them. A header buffer is added for every input buffer to keep track of the header flits of packets stored in the input buffer. The snapshot hardware captures the header data as well as information from the route computation (RC) and virtual channel allocation (VA) modules of the router.

augmenting the router with one *header buffer* for each input buffer. When a router receives a packet, it stores the packet in its input buffer and stores a copy of the header flit in the corresponding header buffer. Note that the size of the header buffer depends on how many packets can be in a router's input buffer at any point in time, which in turn depends on the minimum number of flits in a packet. When a header flit is identified, the snapshot hardware can then obtain its source and destination nodes (usually found in the header). In addition, we require that the header flit also includes a packet ID (unused space in the header is often available). This ID is provided by a sequential number generated by the source node, which, along with the source and destination, forms a unique identifier of that packet. It is also useful to log additional information about the packets, depending on their status within the router. For example, packets that have completed the route computation phase are assigned an output port, so logging the output port allows us to determine the next router on the path.

In addition to packet-specific information, being able to trace these packets over time is important for debugging. Thus, a snapshot also stores the physical time at which it was taken. However, if the network uses multiple clock domains, we also rely on the notion

of logical time implemented through lamport clocks[41], where every router has a logical clock that is advanced when a new packet is received. Packet headers also include a timestamp, which monotonically increases with every hop. When a router receives packets, it sets its logical clock to the maximum timestamp of the received packets and then increments it. When a packet leaves the router, its timestamp is updated to the logical time of the router. Thus, in the case of multiple clock domains, a snapshot entry also includes the logical timestamp of the packet. Note, snapshot entries that are part of the same snapshot have the same physical timestamp, we therefore do not replicate this information and store it only once for the entire snapshot.

Overall, every snapshot consists of several entries, one for each packet. Every entry contains a packet ID (counter, source, destination), its input port, input virtual channel, output port (if allocated), output virtual channel (if allocated), and its logical timestamp. Figure 5.3 shows the information logged in each snapshot.

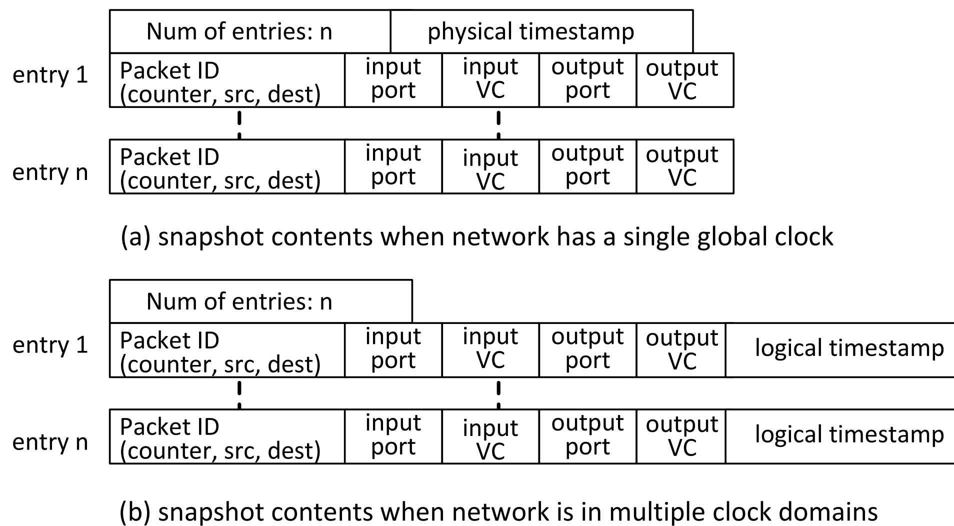

Figure 5.3 Information logged in each snapshot. A snapshot consists of several entries, one for each packet in the router. Each entry includes the packet ID, the input port and input virtual channel where the packet is queued, the output port and output virtual channel if they are allocated, and the packet's logical timestamp if the network has multiple clock domains.

5.1.2 Log Storage in Local Caches of a CMP Design

In a CMP design, when a snapshot is captured in the router, it is transferred through a dedicated link to the network interface and then to the local L2 cache of that node, as shown in Figure 5.2. Therefore, the local L2 cache, which is typically set-associative, designates

one or more ways to be used solely for the storage of snapshot data. This effectively disables these set entries from the perspective of the processor and the cache allocation and replacement policies.

From the perspective of the snapshot data, even though the reserved cache lines belong to different sets, they can be collectively viewed as one big FIFO. Snapshots are written and read from each line in order of increasing cache index. Depending on the size of the snapshot relative to the cache line size, a snapshot could be padded to fill the line or it could span multiple cache lines. Since every snapshot stores the number of entries it contains, and given that the entry size is constant, determining the size of each snapshot is straightforward and it is needed for addressing the cache, reading the snapshots, and flagging a full log. In order to write/read from the snapshot log, an address-generating unit generates the cache index to address the line in the next set. In addition, a running counter can keep track of the total size of snapshots logged so far. When this counter exceeds the user-sepecified log size, a local check is triggered. Similarly, snapshots are read from one line at a time. Having the number of snapshot entries and knowing the fixed snapshot entry size, allows proper parsing of the data and the snapshot boundaries.

In our experimental evaluation of NoCDebug, detailed in Section 5.3, the average size of snapshots for bitcomp traffic at varying injection rates ranges between 9 to 12 bytes. Similarly, the average snapshot size for the Parsec benchmarks is approximately 9 bytes. Considering, as an example, a 256KB L2 cache that is 4-way set associative with 32B lines, then we can configure one of those ways to be used to store only the incoming snapshots, which is equivalent to 64KB. However, in practice, it maybe useful to trigger a local check before all the physically available storage space has been used, so that the bug is detected closer to its ocurrence and debug data is more relevant. In our experiments, we set the maximum log size to be 30KB. Therefore, we rely on the running counter to keep track of the total size stored so far and to trigger a local check when it exceeds the preset limit. Finally, note that if snapshots are very large and span several lines, then it is possible to exhaust the entire reserved space before the maximum log size is reached. In this case, the local check can be triggered earlier or the execution can be repeated while reserving more ways for snapshot data.

5.2 Error Detection and Debugging

The snapshot data collected by NoCDebug provides detailed observability of the network's execution, and thus it primarily targets debugging purposes. In terms of detecting the oc-

currence of a functional bug, NoCDebug's debug data collection technique can be used in conjunction with any error detection approach. In chapter 2, we surveyed different error detection approaches, end-to-end checking, hardware localized checkers, and software-based localized checkers, and we noted the trade-offs of each. It is possible to utilize NoCDebug's data collection along-side localized hardware checkers. In this case, the hardware checkers monitor execution and flag errors on-the-fly, which eliminates the need for performing the local check phase in NoCDebug's verification flow (5.1) In such a setup, by the end of an epoch, if none of the hardware checkers had flagged an error, the debug data that was collected is simply over-written in the following epoch. Whereas, if an error is detected, the system directly enters the global check phase, where the router logs are aggregated and analyzed. However, since NoCDebug collects detailed execution logs for each router, it lends itself to the use of software-based checkers to detect communication errors. In this case, execution must be periodically halted to analyze the logs and perform the checking. In our solution, this process is initiated when the designated log storage is exhausted, and we refer to it as the local check phase. Moreover, instead of aggregating all the logs and then checking them for correctness, our local check phase is designed to detect errors by independently examining each router's log. This is intended to delay the time-consuming process of aggregating logs until an error is suspected to have occured. For example, in the context of a CMP design, the local check phase can run on the processing cores, directely accessing the local snapshot log stored in each node's cache, without the need to transfer the logs from all other nodes in the system.

NoCDebug's local check phase is designed to identify the following communication errors: deadlocks, livelocks, misroutes and starvations. As for the other types of errors discussed in Section 2.3.3, such as packet corruptions and dropped/duplicated packets, these errors are more efficiently detected using a simple hardware or software checker, and thus we do not consider them in our local check phase.

5.2.1 Local Checks

Livelock. Since the local check algorithm only has access to the local snapshot log of each router, the occurrence of a network livelock must be detected locally, without considering the logs of other routers. For a network with a finite number of nodes, a livelocked packet will eventually traverse the same router twice. Provided that the epoch length is long enough for the livelock cycle to form, such errors can be detected locally. In Figure 5.4 (lines 3-7), the algorithm retrieves the physical timestamp of the packet's snapshot entries. If the difference in time between two successive snapshot entries is greater than

the snapshot interval, then they were captured in non-consecutive snapshots. This is an indication that the packet traversed the router at different non-consecutive times and the algorithm flags a livelock error.

Starvation. A starvation error exists if a packet is temporarily blocked waiting to acquire resources that are given to other packets. Packets traversing the network can only be blocked in a router's input buffers, as this is the only storage available in the network. Therefore, a starved packet must appear in several consecutive snapshots in a router. Hence, the checking algorithm determines the number of consecutive snapshots in which each packet appears (Figure 5.4, line 9), and based on the snapshot rate, it deduces how long the packet has been waiting and flags a starvation error when this value exceeds a user-set threshold.

Deadlock. At the network-level, a deadlock can be identified by the existence of a cyclic dependency of resources. However, identifying a deadlock at the router-level by examining only the local snapshot log reduces to the problem of identifying a blocked packet. Similar to the starvation bug, a packet is blocked if it appears in several consecutive snapshots. However, a deadlocked packet is permanently blocked, which means it must also be seen in the latest snapshot of the epoch. Therefore, the local check algorithm checks if any of the packets in the snapshots satisfy both these conditions and flags them as deadlocked (lines 10-11 in Figure 5.4). In the case that a starved packet happened to still be in the router at the time the last snapshot was taken, it is possible to misclassify the starvation error as a deadlock.

Misroute. In a network employing a deterministic routing protocol, all valid paths between each source-destination pair are known, since they can easily be collected theoretically or experimentally beforehand. To detect misroute errors during a test's executiom, this information is stored in each local cache in the form of a bit vector that indicates, for each src-dest pair, whether the corresponding router belongs to the valid path. Then, the local check algorithm sumply iterates through the snapshot entries, obtains the source and destination of each entry, and checks it against the valid paths information (lines 13-16 in Figure 5.4).

In the absence of errors, the snapshot data is cleared and the NoC resumes execution. However, if an error is detected, the logs are aggregated at the central debug unit (CDU), which can be any of the network nodes. In-flight packet are dropped and the logs are sent from one cache at a time to the CDU. Sending from one node at a time reduces the complexity of network operations during the transmission of the logs, which increase our confidence that its error free.

Sampling. In order to reduce the time spent in the local check phase, we also provide an

optimization that trades-off the ability to detect errors with the execution time of the local check algorithm. Instead of analyzing the entire snapshot log, each core can downsample the information within its local log, such that it only looks at uniformly distributed fraction of the snapshot entries. This *sampling rate* is another user-defined value, and we evaluate its trade-offs in Section 5.3.1.

```
1. LocalCheck (snapshotLog){
2.  foreach packet in snapshotLog {

3.   foreach ntr in GetSnapshotEntries(packet) {
4.    time = PhysicalTimeEntry(ntr);
5.    next_time = PhysicalTimeEntry(ntr+1)
6.    if (next_time - time > snapshotInterval)
7.     FlagError(Livelock)
8.   }

9.   if (CountEntries(packet) > threshold)
10.     if packet in lastSnapshot(snapshotLog)
11.       FlagError(Deadlock)
12.     else FlagError(Starvation)

13.   src = GetSrc(packet)
14.   dest = GetDest(packet)
15.   if router !InPath(src, dest)
16.    FlagError(Misroute)
17. } }
```

Figure 5.4 Local check algorithm. Each router examines it snapshots log for packets exhibiting irregular behavior, which signifies the presence of an error. To detect livelock, the algorithm checks if a packet appears in non-consecutive snapshots. To detect blocked packets (deadlock and starvation), it checks if a packet appears in several consecutive snapshots, whose number exceeds a user-defined threshold. Finally, it checks for possible misroute errors by determining if it the router belongs to the set of valid paths between the packet's source and destination.

5.2.2 Global Checks

The goal of the global check phase is to provide useful information that can facilitate debugging the detected error. It combines the collected snapshots to reconstruct the paths of observed packets, and it gives an overview of the traffic that passed through the network during the logging phase. Snapshots entries pertaining to the same packet are grouped together. Packet routes are then reconstructed by sorting these entries in increasing order of

the snapshot's physical timestamp, when a global clock is present, or the packet's logical timestamp, when the network uses multiple clock domains. In the latter case, because the logical timestamp of a packet is monotonically incremented in every hop, sorting according to increasing timestamp values allows the reconstruction of the path in the correct order. We also use the input port and output port fields of each snapshot entry to try to reconstruct the path beyond the router in which the packet was observed. We determine the upstream (downstream) router, based on the input port (output port) field and the network topology.

Besides reconstructing packet routes, the global check algorithm highlights the packets that were present in each router at the time the snapshot was taken, which exposes the interactions within the router that could have contributed to the error. For example, by examining a router's snapshot log, we can determine a subset of the packets that traversed it and based on that deduce the router's internal states, such as the buffers that were in-use and the virtual channel and output port allocations at the time.

