Routing and Topology Reconfiguration for Networks-on-Chip's Runtime Health

by

Ritesh Parikh

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
2014

Doctoral Committee:

Associate Professor Valeria M. Bertacco, Chair
Assistant Research Scientist Reetuparna Das
Professor Scott Mahlke
Assistant Professor Zhengya Zhang

To my family

Acknowledgments

It is a pleasure to thank those who have been a part of my PhD journey and made this thesis possible. I am deeply grateful to my advisor, Professor Valeria Bertacco, for her insightful, supportive and motivational mentorship over the years. Having closely observed Prof Bertacco essaying the role of a teacher, mentor or manager with equal ease, I have learnt a great deal in the process and many of her practices will always be a benchmark for me.

I would like to extend my heartfelt thanks to my committee members for their active participation in steering the thesis in the right direction since the proposal presentation. Prior to that, I was fortunate to have interacted with them closely during classes, projects and collaborations. I would like to personally thank - Professor Scott Mahlke, also a part of my preliminary examination committee, whose advice was instrumental in shaping my thesis; Dr. Reetuparna Das, whose domain expertise contributed greatly to the quality of the research output, and Professor Zhengya Zhang, under whom I took my first and probably the hardest class at Michigan.

I was fortunate to have been introduced to the graduate environment by Andrew DeOrio and Debapriya Chatterjee, whose help was essential in getting started and throughout the many roadblocks of graduate work. I am also grateful to Joseph Greathouse, Andrea Pellegrini and Jason Clemons for the tip and tricks to navigate through graduate school and the research brainstorming sessions we had. I consider myself lucky to have successful collaborations with many smart people over the years, contributing in areas where I lacked insight and understanding. I particularly thank Rawan Abdel-Khalek, Andrew DeOrio, Doowon Lee, Reetuparna Das, Amirali Ghofrani, Kwang-Ting Cheng, Animesh Jain and Yu Xing. However, I will always regret missing out on collaborating with Biruk Mammo, who is an exceptional engineer, lab-mate and a good friend. Nonetheless, Rawan, Biruk and I formed a formidable brainstorming team that met bi-weekly for over two years to discuss research topics. I would also like to thank Abhijit Davare and Yatin Hoskote for providing solid guidance during my internship at Intel Corporation and improving my understanding of real-world computer architecture.

I am blessed with a great set of friends that made my stay at Ann Arbor joyful and

memorable. I thank Ayan Das for making me do things I felt too shy about, Vivek Joshi for teaching me interpersonal skills, Abhayendra Singh for helping both with research and fun, Daya Khudia for the multi-disciplinary discussions during coffee-breaks, Gaurav Chadha for being the best roommate, Animesh Banerjee for his sheer ability to energize any meeting/activity, Soumya Kundu for teaching the true meaning of taking it easy, Ankit Sethia for teaching me the importance of family, and Anchal Agarwal for rare but essential gossip. I also thank Deepti Joshi, Gaurav Pandey and Mukesh Bachhav for the wonderful times. Finally, Shweta Srivastava supported me from half-way across the globe, like no other friend could have. It is thus only natural to feel fortunate that I married her midway through my PhD.

These past few years have been a roller coaster ride but I am glad we made it. Shweta has been the more mature and understanding person in our relationship and I cannot thank her enough for that. Finally, I am grateful to my father, mother and sister for their love and support. I thank my father for always backing my decisions and fighting our battles together, and I hope we continue to do so. I thank my mother for her unparalleled love, and my sister for being my longest acting and most faithful friend, confidant and well-wisher.

Preface

As silicon technology evolves, chip multi-processor (CMP) and system-on-chip (SoC) designs are dramatically changing from limited, robust and homogeneous logic blocks to integrating billions of fragile transistors into complex and heterogeneous cores/IPs. This increased integration has compelled architects to design resource-heavy, complex and power-hungry on-chip interconnects, moving towards network-on-chip (NoC) structures. In addition, the waning reliability of silicon poses a great threat to these communication structures as they could potentially be a single point of failure. Further, the heterogeneity and fast time-to-market of upcoming devices makes it nearly impossible to thoroughly verify NoC architectures and optimize them for power at design-time. Failure of NoC architectures to meet correctness, reliability and power-budget requirements has detrimental effects on the runtime operation of NoC-based CMPs and SoCs. Therefore, highly efficient runtime detection and reconfiguration mechanisms are becoming a key requisite to unlock the full potential of future CMPs and SoCs. Such mechanisms can overcome both functional bugs that escaped design-time verification and device failures due to an unreliable silicon substrate. Similarly, these runtime reconfiguration solutions can also be leveraged to optimize the communication paths dynamically; particularly, to minimize power dissipation and prevent overheating of the NoC structures.