5.3 Experimental Evaluation

To evaluate our debug platform, we modeled a CMP interconnect with Booksim. We considered our baseline system to be an 8x8 mesh NoC with input-queued virtual channel routers. Each router has 5 ports, 2 virtual channels and 8 flit-buffers. We modified the simulator to instrument routers to periodically take snapshots of packets traversing them. We also implemented the local check functions to analyze the collected snapshots. We evaluated the system by simulating two types of workloads: directed random traffic (uniform, bitcomp) and applications from the PARSEC benchmark suite [16].

	snapshot interval=10 cycles			snapshot interval=50 cycles		
injected bugs	**no sampling**	**50% sampling**	**20% sampling**	**no sampling**	**50% sampling**	**20% sampling**
misroute	19%	14%	2%	6%	0%	0%
deadlock	100%	100%	100%	100%	100%	100%
livelock	100%	100%	100%	100%	100%	100%
starvation	24%	7%	2%	0%	0%	0%

Figure 5.5 Error detection rate for different types of bugs that were injected into the baseline system. The results are reported for two snapshot intervals, with and without local check sampling.

		snapshot rate 10 cycles		snapshot rate 50 cycles	
low injection	%packets observed	54%		20%	
	avg. path reconstruction	53%		30%	
	avg. path reconstruction rate of faulty packets per bug type	misroute-1	43%	misroute-1	0%
		deadlock	63%	deadlock	54%
		livelock1	74%	livelock1	66%
		starvation	36%	starvation	0%
		livelock2	29%	livelock2	6%
		misroute-3	56%	misroute-3	0%
		misroute-9	58%	misroute-9	0%
medium injection	% of packets observed	67%		28%	
	avg. path reconstruction	52%		31%	
	avg. path reconstruction rate of faulty packets per bug type	misroute-1	55%	misroute-1	0%
		deadlock	67%	deadlock	49%
		livelock1	57%	livelock1	48%
		starvation	47%	starvation	0%
		livelock2	37%	livelock2	9%
		misroute-3	59%	misroute-3	22%
		misroute-9	64%	misroute-9	17%
high injection	%packets observed	85%		50%	
	avg. path reconstruction	58%		35%	
	avg. path reconstruction rate of faulty packets per bug type	misroute-1	60%	misroute-1	0%
		deadlock	77%	deadlock	54%
		livelock1	73%	livelock1	56%
		starvation	0%	starvation	0%
		livelock2	47%	livelock2	18%
		misroute-3	40%	misroute-3	23%
		misroute-9	67%	misroute-9	11%

Figure 5.6 Error diagnosis. We quantified our solution's ability to diagnose errors by measuring the percentage of packets observed, the average percentage of each path we were able to reconstruct, and the average percentage of reconstruction for the paths followed by the erroneous packets.

5.3.1 Error Detection

We first analyzed our platform's ability to detect functional errors. We modeled four types of bugs in the baseline system, each representing an error that would prevent the network from making correct forward progress. These include a deadlock and a livelock bug, a misrouting bug, where a packet is misrouted once along its path to the destination node, and a starvation bug, where a packet is temporarily prevented from acquiring the resources it needs to progress along its path. We ran both the random traffic and PARSEC workloads, while triggering each bug once during the simulation and repeated each experiment with 11 random seeds for statistical confidence.

Figure 5.5 shows the detection rate when simulating bitcomp traffic for our four bugs

and two snapshot intervals (every 10 cycles and every 50 cycles). In addition, we varied the sampling rate of the local check algorithm, which, as explained in Section 5.2.1, constitutes a trade-off between the ability to detect errors and the time it takes to complete the local check phase. In these experiments, the threshold for detecting starvation and deadlock was set to 100 snapshots (refer to Section 5.2.1). Results show that deadlock and livelock bugs are always detected, whereas misroute and starvation have a much lower detection rate (0% -24%). This is because, when a packet is deadlocked or livelocked, it remains in this state from when the bug manifests until the end of the simulation, which increases its probability of being captured by the snapshots. On the other hand, misroute and starvation errors are transient and the affected packets can be missed and never observed. This effect is more pronounced when the snapshot interval is increased to 50 cycles, because snapshots are now taken less frequently. In addition, if sampling is activated during the local check phase, the detection of misroute and starvation decreases, again because of the transient nature of these errors. Whereas, local check sampling does not affect the detection of livelock and deadlock. Finally, we noticed that for the snapshot interval of 10 cycles, the simulations where the traffic injection rate was high (close to network saturation), exhibited false positives due to the false detection of starvation bugs. Starvation errors were flagged even before our bugs were injected. This is because the network is highly congested and the chosen detection threshold was small. However, at a snapshot interval of 50 cycles, no false positives were reported.

5.3.2 Error Diagnosis

We also evaluated the quality of information that can be obtained from the snapshots when they are aggregated during the global check phase. Figure 5.6 highlights the results for bitcomp random traffic at low, medium and high injection rates and two snapshot intervals (10 cycles and 50 cycles) and with a sampling rate of 50%. We particularly looked at 3 measurements. First, we calculated the percentage of packets that were observed in at least one snapshot out of the total number of packets injected in the simulation. Second, we looked at path reconstruction, which is the average percentage of each route we were able reconstruct from the aggregated snapshots. Finally, we measured the path reconstruction of the packets that were detected as faulty (*ie.*the packets where the detected error manifested).

For a snapshot interval of 10 cycles, the percentage of observed packets is 54% at low injection and increases to 85% at high injection. This increase is due to the fact that at higher injection rates, the network is more congested and routers have more packets traversing them, which allows each snapshot to capture a larger fraction of the packets in

flight. As for path reconstruction, we note that on average 52% to 58% of each route was reconstructed from the snapshots. When looking at the path reconstruction of the erroneous packets, we notice that when the bug is detected, we can reconstruct on average 36%- 60% of its path. For the starvation bug at high injection, the path reconstruction of the faulty packet is reported as 0%, since the error was not detected in any of the runs.

When the snapshot interval is increased to 50 cycles, the percentage of packets that are observed in the collected snapshots decreases to 20% at low injection and 50% at high injection. Similarly, the overall average path reconstruction decreases to 35%. With higher snapshot intervals, snapshots are taken less frequently and thus more packets are missed. Therefore, the snapshot interval directly influences network observability. Moreover, the impact of the chosen snapshot interval varies depending on the injection rate. For test cases that have low traffic injection, a smaller snapshot interval is required to observe at least 50% of all packets injected. However, with high injection rate a larger snapshot value would be sufficient to achieve the same result, because of the higher congestion in the network.

5.3.3 Performance Evaluation

Our NoC debug platform periodically stops network execution to locally check the collected snapshot logs. The time spent in the local check phase is therefore the main source of performance overhead. To evaluate this overhead, we implemented and integrated our snapshot and local check algorithms with the Booksim simulator. Note that, since the local checks are intended to run in software on the CMP's cores, our implementation of these algorithms could not be cycle-accurate. Therefore, we instrumented the local check algorithms to utilize the x86 timestamp counters and report the execution time in cycles while running on a 2.4GHz Core2 Quad machine. We then added the execution time (in cycles) of the local checks occurring in each run to the benchmark execution time (in cycles) that was obtained from the cycle-accurate Booksim simulator. We compared these results to the benchmark execution time on the baseline system, which does not have any of our debug functionalities. For our experiments, we simulated both PARSEC and random traffic benchmarks. We also assumed that 30KB of the local L2 cache were reserved for the snapshot log, when simulating random traffic workloads, and 10KB when simulating the PARSEC benchmarks. Note that the snapshot storage is approximately 10% or less of a typical L2 cache size of 256KB. In addition, the lower injection rate of the PARSEC benchmarks forced us to choose a lower storage size so that the local log could fill up and trigger a local check at least once during the benchmark's execution.

We first examined the performance **impact of varying snapshot interval values**. Fig-

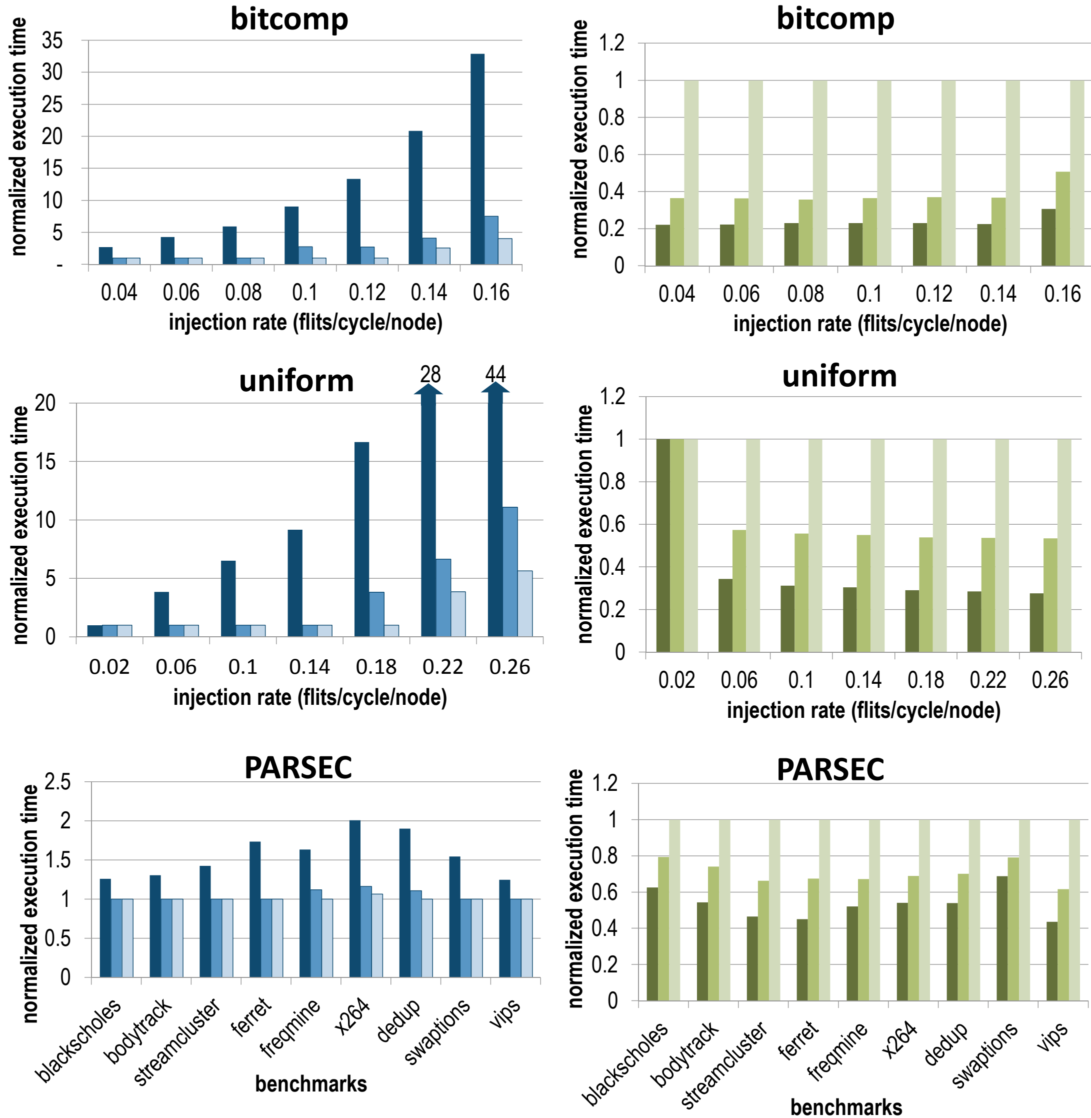

Figure 5.7 Normalized execution time for PARSEC and random traffic workloads with varying snapshot intervals and local check sampling rates.

ure 5.7a) shows the the execution time of the different workloads normalized to their respective execution times on the baseline system (without any of our NoC debug functionalities). In these experiments, the local check sampling rate was set to 20%. The normalized execution time only takes into account the overhead of the periodic local checks triggered during each run, as this is the main source of overhead in our framework. For ran-

dom traffic, we varied the injection rate from low injection (when the network is close to zero-load latency) to high injection (before network saturation). With larger snapshot intervals, local checks are invoked less frequently and the overall benchmark execution times are smaller. This can be observed in Figure 5.7a), where larger snapshot intervals result in smaller values for normalized execution relative to the baseline system. For example, when the snapshot interval is set to 10 cycles, the execution time for bitcomp traffic ranges from 2x to 32x slower than a baseline system that does not implement our solution. When increasing the snapshot intervals to 50 and a 100 cycles, the execution time improves by a factor of 5 and 10, respectively. Note that despite this slowdown, the speed at which benchmarks are executed is still relatively high, especially when compared to the speeds of simulations during pre-silicon verification, which are typically in the order of 10-100Hz. However, as explained in Section 5.3.2, a larger snapshot interval reduces the observability of in-flight packets and the ability to reconstruct their paths. A similar trend is observed for the uniform and PARSEC workloads. PARSEC workloads experience on average 1-2x slowdown when executing on our framework with a snapshot interval of 10 cycles. This overhead consists of the additional time spent on locally checking the logs whenever they are full. However, note that at low injection rates, increasing the snapshot interval might not be feasible, as was the case of the PARSEC benchmarks that would never trigger a local check.

We also evaluated the **effect of varying the local check sampling rate** on performance, in Figure 5.7b). In these experiments, we are using a snapshot interval of 10 cycles while varying the local check sampling rate between 20% and 50%. The results are normalized to a system without any local check sampling. As expected, sampling the logs during the local check phase speeds up the process. For example, at a sampling rate of 50%, where only half of each snapshot log is analyzed, the execution time is twice as fast, on average, for the random traffic. An even lower sampling rate of 20% improves the execution time to at least a factor of 3. A similar trend is observed for the PARSEC benchmarks.

5.3.4 Area Overhead

We also evaluated the area overhead of the router modifications that are needed to capture the local snapshots. The router additions are described in section 5.1 and illustrated in figure 5.2. We synthesized the modified baseline router using Design Compiler [64] with Artisan 45nm target library, and the area overhead was found to be 9%.

5.4 Summary

This chapter presented a router monitoring solution, NoCDebug, that relies on taking periodic snapshots of each router's execution. Once the space allocated for storing these snapshot logs is exhausted, execution is halted and each log is independentely analyzed by a software alogrithm running on the design's processing cores or off-chip on a host machine. This analysis is used to detected signs of erroneous network behavior, such as deadlocks, livelocks, starvation and misrouting errors. Once an error is suspected, the logs are combined and additional debug information is extracted. This debug data includes an overview of the network traffic at the time surrounding the manifestation of the error, as well as a partial reconstruction of the paths followed by packets. Results show that our approach can effectively collect information critical to the detection and diagnosis of functional errors during verification. Moreover, our router monitoring approach does not introduce any perturbation to the in-flight traffic, making it well-suited for running randomly generated traffic patterns as well as real-world application traffic.

Chapter 6

Runtime Correctness for NoC Interconnects

Due to the limitations of design-time verification efforts, the potential of design bugs escaping into the interconnect of a released product is a concern. As discussed in Chapter 2, software-based simulations are limited by their low speeds and cannot exhuastively validate large and compex designs. On the other hand, emulation and post-silicon validation platforms introduce numerous verification challenges that hinder the effective detection and debugging of errors. The end result is that design-time verification is only successful in verifiying a fraction of the interconnect's executions, including its most common functionalities and features. Fortunately, this is often sufficient to discover the majority of functional bugs in the design. However, there always remains a subset of executions that is never verified and that could be hiding design bugs. When these previously undiscovered bugs manifest at runtime, in the end-user product, they can compromise the correctness of the interconnect and the overall system. This chapter explores runtime verification solutions that equip the interconnect with mechanisms to detect the manifestation of design bugs and recover from them.

6.1 Background and Related Work

Faced with the growing complexity of hardware designs and the inherent incompleteness of current verification techniques, the threat of design bugs escaping into released products has fueled the development of numerous runtime verification approaches. Although the fraction of bugs that escape verification is typically small, these bugs can cause significant damages to producing companies, which may be forced to issue product recalls or delay a product's release. Traditionally, many hardware vendors publish errata documents or specification updates revealing a list of bugs discovered after the product's release. A unique characteristic of the majority of escaped design bugs is that they are bugs that manifest

rarely, which is precisely why they were not discovered during development-time. Moreover, these bugs often occur as a result of an unanticipated sequence of events or a set of interactions between design components that was never exercised and validated during the design's development. Once bugs are discovered after a product's release into the market, a common approach to address them is through firmware updates to turn-off or work-around the problematic features or through work-around instructions to the end-user. For example, a bug discovered in AMD's phenom processor forced a look-aside buffer in the design to be disabled to prevent the error from causing the system to hang [40]. However, the consequence was a 10% drop in performance and a notable financial impact to the manufacturer [72]. On the other hand, some bugs cannot be fixed, and depending on their severity, they may require the product to be recalled [47]. In addition, design companies as well as researchers in academia have explored and developed various other runtime verification approaches. Some solutions have focused on ensuring the runtime correctness of processor cores through patching-based techniques that provide programmable mechanisms to side-step the occurrence of bugs at runtime [70, 53, 73]. Whereas, [14] proposes deploying a simple in-order checker core along-side the complex processor core that is prone to design bugs. The checker core is responsible for validating the computation results of the complex core by recomputing the same operations. If the checker core detects a mismatch, its correct results are fed back to the complex core allowing the design to bypass and recover from the error. There have also been several runtime solutions targeting multi-core designs, cache coherence protocols and memory consistency [20, 63, 50, 71].

In the context of NoC runtime correctness, very few solutions exit to address escaped design bugs, with the majority focusing on correctness in the face of transient or permanent faults in the NoC hardware. [54] is one technique specifically targeting escaped design bugs in NoCs. In this work, the authors rely on fromal verification to validate the correctness of router-level operations along-side a runtime solution to ensure network-level communication correctness. On the other hand, source-based acknowledgment-retransmission mechanisms, [51], are among the fault-tolerance schemes that are most relevant to the work presented in this thesis. In source-based retransmission solutions, an error detection code is added to packets at the source node and it is checked at a packet's final destination node. A detected error is resolved by retransmitting another copy of the erroneous packet. One main drawback of employing such a technique is that it requires a large number of retransmission buffers to be reserved at each node to store copies of in-flight packets. Otherwise, the lack of free retransmission buffers stalls packet injection and can lead to significant reductions in network throughput. In this chapter, we present two novel approaches to detect and recover from communication errors in a NoC interconnect. In both solutions, we

avoid storing a copy of every packet injected into network, thus eliminating or reducing the number of retransmission buffers required at each node. We first present, SafeNoC [7], a solution that recovers from communication errors by reconstructing the original packets that are in-flight at the time of the error's occurrence. SafeNoC augments the NoC interconnect with a light-weight network that is formally verified and is guaranteed to be functionally correct. This second network is the basis for detecting and recovering from functionals bugs manifesting in the target interconnect. Then, we discuss, REPAIR [8], a runtime verification approach that utilizes a variant of acknowledgment-retransmission to only protect the subset of in-flight packets that are prone to errors. REPAIR identifies network regions where the execution is likely to expose a design bug and protects only the packets traversing these target regions.

6.2 SafeNoC: Correctness without Packet Replication

SafeNoC is an end-to-end solution that utilizes a novel recovery technique: upon detection of a design error, SafeNoC gathers all in-flight data, reconstructs the original packets and delivers them to their intended destination. By leveraging such an approach to recovery, we avoid the need to store redundant copies of data in-flight. SafeNoC solution relies on adding a simple and lightweight checker network that works concurrently with the original interconnect. This network is designed to be simple enough to be formally verified and guaranteed to be free of any functional errors. As a result, it provides a reliable medium through which we implement our detection and recovery processes. In the detection phase, whenever a packet is to be sent over the primary network, a signature of that packet is computed and sent through the checker network. The signature serves as a look-ahead packet and a unique identifier of the corresponding main packet, and it is used as a basis for detecting errors in the main interconnect. When a destination receives a data packet, it recomputes its signature and compares it against previously received look-ahead signatures. If a match is not found within a certain timeout period, an error is flagged and recovery is initiated. During the recovery phase, in-flight flits and packets are recovered from the network, and reliably transmitted through the checker network to all destinations. Any destination that has a mismatched signature, runs a software-based reconstruction algorithm, in which it uses the recovered flits to reconstruct the original data packets, so that they match their corresponding signature. Figure 6.1 shows a baseline CMP interconnect overlayed with our checker network. Both the checker router and the NoC router connect to the network interface, to which we also add two signature calculation units. Some additions to

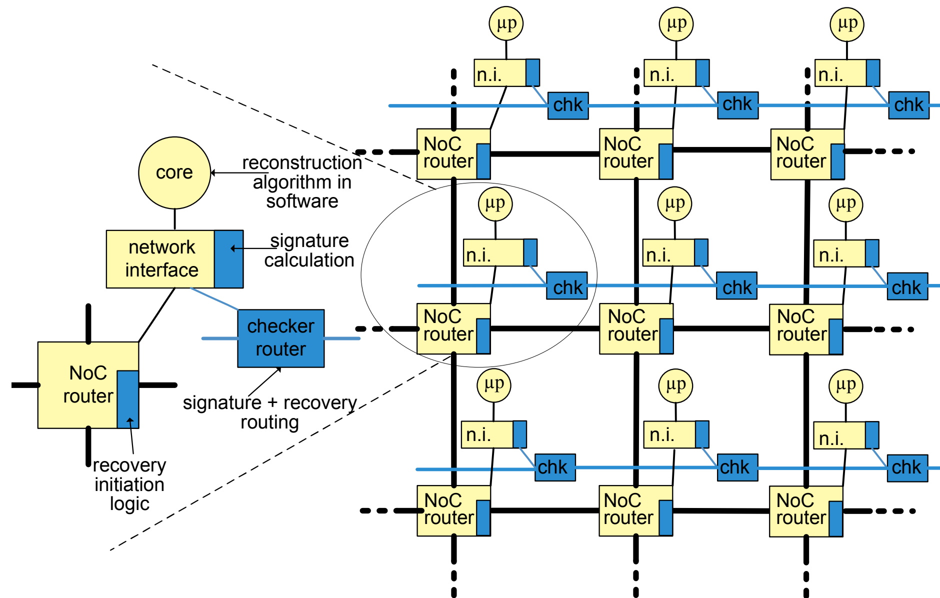

Figure 6.1 High-level overview of SafeNoC. SafeNoC augments the original interconnect with a lightweight checker network. For every data packet sent on the primary network, a look-ahead signature is routed through the checker network. Any mismatch between a received packet's computed signature and its look-ahead signature flags an error and triggers recovery.

the primary NoC routers are also required for recovering in-flight flits. Once all flits have been recovered, the design's microprocessor cores run a software algorithm to reconstruct the original packets.

6.2.1 Checker Network

SafeNoC's checker network is designed to satisfy three main properties: i) it should incur minimal area overhead. ii) it should deploy a simple router architecture, topology and routing algorithm, so that its design can be easily formally verified. iii) Finally, it should have low latency, so that it can deliver look-ahead signatures before actual data packets arrive through the primary network. Note that these properties are not a strict requirement for the correctness of the SafeNoC solution; however additional costs in area, development time and/or performance are incurred when they are not met. A checker network that does not deliver signatures before their corresponding data packets would only introduce a performance penalty but would not prevent detection and recovery. Based on the characteristics of the primary interconnect, the checker network can be carefully designed in order to minimize the occurrence of such cases. In addition, since the checker network is designed to be simple enough to be amenable to formal verification, it can be assumed that it is free of

functional bugs.

6.2.2 Error Detection

For each packet sent over the primary network, SafeNoC sends a corresponding look-ahead signature packet via the checker network. The signature calculation is based on a combination of shift-XOR operations. For a signature to uniquely identify a packet, its value must depend on the flits' data values, as well as their order within the packet. As a result, every flit in the data packet must be augmented with a flit ID, which is transmitted along with the flit data on the primary network.

Assuming that data packets have 64-bit flits, we find that a 16-bit packet signature is successful in achieving a low aliasing probability. In this case, to compute a packet's look-ahead signature, the 64-bit data of each flit is rotated by a fixed amount that depends on the flit's position. The resulting values are XORed together into a 64-bit intermediate value. The intermediate value is divided into 4 parts that are then XORed to give the final 16-bit signature. This solution provides an effective signature mechanism, which has a very low silicon area profile. It also has no performance overhead, since the signature is calculated incrementally and concurrently with the flit's data being injected into the primary network. To estimate the aliasing probability, we set up a Monte-Carlo-based simulation: we randomly created 6,000 data packets and computed their signatures. We then randomly permuted each packet's flits, and for each permutation we re-calculated the signature. If the new signature matched the original, then aliasing had occurred. For each of the 6,000 samples, we tried approximately 30 million distinct permutations, and we obtained a total probability of aliasing of 3.05×10^{-5} with a 95% confidence interval. For interconnects with larger data widths, the signature size can be tailored accordingly so as to maintain a similarly low aliasing probability. However, the probability that an error goes undetected does not just depend on the probability of aliasing, but also on other factors such as the timing of its occurrence. As a result, the overall probability of not detecting an error is much lower than the aliasing probability.

Each destination router maintains a timeout counter for every look-ahead packet it receives. The counter is incremented at every cycle until the data packet is received and its signature is re-computed. If the new signature matches any of the look-ahead signatures, then this packet is considered to have been delivered correctly. However, if there is still no match when the counter times out, an error is flagged and recovery is initiated.

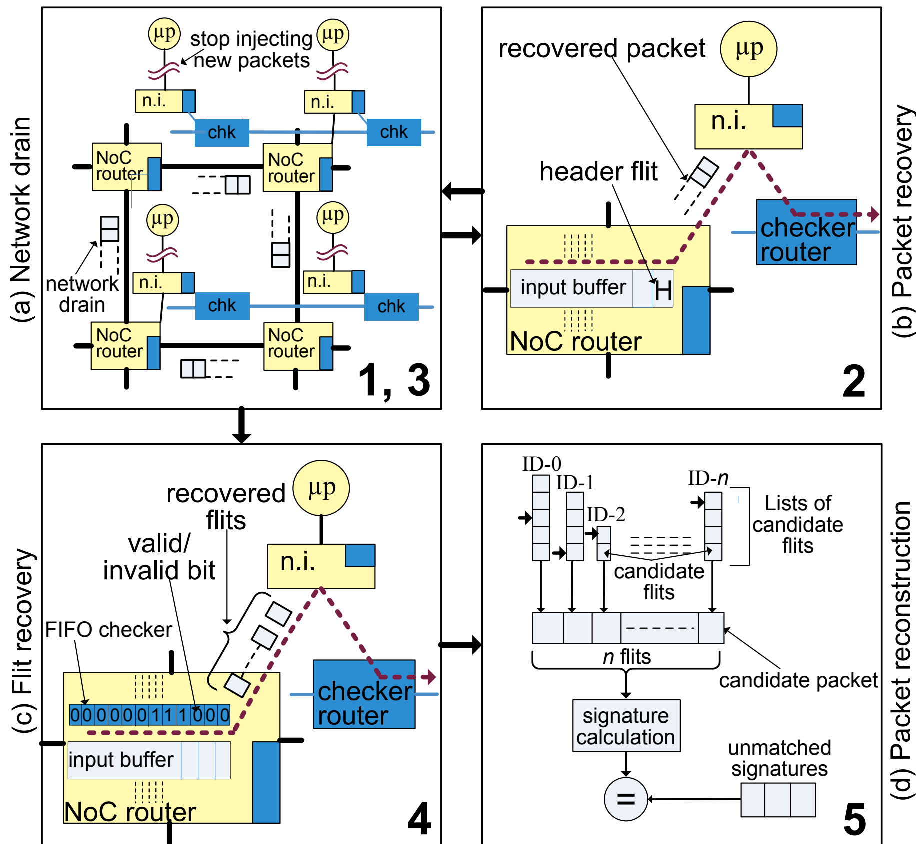

Figure 6.2 SafeNoC recovery process. Recovery proceeds in five steps with network draining occuring twice. The last step is executed in software, while the others are implemented in hardware.

6.2.3 Error Recovery

When an error is detected, the interconnect enters the recovery phase, consisting of five steps: *network drain*, *packet recovery*, another *network drain*, then *flit recovery* and *packet reconstruction*.

In the first step, a *network drain* phase is initiated, during which the network is forced to drain its in-flight packets for a preset amount of time. As shown in the top left corner of Figure 6.2, during this phase, no new packets are allowed into the network, while in-flight packets continue moving towards their destinations. If those packets are error-free, they will match their signatures and be ejected from the network. This draining stage clears the network from all in-flight traffic that was not affected by the functional bug that manifested.

The network then enters *packet recovery* and tries to recover packets that are deadlocked within the network. During this phase, primary routers remain active, but we prevent them from processing new data packets by disabling all the virtual channel allocation functionalities. If there is a deadlock in the network, then there is at least one packet in one router that is blocked waiting for allocation. To cope with this situation, we use a token-based

protocol, in which a token circulates through the checker network. When a primary router notes the token in its corresponding checker router, it checks its input buffers to determine if there is such a deadlocked packet, in which case it transmits it over the checker network, as shown in the top right block of Figure 6.2. Since all other router functionalities are still active, the entire packet can be drained and is then transmitted over the checker network to its destination. Once the token has circulated through all primary routers, they resume their full functionality and the entire network enters the second *network drain* phase. If the previous phase had recovered packets involved in deadlocks, then the remaining packets would automatically drain from the network during this second network drain phase. Note that this situation allows us to avoid the reconstruction of a number of packets that are now being delivered through the primary network, greatly reducing the packet reconstruction computation.

```
1.  Reconstruct (candidate_flits, signatures)
2.    while (!Empty(signature_buffer))
3.    success = false
4.    while(!success && !tried_all_combinations) do
5.     curr_candidates = GetNextCombination()
6.     candidate_pkt = AssemblePkt(curr_candidates)
7.     calc_signature = CalcSignature(candidate_pkt)
8.     success = MatchSig(calc_signature, signatures)
9.    end while
10.   if(success)
11.    RemoveCandidates(curr_candidates)
12.  end while
13. end Reconstruct
```

Figure 6.3 Packet reconstruction algorithm The algorithm reconstructs packets by considering combinations of candidate flits. When a candidate packet's signature matches a look-ahead signature, the data packet is considered correct. Reconstruction ends when all look-ahead signatures have been matched.