The solutions proposed in this thesis address the challenges to NoCs' runtime health by employing a reactive approach, *i.e.*, error detection followed by recovery. As a result, this thesis is able to address a wide range of unforeseen problems; particularly the ones arising from **design errors**, **reliability threats**, and **excessive power dissipation**. Further, the proposed solutions adopt an integrated approach, addressing both detection and recovery from errors at runtime. To attain quick and minimalistic error isolation, an application's execution is divided into fixed-time monitoring windows, or *epochs*, during which distributed checkers at each NoC router monitor the traffic activity to detect anomalous behavior. If a failure is detected, a reconfiguration procedure is triggered at epoch boundaries to circumvent the detected failures. The reconfiguration procedure is implemented with lightweight and distributed hardware, and it utilizes broadcasts to synchronize the operation of all NoC

nodes.

This thesis ensures the design correctness of NoCs at a low cost by observing that aspects that are verified at design-time do not require further checking at runtime. Therefore, it is sufficient to monitor only a subset of the NoC's functionality, along with a lightweight recovery support, to ensure complete design correctness. On the reliability front, this thesis leverages a novel fault model to diagnose and repair transistor failures at a fine granularity. In addition, the end-to-end diagnosis and repair procedures are cohesively designed to offer a graceful performance degradation with an increasing number of faults, losing less than a third of the processing power when compared to other state-of-the-art solutions. Finally, to prevent overheating, this thesis proposes a runtime-adaptable and power-tunable NoC design that dynamically power-gates the NoCs' components based on application's activity. As a result, the leakage power is slashed by up to 37% with less than 2% degradation in the application's performance. All the proposed solutions incur an area and power overhead within 5% of the baseline NoC, and are widely applicable as they require minimal changes to the underlying design. By offering **correct**, **reliable** and **power-aware** NoC operation in the face of adverse silicon trends, this thesis enables functional NoC implementations in future silicon generations that would otherwise have not been possible.

Table of Contents

List of Figures

Figure

List of Tables

Table

Abstract

As silicon technology evolves, chip multi-processor (CMP) and system-on-chip (SoC) designs are dramatically changing from limited, robust and homogeneous logic blocks to integrating billions of fragile transistors into complex and heterogeneous cores/IPs. This increased integration demands greater on-chip communication bandwidth, which, in turn, has compelled architects to design resource-heavy, complex and power-hungry on-chip interconnects, moving towards network-on-chip (NoC) structures. In addition, the waning reliability of silicon poses a great threat to these communication structures as they could potentially be a single point of failure. Further, the heterogeneity and fast time-to-market of upcoming computers makes it nearly impossible to thoroughly verify NoC architectures and optimize them for power at design-time. Failure of NoC architectures to meet correctness, reliability and power-budget requirements has detrimental effects on the runtime operation of NoC-based CMPs and SoCs. Therefore, runtime detection and reconfiguration mechanisms are becoming a key requisite to unlock the full potential of future CMPs and SoCs. Runtime mechanisms can overcome both functional bugs that escaped design-time verification and device failures due to an unreliable silicon substrate. Similarly, runtime reconfiguration solutions can also be leveraged to optimize the communication paths dynamically; particularly, to minimize power dissipation and prevent overheating of the NoC structures.

The solutions proposed in this thesis address these challenges to NoCs' runtime health by employing a reactive approach, *i.e.*, error detection followed by recovery. A reactive approach is a natural choice as it can address a wide range of unforeseen problems; particularly those arising during runtime operation. Further, the solutions proposed in this thesis provide integrated detection and recovery from errors. To attain temporal error isolation, an application's execution is partitioned into fixed-time monitoring windows, or *epochs*, during which distributed checkers, at each NoC router, monitor the traffic activity to detect anomalous behavior. If a failure is detected, a reconfiguration procedure is triggered at epoch boundaries to circumvent it. The reconfiguration procedure is implemented with lightweight and distributed hardware, and it utilizes broadcasts to synchronize the oper-

ations of all network nodes. The solutions are designed to be passive, lightweight and independent of the baseline design. For all solutions, the design complexity is kept at a minimum and the area overhead is within 5% with respect to a baseline NoC. In a nutshell, this thesis provides low-cost NoC-specific solutions that enable: **correct** behavior by avoiding functional bugs, **reliable** execution by circumventing faults and **power-aware** operation by averting overheating. The solution targetting functional correctness ensures correct NoC operation under all execution scenarios; the reliable NoC degrades $>3\times$ more gracefully than other state-of-the-art solutions, while the power-aware NoC design is able to reduce leakage power by up to 37% with less than 2% performance loss. The work presented in the thesis will enable designers to aggressively push scalability and time-to-market limits with respect to NoC design.