In the forth step, *flit recovery*, we recover stray flits from the network. A flit is considered stray if it is stuck in a router buffer or if it has been delivered to the wrong destination. All stray flits are candidates for the final reconstruction process. The bottom left portion of Figure 6.2 illustrates this phase: we added a FIFO checker to every input buffer of each router, to identify valid flits. The FIFO checker has 1-bit entries and its own read and write pointers following those of the input buffer. A write to the input buffer changes a corresponding entry in the FIFO checker to a valid entry and a read invalidates it. Using the same token-based protocol as in the previous phase, a router holding the token examines

its FIFO checkers for valid entries. If any exist, the corresponding flits are transmitted over the checker network to all destinations in the network. As for stray flits at the network interface buffers, their presence in the buffers at this point of the recovery process indicates that they have not matched any signatures, thus they are also candidates for reconstruction, and they are also circulated over the checker network.

During the last phase, *packet reconstruction*, the processor cores that have flagged an error run a software algorithm to reconstruct the original packets destined to them using the flits collected in the previous steps. Candidate flits are organized in separate groups, one for each flit ID, and an index is maintained for each group to indicate which flits have already been considered. The pseudo-code of this algorithm is presented in Figure 6.3. For each flit ID, a candidate a candidate is chosen and added to the set of current candidates. The current candidates are then assembled into a new packet and its signature is computed. If the new signature matches any of the remaining look-ahead signatures, then this packet's reconstruction is deemed sucessful, the packet is delivered to the application and all its flits are removed from the candidate groups. If a match cannot be found, the process is repeated, generating new sets of candidates, until all possible combinations have been tried. The algorithm ends when all look-ahead signatures have been matched.

In the case of multiple functional errors occurring consecutively, SafeNoC is still able to recover successfully. After the first error is detected and recovery is initiated, any subsequent error can only manifest in the primary interconnect during one of the network drain phases. In that case, flits that failed to drain from the network are recovered during step 4, flit recovery, along with the erroneous flits resulting from the first error. Note that during SafeNoC's recovery, functional bugs can not manifest in the checker network or the recovery process since they are formally verified.

6.2.4 SafeNoC Implementation

We modeled a baseline CMP system in Verilog HDL and using the cycle-accurate BookSim simulator. The main network is an 8x8 mesh using XY routing, and the main NoC routers are input-queued VC routers, with 5 ports, 4 pipeline stages, 2 virtual channels, and 8 flit buffers. Data packets consist of 16 flits of 75 bits each, including ECC and flit IDs. The main network was then augmented with a ring checker network and modified to include SafeNoC's detection, recovery and reconstruction functionalities. Based on the properties of the baseline primary NoC, we chose a ring topology for the checker network because of its simplicity and small area overhead. In addition, since the checker network transmits look-ahead signatures of fixed size (16-bits in our case, as we discuss in Section 6.2.2),

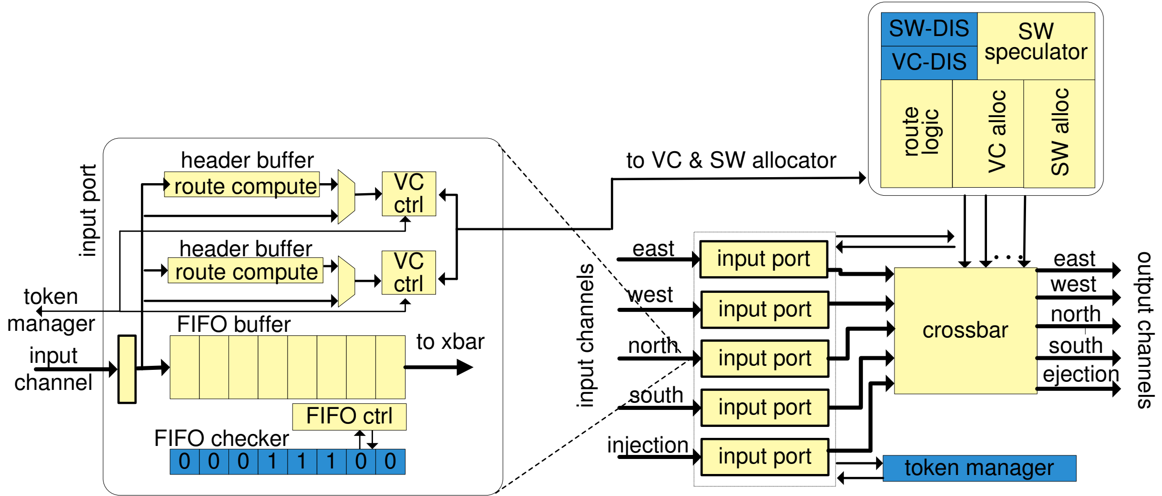

Figure 6.4 SafeNoC's hardware implementation. VC and SW disablers, FIFO checkers, and a token-manager are added to the primary routers to implement SafeNoC's recovery.

we tailor its channel bandwidth accordingly so to achieve both efficient bandwidth utilization and area savings. To optimize the performance of the checker network, we leverage the simple, single-cycle latency, packet-switched router, based on the solution proposed in [35].

Using the hardware implementation, we formally verified the portion of the system involved in recovery, including the checker network, ensuring that it operates correctly. We also analyzed the area overhead of the SafeNoC solution, synthesizing the Verilog design with a 45nm target library. The impact of recovery on performance was evaluated using the C++ simulator modeling a variety of functional bugs in the baseline system. The model was simulated with two different types of workloads: directed random traffic (uniform, transpose and bit complement), as well as application benchmarks from the PARSEC suite [16]. We also integrated SimpleScalar [19] in our architectural simulation to estimate the reconstruction algorithm's execution time. The network drain time (steps 1 and 3 of recovery) was set to 2,000 cycles and the packet delivery timeout to 4,000 cycles.

HW additions and Area Overhead

In order to implement SafeNoC, we added a few functionalities to a baseline interconnect design, categorized in three groups: to the network, to network interfaces and to the primary routers. At the network level, we add the checker network, using a ring topology with simple single-cycle latency routers, as overviewed in Section 6.2.1. Checker routers are packet-switched, with 2-entry input buffers and use a rotary flow control, giving packets already in the network priority over packets waiting to be injected into the network, so to

guarantee forward progress for packets within the checker network at each cycle. In [35], the authors show that the checker router design leads to a per-hop latency of one cycle and a deadlock-free ring network. The resulting router is sufficiently simple that it can be formally verified for correct operation, as we discuss in Section 6.2.4, and it has very low area overhead, as shown in Table 6.1. In addition, to transmit the signature packets, the checker network augments each packet with a destination address. During recovery, data flits sent over the checker network are partitioned into smaller blocks and then transmitted back-to-back, since the links of the primary network are wider than those of the checker network. SafeNoC also adds a flit ID to each data flit, slightly increasing the primary network's channel width. At each network interface, we add two *signature generation units*, one to generate signatures for packets entering the primary network and the other to verify signatures of packets reaching their destination. Signatures received through the checker network are also stored in *signature buffers* and they are compared in a *signature comparator unit* against those of incoming packets.

Figure 6.4 shows the router level hardware additions to the primary routers. First, a token manager is added to the routers to manage the passing of the token during the packet and flit recovery steps. In addition, a *virtual channel allocation disabler* (VC-DIS) and a *switch speculation disabler* (SW-DIS) are included to prevent routers from processing new packets during the packet and flit recovery phases. We also disable switch speculation during the entire recovery process to keep the SafeNoC operation simple and easily verifiable for correctness. Besides these modifications, the packet recovery phase is implemented with very little overhead, as it relies on the router's existing functionalities to retrieve packets from the buffers, with the exception of a FIFO checker for every input buffer to keep track of valid entries.

	design	**area** (mm^2)	**%**
Baseline	8x8 mesh	4.8	100
SafeNoC additions	router additions	0.15	3.3
	NI additions	0.18	3.75
	checker router	0.11	2.3
SafeNoC overhead over 8x8 mesh interconnect: 9.35%			
SafeNoC overhead over complete 64 SPARC CMP: 2.41%			

Table 6.1 SafeNoC area overhead.

We evaluated the area overhead of these modification. Our results, reported in Table 6.1, indicate that SafeNoC has a 9.35% silicon area overhead over an 8x8 mesh primary network, corresponding to a 2.41% overhead over a complete CMP with 64 SPARC cores and the same baseline 8x8 NoC. We compared this overhead to a mainstream end-to-end,

acknowledgement-based error recovery scheme as in [51]. Area overhead in these systems is primarily due to large data buffers needed to store the packets in-transit. We estimated the size of these buffers by monitoring the number of packets at each source waiting for acknowledgement. For an 8x8 primary network with 16-flit data packets, up to 4 data packets can be awaiting acknowledgements at a single source and correspondingly the retransmission-based system incurs an area overhead of 66.3% over the baseline network and 17.4% over the CMP. Therefore, SafeNoC offers more than a 7x improvement in area overhead compared to the commonly used retransmission-based method. Our simulations also show that SafeNoC buffer storage efficiency grows with data packet to signature compression ratio, and with a data packet size of 64 flits in a 8x8 mesh, SafeNoC uses less than a fourth of the buffer space of the retransmission-based scheme. This gain can be attributed to SafeNoC's ability to provide correctness by storing small signatures, in contrast to large data packets.

Correct Functionality

To guarantee correct functionality and forward progress of detection and recovery, all components involved in these processes must be formally verified. Detection is verified by ensuring that un-matched signatures initiate recovery after the timeout threshold. Recovery initiation by a single router can be trivially verified and thus we approached the formal verification of the recovery process in two steps:

Checker network. We must verify the functionality of the checker network to ensure that all packets are delivered unaltered to their correct destination within a bounded time. This goal was partitioned into three sub-goals: *eventual injection*, which guarantees that a packet awaiting injection will eventually enter the network; *forward progress* ensuring that packets progress on a path towards their destination; and *timely ejection* guaranteeing that packets are eventually ejected at the correct destination. Since the checker network is designed to be simple, its formal verification is not as challenging as that of the primary interconnect. The formal verification is even further simplified by the fact that the checker network transmits one flit signatures and not large data packets, which makes its router architecture and protocol inherently simpler.

Interaction with the main network. We must also verify that the checker network interacts correctly with the primary network to recover the data in transit. The large state space of the complex baseline network is a challenge for formal verification. We overcome it by disabling all hardware units not involved in the recovery process, such as the VC allocators and SW speculators. We first verified that the checker network could extract a complete

packet from a primary router. Next, we verified that during flit recovery, all valid flits are extracted from the primary network router. We also validated the complement of the two properties above, to check that only valid data is extracted. Finally, we checked for fairness and exclusivity among the primary routers during recovery, verifying that data is salvaged from one router at a time.

All properties to be verified were expressed with System Verilog Assertions [1], embedded in a 2x2 mesh version of the CMP equipped with SafeNoC, and then formally verified with Synopsys' Magellan [65]. Since the checker network has a simple topology, router architecture and protocol, its formal verification was completed without obstacles. As for the verification of the interactions with the main network, it is sufficient for us to formally verify these properties for a smaller 2x2 interconnect. Indeed, during recovery only one router in the primary network is active at a time, eliminating complex interactions and concurrent communication in the network. Note that this aspect does not hold true during normal operation, during which the correctness of a smaller portion of network does not imply correctness of the full system.

Design Parameters

For an effective SafeNoC implementation, the design must be tailored to the characteristics of the baseline interconnect. The signature size is a design parameter that affects the performance of the checker network and thus needs to be appropriately selected. As explained in section 6.2.1, we aim to have a checker network that delivers signatures to their destinations before the corresponding data packets arrive through the primary NoC, while minimizing the number of times when it lags behind. This can be achieved by choosing a suitable signature size that can identify errors with minimum aliasing, while being sufficiently small so that in most cases signatures can be transmitted on the checker network faster than data packets. In our implementation, we chose 16-bit signatures, and we validated this design decision by conducting experiments to determine how often the checker network "wins" against the primary network. We found that in this setup, at 0.38 flits/cycle per node injection rate (at the onset of saturation), for 99% of the packet-signature pairs, the signature arrives first, and in the few cases where the signature is lagging, the difference is less than 3 cycles. The packet delivery timeout and the signature buffer size are two other design parameters that need to be tuned to reduce area and performance overheads. The former determines when recovery is initiated, with large values delaying the initiation and small values resulting in false positives. To determine an appropriate packet delivery timeout, we measured the lead time of the checker network packet delivery, in a congested

network with both random traffic and PARSEC benchmarks, and found 4,000 cycles to be a conservative timeout. The size of the signature buffer in the network interfaces is determined by the maximum number of outstanding signatures. On the verge of saturation, this value is 11, thus we designed our system with 12 entries to have a safety margin. In the rare event of filling all signature buffers, our system triggers a recovery.

6.2.5 Evaluation of SafeNoC's Performance and Error Recovery

Bug name	Bug description
dup_flit	a flit is duplicated within a packet
misrte_1flit	a flit is misrouted to a random destination
misrte_3flit	3 flits of a packet are misrouted to a random destination
misrte_1pkt	a packet is misrouted to a random destination
misrte_2pkt	2 packets are misrouted to random destinations
misrte_flit_pkt	a packet is misrouted, another packet's flits are misrouted
dup_pkt	a packet is duplicated
dup_misrte_pkt	a packet is duplicated and one copy is misrouted
reorder_flit	flits within packet are reordered
deadlock	some packets are deadlocked in the network
livelock	some packets are in a livelock cycle in the network

Table 6.2 Functional bugs injected in SafeNoC.

To analyze SafeNoC's performance impact and its ability to detect and recover from various types of design errors, we injected 11 different design bugs into our C++ implementation of SafeNoC, as described in Table 6.2. These bugs represent possible flit-level and packet-level manifestations of functional design errors, and are based on the error model proposed in [11]. They includes misrouting of flits or entire packets, replication of flits or packets, reordering of flits within a packet, livelock and deadlock. Note that data corruption within flits is handled by the ECC already available in the baseline system.

We ran the random traffic and PARSEC workloads while triggering the bugs once per execution, and a different bug each simulation. We ran each simulation several times, each time varying when the bug is triggered, so to capture the state of the network at 10 different execution points, every 12,000 cycles after warm-up. We also repeated each experiment with 10 different random seeds for statistical confidence. With SafeNoC, all workloads complete, delivering all packets correctly to their destinations. Figure 6.5 reports the recovery time required by each benchmark, averaged over all random seeds, activation points and bugs, for a total of 11,000 runs. On average, SafeNoC spends approximatly 11,000 cycles in the first four steps of recovery. The drain time is a preset design parameter, which,

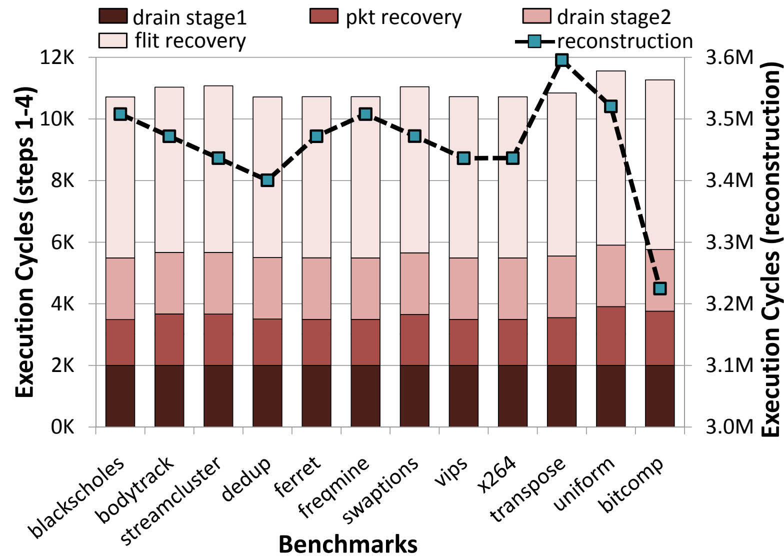

Figure 6.5 SafeNoC recovery time for each benchmark averaged over all bugs. Execution cycles for the first 4 steps of recovery (bars-left axis) and for packet reconstruction (line-right axis).

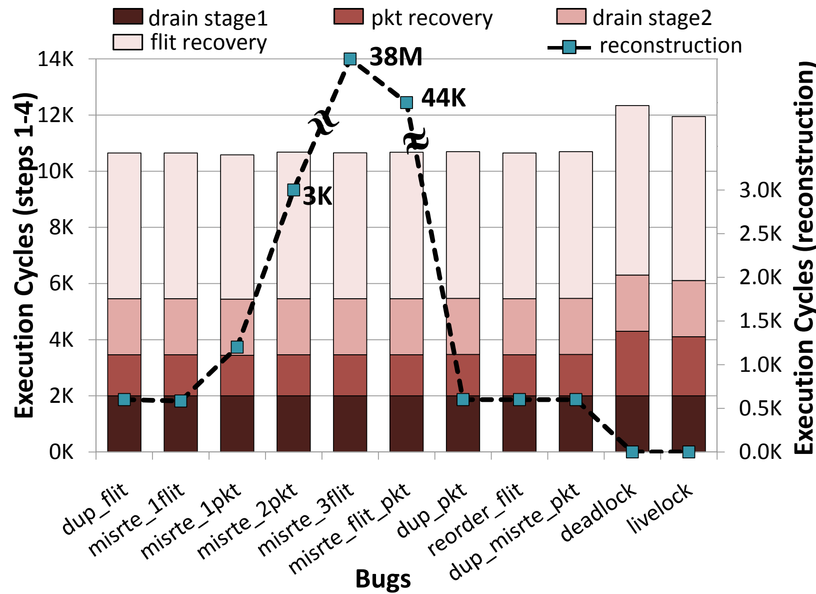

Figure 6.6 SafeNoC recovery time by bug averaged over all benchmarks. Execution cycles for the first 4 steps of recovery (bars-left axis) and for packet reconstruction (line-right axis).

in our case, accounts for a total of 4,000 cycles. The packet recovery and flit recovery steps require on average 1,600 and 5,300 cycles, respectively. In addition, SafeNoC incurs an average of 3.4M execution cycles to reconstruct erroneous packets. The performance overhead of SafeNoC is therefore dominated by the reconstruction algorithm. In Figure 6.6, we analyze SafeNoC's recovery time by bug. The reconstruction time varies widely, depending on the severity of the bug and the number of flits and packets it affected. For example, bug *misrte_1flit* mixes one packet's flit among the flits of another packet. As a

result, the reconstruction algorithm has two candidate flits in the system and it requires only 1,200 cycles to complete. At the opposite end, bug *misrte_2pkt* affects 32 flits in 2 different packets. Therefore, the reconstruction algorithm must consider two candidate flits for each position within the packet, requiring up to 38M execution cycles to complete. As for packet recovery, its time is constant for almost all bugs, at 1,473 cycles, required for the token to traverse all routers. *deadlock* and *livelock* are an exception: in these cases, entire packets must be retrieved from the primary network and transmitted over the checker network. For these two bugs, packet recovery requires 2,900 cycles and salvages 90 packets from the network on average. Once those packets are recovered, they no longer need reconstruction, thus the corresponding reconstruction time is 0. Finally, flit recovery time depends on the severity of the design error. The more packets are affected by the error, the more stray flits are left in the primary network, and the more must be recovered. Thus, on average, SafeNoC requires between 11K to 38M cycles to recover the system from a bug, assuming uniform clock domains on cores and NoC. For a CMP operating at 1GHz, and considering the extreme case when one design error manifests as often as every minute, the performance impact of SafeNoC is thus between $1.83 \times 10^{-5}\%$ and 0.06%.

We further investigate the relation between number of flits in error and reconstruction time by varying the number of flits misrouted in bug *misrte_1flit* from 1 to 14. During recovery, the number of flits retrieved doubles with the number of flits in error. Then, during reconstruction, there are two candidate flits for each missing flit position, leading to an exponential increase in reconstruction time. Therefore, the flit recovery time increases from 5,211 to 5,237 cycles, while reconstruction time increases from 1,200 to 4.9M execution cycles over the range considered.

benchmark	**%**	**benchmark**	**%**	**benchmark**	**%**
blackscholes	0.0	freqmine	0.65	vips	0.0
bodytrack	0.0	swaptions	0.77	x264	0.45
streamcluster	0.52	ferret	0.76	dedup	1.54
average performance overhead: 0.52%					

Table 6.3 Performance overhead in the absence of bugs.

In the absence of bugs, SafeNoC has a negligible performance overhead, as seen in Table 6.3. This overhead is due to false positives in SafeNoC's detection, occurring when congestion in the primary network causes a data packet's delivery to be delayed and the corresponding destination counter to timeout. However, with a carefully calibrated timeout value, the occurrence of such false positives, and thus the performance impact of SafeNoC, is minimal.

6.2.6 Limitations of SafeNoC's Runtime Correctness

SafeNoC can detect and recover from a wide variety of errors affecting both flits and packets, such as misrouting and reordering errors, deadlocks and livelocks, *etc*. However, it does not protect against data payload bit-level errors. Moreover, SafeNoC's recovery mechanism assumes that the data contents of each flit remain intact during a packets' transfer, which requires augmenting flits with error correction code (ECC). As such, SafeNoC can not recover from design errors that result in bit-level corruptions beyond the capabilities of the ECC in-use. In addition, since SafeNoC does not retain a copy of the data in-transit, it can not recover from errors that cause flits to be dropped in transfer. To overcome this limitation, it is necessary to provide additional detection mechanisms to initiate recovery before data is dropped.

6.3 REPAIR: Correctness through Adaptive-Region Protection

Without storing redundant copies of in-flight packets, functional bugs causing bit-level data corruptions and the dropping of flits and packets are difficult to recover from. On the other hand, to protect the network's execution from escaped design bugs, it may not always be necessary duplicate all packets injected into the network. Design bugs that escape into released products tend to be well-hidden and are only exposed when the runtime execution triggers a specific complex set of events or corner-cases that were never tested or observed during design-time verification. However, at runtime, not all regions of the interconnect exhibit the same runtime conditions. Applications running on a CMP or SoC design typically have spatially and temporally varying traffic patterns, such that, throughout an application's execution, the operations observed in some regions of the network are more likely to expose latent design bugs. REPAIR, (**R**untime **E**rror **P**rotection over **A**daptively **I**dentified **R**egions),is a runtime solution that adaptively determines error-prone regions of the network and then uses acknowledgment-retransmission to protect only those packets traversing the target regions.

Before a packet enters an error-prone region, it is copied into a retransmission buffer within the network interface of the routers at the periphery of that region. This packet is then marked as an acknowledgment_required (*ack_required*) packet and is transferred forward, towards its destination. Upon correct delivery to its final destination, an acknowledgment is sent back to the intermediate node that created the copy and the retransmission buffer is freed. If the intermediate node times-out before receiving an acknowledgment,

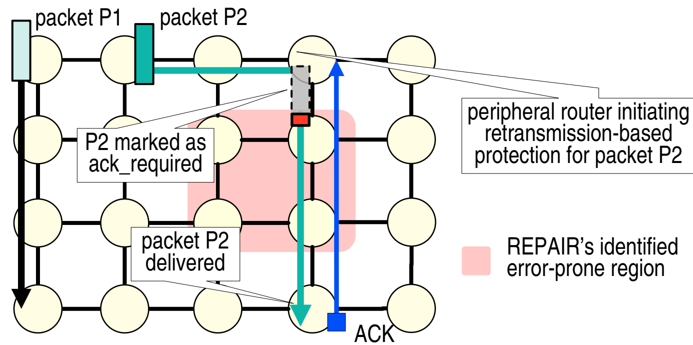

Figure 6.7 High-level overview of REPAIR. REPAIR identifies network regions exhibiting operations prone to exposing latent design bugs, and protects packets traversing these regions from being affected by the bugs.

it initiates REPAIR's recovery scheme. During recovery, routers independently drop all *ack_required* in-flight packets and retransmit a copy of them from the retransmission buffers, in which they are stored. The manifestation of design bugs at runtime is an infrequent occurrence, as bugs are only exposed during specific corner-case execution scenarios that were not verified. By clearing in-flight *ack_required* packets and retransmitting them, the network is extremely unlikely to encounter again those exact same conditions that triggered the original bug. Figure 6.7 depicts a high-level overview of our solution. There, packet P2 is traversing an identified error-prone network region, and it is protected by REPAIR's acknowledgment-retransmission.

There are several key differences between REPAIR'S acknowledgment-retransmission and traditional source-based retransmission solutions. First, in REPAIR, copies are made for only a small subset of packets, and any intermediate node along a packet's path may decide to be the one that initiates protection via a copy. Second, traditional retransmission solutions recover by retransmitting only the timed-out packets, and thus cannot overcome all types of design bugs, such as deadlock-type bugs. In contrast, REPAIR employs a network-level recovery scheme to clear the effect of the bug before retransmitting all ack_required packets.

Example. Consider the case of a deadlock occurring at runtime in an error-prone region, due to a bug in the implementation of the routing algorithm. For the deadlock to have happened, it means that the runtime execution encountered a precise sequence of events that allowed the deadlocked packets to arrive to a specific set of congested routers and request specific resources with the precise timing that caused the cyclic dependency to form. This exact scenario is unlikely to be easily replicated, otherwise this bug would probably have been caught earlier during development-time verification. In a network equipped with REPAIR, routers are instrumented with timeout counters. In a deadlock situation, at least

one counter will timeout, causing its corresponding router to initiate REPAIR's recovery scheme. During recovery, routers drop in-flight ack_required packets, which clears the deadlock. Then, each router independently retransmits a copy of the packets residing in its retransmission buffers. The network conditions, timing of packet injections and transfers, and the availability of resources are likely to have changed upon retransmission, highly reducing the chances of triggering the same corner-case behavior that caused the deadlock to occur.

6.3.1 Identifying Error-Prone Network Regions

The design-time verification of NoC interconnects inititally focuses on verifying the correctness of the NoC's most fundamental operations, which often occur when only a small number of nodes are injecting traffic into the network, consequently generating simpler network operations and interactions. Once this initial verification is complete, additional randomly-generated traffic patterns and application traffic are run to exercise the more complex execution scenarios that could occur in the network. In practice, the majority of design bugs uncovered during the development of NoC-based designs tend to manifest in complex behavior involving a large number of simultaneous interactions between the design's components. However, as the number of active nodes and the contention among in-flight packets increase, more elaborate interactions begin to occur between packets and within network components. As a result, the design space spanned by all possible NoC operations grows exponentially and becomes intractable. Thus, design-time verification efforts focus on validating the fraction of these executions that represents the most common network behavior. The remaining susbet of complex executions are not excercised nor validated, and it is precisely in those scenarios that design bugs remain hidden.

Once the system is released, a large number of users execute a vast spectrum of different applications, stimulating the NoC much more thoroughly than it was possible during development. Note that, during an application's execution, different regions of the network experience different patterns of activity, depending on the application demands. Lightly loaded network regions have a few packets traversing them, and hence observe limited contention between in-flight traffic. Therefore, the operational conditions in these regions are among those simple execution scenarios that are well-verified. Whereas, network regions with intense activity, involving a larger number of contending packets and multiple injecting nodes exercise some of the more complex operations, which may have not been exhaustively verified. Therefore, the execution conditions in those regions are prone to exposing latent design bugs.

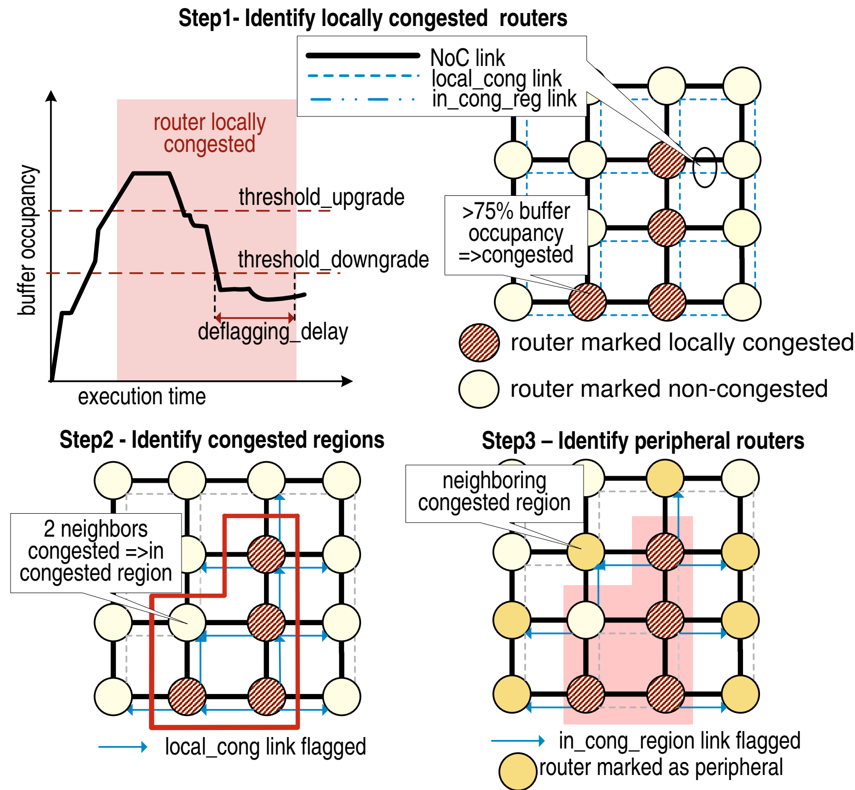

Figure 6.8 Identifying congested regions and peripheral routers. The network is equipped with two 1-bit bidirectional links to allow routers to distributively identify congested regions. First, each router establishes its local congestion status and sets its local cong link. Then, routers identify whether they belong to a congested region and, finally, the un-congested routers determine if theylie at the periphery of a congested region.

In our solution, we seek to identify those network regions with complex activity, so that we can protect packets traversing them from being affected by the latent design bugs. We rely on congestion metrics to approximate when a region of the network is exhibiting complex behavior. Congestion is not only a direct measure of the number of packets in-flight, but it also captures contention, and hence the amount of interactions observed within router components and across routers. Moreover, congestion is often a trigger for some of the more intricate features in an NoC designs. For example, CMPs and SoCs commonly employ power management techniques that utilize various approaches, such as injection throttling and frequency scaling, to restrict power dissipation when the network is heavily loaded. Additionally, congestion plays an important factor in many routing protocols, adapting the routes followed by packets, as well as triggering protocol-level deadlock recovery and congestion avoidance measures. The interactions between such features and the NoC's regular operations can create intricate executions and potentially exercise an unverified corner-case that is hiding design bugs.

To identify congested regions, REPAIR relies on an approximate congestion measure that takes into account the amount of traffic observed at routers and the contention for buffering resources. Our 3-step algorithm, illustrated in Figure 6.8 is distributed and iterative allowing us to dynamically and adaptively classify congested regions.

Step 1 - Identify locally congested routers. Routers determine their local congestion status, based on their internal buffer occupancy, similar to the scheme described in [26]. Each router keeps track of the number of input buffer entries in-use across all of its input buffers. If the number of occupied entries exceeds a certain fraction, then the router flags congestion. We refer to this occupancy threshold as *threshold_upgrade*. Once the congestion flag is set, it can only be deflagged when the buffer occupancy falls and stays below another threshold for a certain number of clock cycles. We refer to this second occupancy threshold as *threshold_downgrade* and the number of clock cycles as *deflagging_delay*.

Step 2 - Identify congested regions. Once the local congestion flag is set at each router, it is passed to all of its direct neighbors, through a 1-bit bi-directional link added between all routers (*local_cong* link). Then, each router determines whether it belongs to a congested region, based on two criteria: 1) Is the router itself congested? or 2) Is it neighboring two congested routers? The second criteria allows grouping congested routers into more contiguous regions. The result of this evaluation sets a new flag, *in_cong_region*, which is again transmitted to all the first-hop neighboring routers, requiring another 1-bit bidirectional link between each router.

Step 3 - Identify peripheral routers. Each router receives the *in_cong_region* flags from its neighbors. If at least one of its neighbors belongs to a congested region, the router determines that it is a peripheral router.

Note that some network-on-chip interconnects employ adaptive routing protocols, where the routes followed by packets are dynamically determined based on the state of the network and the in-flight traffic. Such protocols often measure network congestion and route packets away from the congested areas, thereby balancing the network's load and improving its performance. In a network with adaptive routing, REPAIR can leverage the congestion monitoring infrastructure that is already in-place to identify the high-activity error-prone network regions. However, it may be better for REPAIR to configure different threshold values to flag congestion than the values used by the adaptive routing implementation. By setting lower threshold values, REPAIR can classify the soon-to-be congested regions as error-prone regions before they trigger changes in the routing of packets. Moreover, REPAIR can utilize the changes in the routes of packets as another indication to classify the corresponding routers as part of the error-prone regions. As such, packets that traverse the congested regions as well as those affected by the congestion are protected.

6.3.2 Achieving Communication Correctness

When REPAIR identifies a congested network region, routers at the periphery of the region, and routers within it, initiate our acknowledgment-retransmission scheme to protect packets traversing the congested region. We refer to these routers as the *initiating routers*. As packets are injected into the network, they are augmented with an error-detection code that can be added to their header flits. Once in-flight, packets traversing an initiating router and heading towards a congested network area are copied into one of the router's retransmission buffers. Such packets are marked as ack_required and upon delivery to their destination node, an acknowledgment is sent back to the initiating router. If the router times-out before receiving an acknowledgment, it assumes that an error occurred and transmits a recovery signal across the network through a 1-bit serial link that connects all routers. Afterwards, each router independently starts the recovery process by first dropping all ack_required packets traversing it and then retransmitting a copy of the packets residing in its retransmission buffers. This recovery process does not halt regular execution, as the network's operations proceed normally in conjunction with the retransmissions. Additionally, since the number of cycles needed for the recovery signal to circulate to all routers is fixed, once a router receives a recovery signal, it ignores any other recovery requests for the duration it takes the initial recovery signal to reach all routers. This ensures that if other routers time-out within that duration, recovery is not initiated repeatedly.

REPAIR can overcome from design bugs causing packet corruptions as well as those causing incorrect and delayed packet delivery. The error-detection code that is added to every packet is used by the destination to detect any corruptions in-transfer. This along with the timeout functionality at initiating routers is sufficient to flag bugs causing packet corruptions and incorrect packet delivery. Moreover, REPAIR's recovery scheme can also overcome both of these errors categories, even when the retransmitted packets follow the same paths as the original ones. First, by dropping all ack_required packets, REPAIR ensures that the effects of the detected bug are cleared from the network. Second, design bugs during runtime operation manifest rarely and are exposed only under specific execution conditions. During recovery, when all ack_required packets are retransmitted, they are likely to encounter different operational conditions and timings that will not trigger the same bug to re-manifest, allowing REPAIR to side-step the design bug.

Creating Packet Copies at Initiating Routers

REPAIR maintains a congestion status table per router to record which of the router's output directions are congested. We also assume that the retransmission buffers at each node

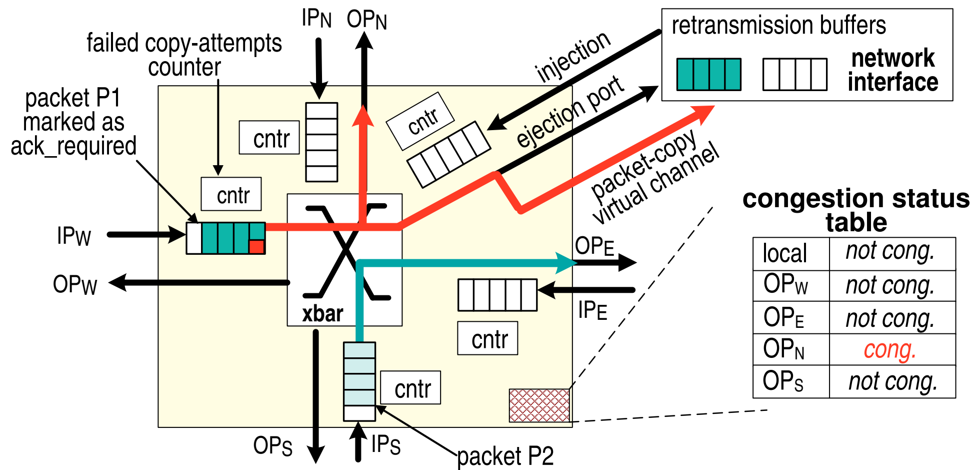

Figure 6.9 Applying acknowledgment-retransmission at a peripheral router. Since Packet P1 is heading to the North (entering a congested area), it is protected by REPAIR. This peripheral router duplicates P1 and forwards a copy to be stored in one of the retransmission buffers.

reside in the corresponding network interface. Thus, when an initiating router decides to apply acknowledgment-retransmission to a particular packet, it must duplicate it and eject a copy to its network interface. The packet is then marked as ack_required and no other router will make another copy of it. Figure 6.9 shows a peripheral router applying acknowledgment-retransmission to packet P1. P1 is heading towards the north direction, which is marked as congested. As P1 is forwarded to its destination output port (OP_N), a copy of it is ejected to be stored in one of the retransmission buffers. If the initiating router is itself in a congested region, it initiates REPAIR's protection for any previously unprotected packet traversing it and for every packet it is injecting into the network. This situation may arise because our congested regions vary during execution, and a router may become marked as congested while packets reside in its buffers.

In a typical wormhole router, packets undergo four main steps: route computation (RC), virtual channel allocation (VCA), switch allocation (SA) and crossbar traversal. Once a packet's output direction is known after the RC step, a simple look up in the congestion status table allows REPAIR to classify whether this packet requires protection. If the packet requires protection, then, in the VCA step, it must request a virtual channel in its assigned output port and in the ejection port. The packet completes VCA only if both virtual channels have been granted. If all retransmission buffers are full, the packet must stall. REPAIR utilizes a separate virtual channel to eject packet copies to the retransmission buffers, as we discuss further in Section 6.3.2. Similarly, during the SA step, the flits of the packet requiring protection must simultaneously request two crossbar connections, one to its desired output port and another to the ejection port. When both connections are granted, the orig-

inal flits proceed to their destination output port, and a copy of each flit is ejected to be stored in a retransmission buffer. REPAIR's packet duplication is similar to multicasting, a common technique in NoCs, but, in our case, the multicast is limited to the ejection port and the packet's desired output port.

Cyclic Dependencies and Deadlock Avoidance

In REPAIR, we can categorize in-flight traffic as ack_required packets, acknowledgment packets (ACKs), or regular packets. We can further group regular packets into those that are about to be ejected from the network to their destination (ejecting_packets) and packets that are still traversing the network to their destination (traversing_packets). Cyclic dependencies can arise between ack_required packets and the other three types of traffic. In this section, we discuss the situations where these dependencies can lead to deadlocks and we provide mechanisms to overcome them.

ACKs vs. ack_required packets: To prevent the blockage of in-flight ACKs behind stalled packets waiting for available retransmission buffers, we utilize a separate virtual channel for ACKs. This virtual channel can be designed with fewer buffering resources than the regular virtual channels of the network, as ACKs are only 1-flit long. The separation of acknowledgment packets from the remaining traffic is common in acknowledgment-based solutions and it is also typical for NoCs to have separate virtual networks for different types of traffic to eliminate message-dependent deadlocks.

Ejecting_packets vs. ack_required packets: At an initiating router, an ack_required packet traverses the VCA and SA steps only when the ejection and its desired output ports are both available. Such a packet can acquire a virtual channel in both ports, but then stall during SA, impeding other packets from being ejected. Thus, we add another ejection virtual channel to be used only for ejecting packet copies to retransmission buffers. The separation of ejection resources ensures that all packets can eventually be drained at their destination.

Traversing_packets vs. ack_required packets: Packets stalled waiting for a free retransmission buffer can block other in-flight packets, including those packets whose copies are occupying the retransmission buffers. If those packets cannot reach their destination, ACK packets cannot be sent back to release the retransmission buffers and resolve the blockage. To prevent the occurrence of a deadlock, we allow a fixed number of attempts in acquiring a retransmission buffer. If the packet fails all the attempts, we override the decision to copy the packet and we allow it to proceed to the downstream router. In our experimental setup, we set the maximum stalling duration to 256 cycles and found that this countermea-

sure affects very few packets. In all 15 workloads, REPAIR successfully protects >99% of packets traversing congested regions.

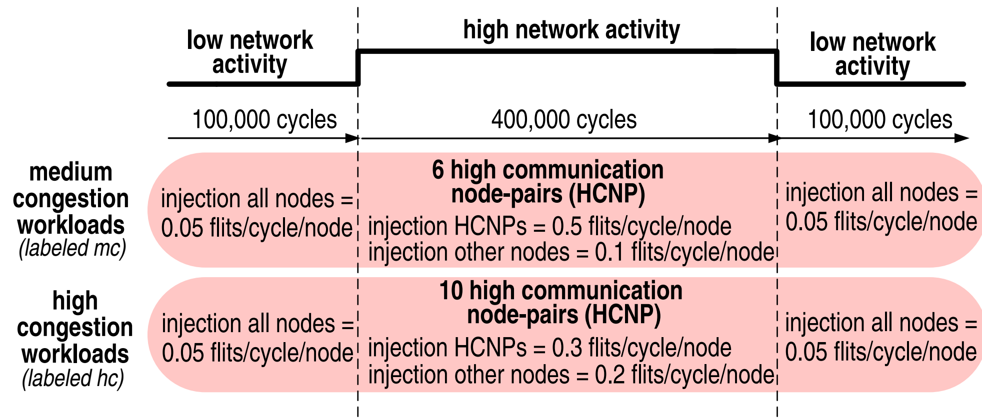

Figure 6.10 Traffic workloads considered for evaluating REPAIR. The second phase of each workload is a period of higher network activity.

6.3.3 Evaluation of REPAIR

We modeled an 8x8 mesh interconnect using the Booksim simulator and implemented REPAIR. We also generated directed random traffic workloads to model communication patterns of applications running on CMPs and SoCs where, depending on the type and scheduling of applications, some nodes communicate more than others. Moreover, the rates and the frequency of communication, and the nodes involved might change throughout execution, creating different execution phases. Thus, our workloads consist of of three execution phases: The first phase is a period of low network activity, where all nodes are injecting packets at a low rate. The second phase models a more active communication period, where a certain number of randomly chosen node-pairs communicate more frequently than others. The third phase of each workload is identical to the first and re-creates another low activity period. By varying the number of node-pairs, the nodes that constitutes each pair and the injection rates, the directed random workloads create a wide range of traffic patterns, exhibiting different levels of congestion and varying congested regions.

In this section, we show results for 15 distinct traffic workloads generated by the approach described above and summarized in Figure 6.10. In every workload, the length of each of the low-activity phases (phases 1 and 3) is set to 100,000 cycles and the length of the high-activity period (phase 2) is set to 400,000 cycles, so that it is dominant and the strengths and limitations of REPAIR can be evaluated. In the workloads labeled *mc*, we set 6 high-communication node-pairs, chosen at random, to send traffic at an injection rate

Bug	Description
Bug_A	$AB = 6, AI = 3, \geq 10$ flits in E,W,L ports $sw_req = W-S$, $vc_req = E.0-N.0, E.1-N.1, W.0-E.0$
Bug_B	$AB = 7, AI = 4, \geq 16$ flits in E,S ports, ≥ 8 flits in W,L prorts $sw_req = W-S$,$vc_req = E.0-N.0, E.1-N.1, W.0-E.0$
Bug_C	$AB = 7, AI = 4, \geq 10$ flits in E,L ports $sw_req = L-S, E-W$, $vc_req = E.0-W.0, N.0-L.0$
Bug_D	$AB = 6, \geq 10$ flits in L ports $sw_req = N-S, W-E, E-L$, $vc_req = S.1-L.0$
Bug_E	$AB = 7, \geq 16$ flits in E, S ports, ≥ 10 flits in L port. $sw_req = E-W, W-E, N-L$ $vc_req = S.0-N.0, S.1-N.1, E.1-W.0$
AB: # active buffers. AI: # active input ports. N,S,E,W,L: North, South, East, West and Local ports. sw_reqs: switch allocator requests. vc_reqs: virtual channel allocator requests. *e.g.* W-S: packet in the West port is requesting the South port. E.0-N.0: packet in VC=0 of the East port is requesting VC=0 of the North port.	

Table 6.4 Description of modeled bugs.

of 0.5 flits/cycle/node, while other nodes inject uniform traffic at a much lower rate of 0.1 flits/cycle/node. As such, the generated *mc* workloads model traffic patterns with small to medium sized congested regions, and we observe regions spanning 7 routers, with peaks up to 20 routers, on average. In the remaining workloads, labeled *hc*, we assume 10 randomly chosen high-communication node-pairs that inject traffic at a rate of 0.3 flits/cycle/node, while other nodes inject uniform traffic at a rate of 0.2 flits/node/cycle. We utilize different random seeds to generate 5 variations of this type of workload. The *hc* workloads create larger congested regions, consisting of 16 up to 35 routers, on average. In all 15 workloads, the first and third phases exhibit low traffic activity, with all nodes injecting uniform traffic at a rate of 0.05 flits/node/cycle.

Bug Detection and Recovery

To evaluate REPAIR's bug detection and recovery, we modeled 5 bugs, each triggered when routers encounter a different set of conditions due to several contending packets traversing it. As soon as the conditions are met, the corresponding router drops one of its packets. This experimental framework prevents the correct delivery of a packet upon a bug manifestation, which models any type of error that could occur due to a bug, and not just dropped-packet type errors. Table 6.4 describes our bugs and their manifestation conditions. We then ran our 15 workloads and observed the total number of times each bug manifested across all workloads and the number of workloads that were affected by the bug. Being representatives of design bugs that occur at runtime, our bugs are only triggered in corner-case

Bug	# times manifested	# affected workloads	# times recovered
Bug_A	6	3 out of 15	6
Bug_B	2	1 out of 15	2
Bug_C	7	5 out of 15	7
Bug_D	3	3 out of 15	3
Bug_E	3	2 out of 15	3

Table 6.5 REPAIR's bug recovery.

executions and do not manifest very frequently or in all workloads, as shown in Table 6.5. In all cases, REPAIR detects and recovers from the bug, as these complex corner-case conditions arise, as expected, in high-traffic regions, where packets are protected by REPAIR's acknowledgment-retransmission. Thus, a copy of each erroneous packets is retransmitted and delivered correctly to its destination.

Network Performance

We also evaluated the network performance when using REPAIR. Since source-based retransmission is traditionally adopted to ensure correct communication in the presence of serveral types of errors, such as dropped packets, packet corruption and misrouting, we also compared REPAIR against source-based retransmission. Figure 6.11 shows the execution time of our workloads when using REPAIR and source-based retransmission, normalized to the execution time of a baseline system that is not equipped with any retransmission capabilities. In both source-based and REPAIR, we assume two retransmission buffers per node. A network equipped with REPAIR can achieve communication correctness with 58% faster execution times than source-based retransmission. REPAIR subjects only 25% of packets, on average, to acknowledgment retransmission, creating less contention for the retransmission buffers. Whereas, in source-based retransmission, there are many packets stalled at injection waiting for retransmission buffers, which reduces the network's throughput and causes, on average, a 75% slowdown. We further evaluated the performance trade-offs by varying the number of retransmission buffers utilized in source-based retransmission, and compared it against the execution time of a REPAIR implementation that utilizes only 2 retransmission buffers per node (Figure 6.12). The results are normalized to the execution time of a source-based retransmission solution using 8 buffers. As shown in this figure, REPAIR achieves better performance with 2-3x fewer buffer resources than traditionl source-based solutions.

Despite providing better network throughput, packet latencies may increase when using

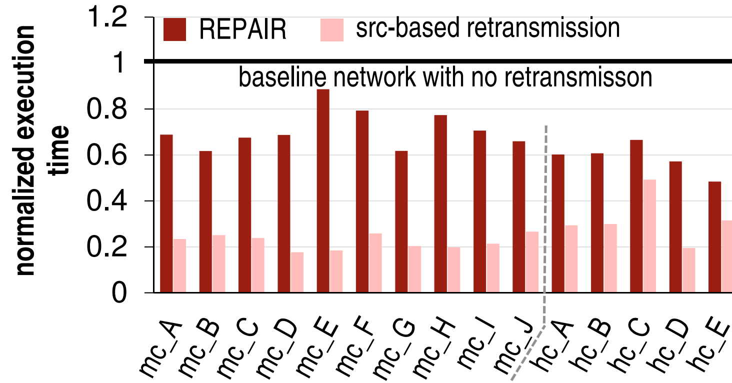

Figure 6.11 Execution time REPAIR compared to source-based retransmission. Results are normalized to the execution time of a baseline NoC, without any retransmission capabilities. REPAIR's performance is 58% better, on average, than source-based retransmission.

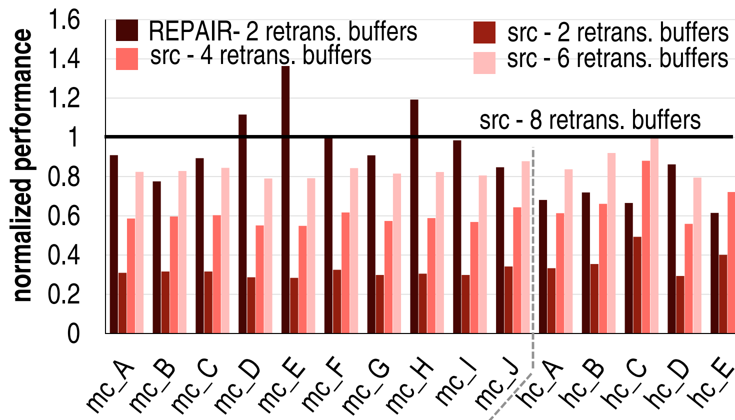

Figure 6.12 Buffering requirements of REPAIR vs. source-based retransmission. Execution time is normalized to source-based retransmission utilizing 8 retransmission buffers per node. On average, REPAIR achieves better performance, even with 2-3x fewer retransmission buffers.

REPAIR. In the absence of free retransmission buffers at initiating routers, packets needing to be copied will stall, contributing to the increased average packet latency. The average latency of packet traversal through the network when using REPAIR, with 2 retransmission buffers, is up to twice that of a throughput-equivalent source-based retransmission solution. In contrast, in source-based retransmission, the lack of free buffers causes packets to stall at injection, leading to a reduction in throughput, but once packets are in-flight, no additional stalling occurs.

Implementation Overhead

Assuming routers with 5 ports, 2 virtual channels per port and 8-flit buffers, REPAIR's congestion detection requires three 7-bit registers per router to hold each of the occupancy

counter, the *upgrade_threshold* and the *downgrade_threshold*. We set the *downgrade_delay* to 10 bits, allowing a maximum delay of 1,024 cycles before toggling the local congestion flag. The congestion status table consists of 1 entry to store the router's local congestion and 4 entries to store the congestion statuses of its neighbors. The congestion status itself is encoded with 1 bit. To implement REPAIR's protection scheme, a *failed_copy_attempts* counter is needed per input buffer to monitor the number of times a packet fails to acquire a retransmission buffer at an initiating router. We set this counter to 8 bits, capping the waiting-time to 256 cycles. Thus, the total storage overhead of REPAIR is 116 bits per router, less than a one-flit entry of a router's input buffer, which is typically at least 128 bits. As for the control logic to update these counters, its area overhead is minimal (<0.1%). Lastly, to instrument routers to create copies of in-flight packets, we require an additional output virtual channel in the ejection port and modifications to the virtual channel allocator to account for it. We modeled and synthesized a baseline router using Synopsys Design Compiler with a 45nm target library and found that the area overhead for implementing this functionality is 0.86%. The total area overhead of REPAIR is <1% (0.86% + 116 bits).

6.4 Summary

In this chapter, we present two solutions to achieve runtime correctness in network-on-chip interconnects in the face of design bugs that have escaped desig-time verification efforts. These solutions trade-off the amount of packet redundancy needed to detect and recover from communication errors. SafeNoC avoids storing any redundant copies of data in-flight. Instead, it computes a signature of every packet and sends that signature through a lightweight and simple checker network that is augmented to the original interconnect and that is guaranteed to be functionally correct. SafeNoC detects functional errors by comparing the signature of every received data packet with its look-ahead signature that was delivered through the checker network. In case of mismatches, SafeNoC's recovery mechanism collects blocked packets and stray flits from the primary network and distributes them over the checker network to all processor cores, where its reconstruction algorithm reassembles them. Using such an approach to recovery, SafeNoC can detect and recover from a broad range of functional design errors, while incurring a low performance impact and only a 2.41% area overhead in a 64-core CMP system. However, one limitation of such an approach is that the lack of redundant packet copies prevents it from recovering from design bugs causing packet data loss. In contrast, REPAIR is a solution that adaptively determines the set of packets that are prone to being affected by a design bug,

and it protects only those packets using an acknowledgment-retransmission approach. Regions of the network exhibiting high-traffic activity and high congestion encounter more complex execution scenarios that are more susceptible to the manifestation of latent design bugs. Therefore, REPAIR identifies such regions by measuring local congestion at network routers and dynamically delineating congested regions. Routers at the periphery of these target regions make a copy of each packet that is about to traverse the error-prone region. These protected packets are marked as ack_required and, upon their delivery to their final destination nodes, an acknowledgment packet is sent back to the peripheral router that created the copy. Communication errors are then recovered using a network-level recovery scheme, where all ack_required packets are dropped and retransmitted. REPAIR is successful in detecting and recovering from errors causing packet data corruptions as well as incorrect and delayed packet delivery. It also achieves better performance than a traditional source retransmission solution, while requiring 2-3x fewer retransmission buffers per node.

Chapter 7
Conclusion

This dissertation targets the problem of achieving correctness in the interconnect subsystem that has become a central component in modern computing devices. Over the years, the development of hardware systems has been exhbiting a clear trends towards the design of highly intergrated and massively parallel architectures. These trends have in turn fueled a shift in the design of the interconnect subsystem that enables the communication between on-chip components. Interconnect systems have largely transitioned to network-on-chip designs, which provide a flexible, distributed and high-bandwidth infrastructure capable of meeting the intricate communication requirements of CMP and SoC designs.

A major threat to the functional correctness of the interconnect subsystem lies in the presence of potential bugs in its design and implementation. The sheer complexity and size of NoC designs pose significant challenges to the interconnect's functional validation. The work presented in this dissertation addresses the correctness problem through a complementary set of solutions that facilitate verification efforts during development-time and that protect the interconnect from bugs that could escape into released products. Through these solutions, this dissertation enhances the functional correctness guarantees of interconnect designss, thereby enabling the continued innovation and progress in the design and implementation of these interconnect subsystems in current and future computing devices.

7.1 Summary of Contributions

Addressing verification challenges during development-time. During the design and development of the interconnect subsystem, emulation and post-silicon platforms are heavily utilized to test the interconnect under different execution scenarios. However, to exploit the performance advantage of using such platforms, it is necessary to address the challanges that they introduce. One of the major problems encountered is that emulation and post-silicon validation platforms offer limitied observability into the interconnect's internal operations. The lack of observability translates to a difficulty in detecting errors in the

interconnect's execution, localizing them within the interconnect's subcomponents, and debugging them. In this dissertation, we developed two complementary approaches to boosting observability. We first introduced a *packet monitoring* mechanism, where the interconnect is instrumented to log debug information capturing the progress of in-flight packets through the network. This data data is collected by incrementally storing it within the contents of the in-flight packets. The analysis of the collected debug information provides a reconstruction of each packet's path and the overall network execution, and it also highlights packet interactions within the network's routers and important performance statistics. By boosting observability and providing debugging support, this appraoch enhances the functional verification of the interconnect, particularly when running synthetic traffic workloads and application traces. The second solution developed in this context adopts a *router monitoring* approach by taking periodic snapshots of each router's execution. The snapshots collected from execution are then be used to reconstruct the flow of traffic and the internal operations of the network. Our router monitoring approach facilitates debugging, while also extending the capabilities of the verification framework to not only run synthetic workloads but also real application test-cases.

Addressing escaped bugs during runtime operation. In addition to supporting and improving functional validation efforts during the development of the design, this dissertation also addresses the correctness problem during the interconnect's runtime operation. We first explore and develop *SafeNoC*, a solution to provide runtime correctness guarantees without needing to create backup copies of in-flight packets. SafeNoC implements a novel error detection and recovery solution, where the interconnect is augmented with another light-weight and functionally correct network, the checker network. A look-ahead signature of each packet is computed upon its injection and is sent via the checker network to each packet's destination node. These signatures are used as the basis of error detection as well as recovery. A mismatch between a delivered packet and its look-ahead signature flags an error in exection, upon which the recovery process is initiated. During recovery, the checker network is used to extract all erroneous in-flight packets and flits. Then, the original packets are reconstructed from their erroneous counterparts so as to match their look-ahead signatures. Although SafeNoC is successful in detecting and recovering from a wide spectrum of communication errors, the lack of backup copies of packets prevents this approach from overcoming bugs causing packet or data loss. Given this limitation, we investigate and develop an alternative solution, *REPAIR*, which ensures correctness by relaxing the constraint of not storing any redundant packet copies. REPAIR adaptively identifies high-traffic regions of the network, where the runtime execution is prone to exposing hidden design bugs. Then, it protects packets traversing these error-prone regions

through a retransmission-based solution. As such, REPAIR creates backup copies for only the subset of packets in need of protection, thereby achieving communication correctness over the error-prone regions while minimizing the incurred hardware overhead.

7.2 Future Directions

The work presented in this dissertation opens the door to other future research directions in the area of runtime verification of the interconnect. Runtime verification solutions can alleviate the burden of exhaustively verifying a design during development-time by providing mechanisms to detect and recover from errors at runtime. Moreover, as the REPAIR solution demonstrated, the overhead of these runtime techniques can be greatly reduced if they are capable of discerning when error-prone runtime conditions are occurring in the network and then activating protection for only those error-prone network regions. In this context, the knowledge and experience of design-time verification can be leveraged to explore solutions that allow a more accurate identification of error-prone regions at runtime. Another interesting research avenue to explore is adapting the error-prone regions to the characteristics of applications running on the system. In many recent research works, it has been shown that different applications can have varying correctness requirements, with some types of application capable of tolerating errors. From this perspective, runtime verification solutions can be designed to adapt to the correctness requirements of applications in an effort to achieve the desired correctness requirements while reducing area and performance overheads.

Moreover, the packet monitoring and router monitoring solutions developed in this thesis, and described in Chapters 4 and 5, can be leveraged to improve the performance validation of the interconnect subsystem, as well as to perform design space exploration. FPGA-based emulation frameworks are often utilized to perform faster design space exploration in the early stages of the the interconnect's development. In addition, during the pre-silicon and post-silicon stages, the interconnect undergoes extensive performance validation. In these contexts, our approaches to monitoring the internal operations of the network can provide valuable performance statistics. These statistics can in turn aid in analyzing the performance of the interconnect and identifying potential bottlenecks and performance violations.

This dissertation also introduces research opportunities that extend beyond the interconnect subsystem. The correctness of the overall communication system is dependent on the functional correctness of the physical interconnect fabric, as well as the correctness

of the overlying communication protocols and inter-component interactions. While this dissertation focused primarily on the interconnect fabric, further research is necessary to address the functional verification of the communication protocols and component integration. In fact, the verification of these communication protocols is faced with similar challenges as the verification of the underlying interconnect fabric. The large number of interacting on-chip components and the intricacy of the communication patterns creates a distributed and parallel subsystem with an intractable design-space that must be verified. Additionally, in this context, emulation and post-silicon platforms are also heavily relied on to speed up verification efforts and achieve higher correctness guarantees. Therefore, novel solutions are needed to improve the error detection and debugging capabilities of these verification platforms while targeting specifically the verification of the communication protocols and inter-component interactions. Solutions are also needed to better direct the verification efforts to efficiently exercise the design-space that must be validated, so that the necessary correctness guarantees can be met within the short time-to-market windows of modern computing devices.

Bibliography

[1] System Verilog Assertions. http://www.systemverilog.org/.

[2] *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE, New York, NY, 2001.

[3] *Intel Core 2 Duo and Intel Core 2 Solo Processor for Intel Centrino Duo Processor Technology Specification Update*, September 2007.

[4] R. Abdel-Khalek and V. Bertacco. Functional post-silicon diagnosis and debug for networks-on-chip. In *Proc. ICCAD*, 2012.

[5] R. Abdel-Khalek and V. Bertacco. DiAMOND: Distributed alteration of messages for on-chip network debug. In *Proc. NOCS*, 2014.

[6] R. Abdel-Khalek and V. Bertacco. Post-silicon platform for the functional diagnosis and debug of networks-on-chip. *ACM Trans. Embedded Comput. Syst.*, 13(3s):112:1–112:25, 2014.

[7] R. Abdel-Khalek, R. Parikh, A. DeOrio, and V. Bertacco. Functional correctness for CMP interconnects. In *Proc. ICCD*, 2011.

[8] R. Abdel-Khalek, R. Parikh, A. DeOrio, and V. Bertacco. Runtime correctness of NoCs through adaptive region protection. In *Proc. DATE*, 2016.

[9] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. DAC*, 2006.

[10] R. Al-Dujaily, T. Mak, Fei Xia, A. Yakovlev, and M. Palesi. Run-time deadlock detection in networks-on-chip using coupled transitive closure networks. In *Proc. DATE*, 2011.

[11] A. Alaghi, N. Karimi, M. Sedghi, and Z. Navabi. Online NoC switch fault detection and diagnosis using a high level fault model. *Proc. DFT*, 2007.

[12] K.V. Anjan and T.M. Pinkston. DISHA: A deadlock recovery scheme for fully adaptive routing. In *Proc. IPPS*, 1995.

[13] K.V. Anjan, T.M. Pinkston, and J. Duato. Generalized theory for deadlock-free adaptive wormhole routing and its application to Disha Concurrent. In *Proc. IPPS*, 1996.

[14] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. MICRO*, 1999.

[15] K. Basu and P. Mishra. Efficient trace signal selection for post silicon validation and debug. In *Proc. VLSI design*, 2011.

[16] C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. PACT*, 2008.

[17] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A generic model for formally verifying noc communication architectures: A case study. In *Proc. NOCS*, 2007.

[18] M. Boul and Z. Zeljko. *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer Publishing Company, Incorporated, 2008.

[19] D. Burger and T. Austin. The SimpleScalar Toolset, version 3.0.

[20] J. Cantin, M. Lipasti, and J. Smith. Dynamic verification of cache coherence protocols. In *Workshop on Memory Performance Issues*, 2001.

[21] D. Chatterjee, C. McCarter, and V. Bertacco. Simulation-based signal selection for state restoration in silicon debug. In *Proc. ICCAD*, 2011.

[22] C. Ciordas, T. Basten, A. Radulescu, K. Goossens, and J. Meerbergen. An event-based network-on-chip monitoring service. In *HLDVT*, 2004.

[23] C. Ciordas, K. Goossens, T. Basten, A. Radulescu, and A. Boon. Transaction monitoring in networks on chip: the on-chip run-time perspective. In *IES*, 2006.

[24] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[25] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Computers*, 1987.

[26] R. Das, S. Narayanasamy, S.K. Satpathy, and R.G. Dreslinski. Catnap: Energy proportional multiple network-on-chip. In *Proc. ISCA*, 2013.

[27] A. DeOrio, I. Wagner, and V. Bertacco. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. In *Proc. HPCA*, 2009.

[28] H.D. Foster. Trends in functional verification: a 2014 industry study. In *Proc. DAC*, 2015.

[29] N. Genko, D. Atienza, G. De Micheli, and L. Benini. Feature - NoC emulation: a tool and design flow for MPSoC. *Circuits and Systems, IEEE*, 2007.

[30] N. Genko, D. Atienza, G. De Micheli, J.M. Mendias, R. Hermida, and F. Catthoor. A complete network-on-chip emulation framework. In *Proc. DATE*, 2005.

[31] A. Ghofrani, R. Parikh, S. Shamshiri, A. DeOrio, Kwang-Ting Cheng, and V. Bertacco. Comprehensive online defect diagnosis in on-chip networks. In *Proc. VTS*, 2012.

[32] C.J. Glass and L.M. Ni. The turn model for adaptive routing. *SIGARCH Comput. Archit. News*, 20(2), 1992.

[33] Intel Corporation. *Intel(R) Pentium(R) Processor Invalid Instruction Erratum Overview*, July 2004. `http://www.intel.com/support/processors/pentium/sb/cs-013151.htm`.

[34] Intel Corporation. *Intel Core i7-900 Desktop Processor Series Specification Update*, July 2010.

[35] J. Kim and H. Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proc. NoCArc*, 2009.

[36] H.F. Ko, A.B. Kinsman, and N. Nicolici. Distributed embedded logic analysis for post-silicon validation of socs. In *Proc. ITC*, 2008.

[37] H.F. Ko and N. Nicolici. Automated trace signals identification and state restoration for improving observability in post-silicon validation. In *Proc. DATE*, 2008.

[38] H.F. Ko and N. Nicolici. Resource-efficient programmable trigger units for post-silicon validation. In *Proc. European Test Symposium*, 2009.

[39] M. Kouadri, M. Abdellah, B. Senouci, and F. Petrot. Large scale on-chip networks:an accurate multi-FPGA emulation platform. In *Proc. DSD*, Sept 2008.

[40] K. Kubicki. Understanding AMD's TLB processor bug. *Daily Tech*, December 2007.

[41] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21, 1978.

[42] S. Lee. A deadlock detection mechanism for true fully adaptive routing in regular wormhole networks. *Comput. Commun.*, 30(8), 2007.

[43] X. Liu and Q. Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *Proc. DATE*, 2009.

[44] P. Lopez, J. Martinez, and J. Duato. A very efficient distributed deadlock detection mechanism for wormhole networks. In *Proc. HPCA*, 1998.

[45] S. Ma, N. Enright Jerger, and Z. Wang. Whole packet forwarding: Efficient design of fully adaptive routing algorithms for networks-on-chip. In *Proc. HPCA*, 2012.

[46] B.W. Mammo, V. Bertacco, A. DeOrio, and I. Wagner. Post-silicon validation of multiprocessor memory consistency. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 34(6), 2015.

[47] J. Markoff. Burned once, Intel prepares new chip fortified by constant tests. *New York Times*, November 2008.

[48] J.M. Martínez, P. Lopez, J. Duato, and T.M. Pinkston. Software-based deadlock recovery technique for true fully adaptive routing in wormhole networks. In *Proc. ICPP*, 1997.

[49] J.M. Martínez-Rubio, P. López, and J. Duato. A cost-effective approach to deadlock handling in wormhole networks. *IEEE Trans. Parallel Distrib. Syst.*, 12(7), 2001.

[50] A. Meixner and D.J. Sorin. Dynamic verification of sequential consistency. In *Proc. ISCA*, 2005.

[51] S. Murali, T. Theocharides, L. Benini, G. De Micheli, N. Vijaykrishnan, and M.J. Irwin. Analysis of error recovery schemes for networks on chips. *IEEE Design & Test*, 2005.

[52] J. Nan, D.U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D.E. Shaw, J. Kim, and W.J. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Proc. ISPASS*, 2013.

[53] S. Narayanasamy, B. Carneal, and B. Calder. Patching processor design errors. In *Proc. ICCD*, 2006.

[54] R. Parikh and V. Bertacco. Formally enhanced runtime verification to ensure NoC functional correctness. In *Proc. MICRO*, 2011.

[55] L. Peng, X. Chunchang, W. Xiaohang, X. Binjie, L. Yangfan, W. Weidong, and Y. Qingdong. A NoC emulation/verification framework. In *Proc. ITNG*, 2009.

[56] S. Prabhakar and M. Hsiao. Using non-trivial logic implications for trace buffer-based silicon debug. In *Proc. ATS*, 2009.

[57] A. Prodromou, A. Panteli, C. Nicopoulos, and Y. Sazeides. NoCAlert: An on-line and real-time fault detection mechanism for network-on-chip architectures. In *Proc. MICRO*, 2012.

[58] V. Rantala, T. Lehtonen, and J. Plosila. Network-on-chip routing algorithms. Technical Report 779, Turku Centre for Computer Science, 2006.

[59] J.M. Martínez Rubio, P. López, and J. Duato. Fc3d: Flow control-based distributed deadlock detection mechanism for true fully adaptive routing in wormhole networks. *IEEE Trans. Parallel Distrib. Syst.*, 14(8), 2003.

[60] R.R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6), 1997.

[61] H. Shojaei and A. Davoodi. Trace signal selection to enhance timing and logic visibility in post-silicon validation. In *Proc. ICCAD*, 2010.

[62] Yong Ho Song and Timothy Mark Pinkston. A new mechanism for congestion and deadlock resolution. In *ICPP*, 2002.

[63] D.J. Sorin, M.D. Hill, and D.A. Wood. Dynamic verification of end-to-end multiprocessor invariants. In *Proc. DSN*, 2003.

[64] Synopsys. Synopsys Design Compiler.

[65] Synopsys. Synopsys Magellan. http://www.synopsys.com.

[66] S. Tang and Qiang Xu. A multi-core debug platform for NoC-based systems. In *Proc. DATE*, 2007.

[67] M. Thottethodi, A.R. Lebeck, and S.S. Mukherjee. Self-tuned congestion control for multiprocessor networks. In *Proc. HPCA*, 2001.

[68] B. Vermeulen and K. Goossens. A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs. In *Proc. VLSI-DAT*, 2009.

[69] B. Vermeulen, T. Waayers, and S.K. Goel. Core-based scan architecture for silicon debug. In *Test Conference, 2002. Proceedings. International*, 2002.

[70] I. Wagner and V. Bertacco. Engineering trust with semantic guardians. In *Proc. DATE*, 2007.

[71] I. Wagner and V. Bertacco. Caspar: Hardware patching for multicore processors. In *Proc. DATE*, 2009.

[72] I. Wagner and V. Bertacco. *Post-silicon and Runtime Verification for Modern Processors*. Springer, 2011.

[73] I. Wagner, V. Bertacco, and T. Austin. Shielding against design flaws with field repairable control logic. In *Proc. DAC*, DAC '06, 2006.

[74] L. Yangfan, L. Peng, J. Yingtao, Y. Mei, W. Kejun, W. Weidong, and Y. Qingdong. Building a multi-FPGA-based emulation framework to support networks-on-chip design and verification. *International Journal of Electronics*, 97, 2010.

[75] H. Yi, S. Park, and S. Kundu. A design-for-debug (DfD) for NoC-based SoC debugging via NoC. In *Proc. ATS*, 2008.

[76] H. Yi, S. Park, and S. Kundu. On-chip support for NoC-based SoC debugging. *IEEE Trans. on Circuits and Systems*, 57(7), 2010.