Chapter 1
Introduction

Digital computers, lead by advancements in microelectronics technology over the past few decades, have become pervasive in modern society. These advancements, for the most part, can be summarized by Moore's law, which states that the number of transistors integrable on a single chip doubles every 18 months [108]. Today, most people, as well as corporations and institutions, are relying on computers to handle their daily tasks more than ever before. Computers are the driving force behind phones we use to communicate with each other; they run banking and stock market systems for business and commerce. Even automobiles and airplanes are dependent on computer systems for enhanced efficiency, safety and user-experience.

The penetration of computing systems in human society has been driven by the constant efforts of computer architects to design innovative architectures that fully utilize the additional transistors made available by Moore's law. To this end, computer architects have traditionally prioritized constraints such as performance, energy efficiency and cost for the commercial success of their products. On a broad scale, the computing industry has seen three major design paradigms. Up until the early 2000s, the focus was on adding complex features to microprocessor architectures to boost single thread performance. However, the resulting processors exhibited unmanageable power dissipation density due to their many complex performance-oriented components working in tandem to execute the same task [62, 3].

To continue benefiting from technology scaling, the trend since early 2000s has been towards a growing number of simpler, mostly homogeneous power-efficient cores on-chip. The idea is to utilize the greater number of cores to better exploit thread-level parallelism. As a result of the increase in the number of on-chip entities that operate collaboratively, there has been a significant rise in inter-core communication demands. This trend has caused a paradigm shift from computation-centric to communication-centric designs. Recently, due to the faster growth in component count in comparison to the reduction in per-transistor power, multi-cores have also hit the power wall. Homogeneous application-

oblivious multi-core architectures, no matter how power-efficient, cannot be scaled further without operating portions of the multi-core system at throttled frequencies, or even completely switching some components off [37].

Moving forward, industry experts envision the use of additional silicon area available to deploy a wide variety of heterogeneous application-specific components as a way to overcome power constraints. This shift has led the transition of homogeneous CMPs to heterogeneous collections of IPs that resemble systems-on-chip (SoCs) in many aspects. Therefore, in addition to providing low-latency communication between the many components, an efficient on-chip interconnect should also adapt to the vastly varying applications that run on these diverse architectures. As a result, the complexity of the on-chip communication substrate has exacerbated.

Naturally, the transition from computation-centric to communication-centric design has rapidly sidelined traditional interconnects, such as simple buses, due to their limited bandwidth and poor scalability. Networks-on-chip (NoCs), characterized by highly concurrent communication and better scalability, have emerged as the *de facto* choice for on-chip interconnects. With efficient communication at the center of current and upcoming computer designs, NoCs have grown in importance and have attracted increasing attention from computer architects. However, similar to the processor designs of the past, the focus till now has been on increasing communication performance, occasionally focusing on energy efficiency. The ever increasing need for faster and more capable NoCs, along with the astronomical increase in the number and variety of cores/IPs they bind together, has caused an enormous growth in the complexity of NoC architectures and of the communication subsystem as a whole.

Technology experts predict future computers to be complex machines composed of several billions of minuscule and fragile transistors. At future technology nodes, they expect frequent errors throughout the lifetime of a system [115, 15] due to failing silicon devices. Even for existing silicon, large scale studies have shown error rates that are orders of magnitude higher than previously assumed [86]. As a result, CMP and SoC designs are transitioning from limited, robust and homogeneous logic blocks to complex systems integrating a vast sea of fragile transistors into heterogeneous cores/IPs. This waning reliability of silicon poses a challenge for on-chip communication reliability, since the communication hardware is typically not a redundant resource. Faulty communication can cause loss of valuable data, corruption of program's output, or even crash of the entire software application. NoC designs can make a successful transition to future fragile silicon generations only if they are equipped with capable and pervasive robustness mechanisms that protect them from failing. Providing these mechanisms is precisely the goal of this dissertation.

As the variety and quantity of heterogeneous components comprising future multi-cores/SoCs grows, the interconnect design methodology must also evolve. Current NoCs without any runtime reconfiguration support often fail to adapt to dynamic application environments. This can be particularly harmful from the perspective of power dissipation. Modern NoCs consume a significant portion of the on-chip power budget (as much as 30% of the entire chip power [103, 130]). If power is not accurately monitored and limited, it can lead to overheating emergencies. Power emergencies in turn can cause unnecessary throttling of other components on-chip, and consequently lead to performance loss. Increased power dissipation also translates to higher temperatures on chip, and it can considerably accelerate the transistor breakdown phenomenon [59]. Therefore, monitoring and limiting power consumption by an NoC, while still maintaining good performance, is of paramount importance to NoCs' runtime health.

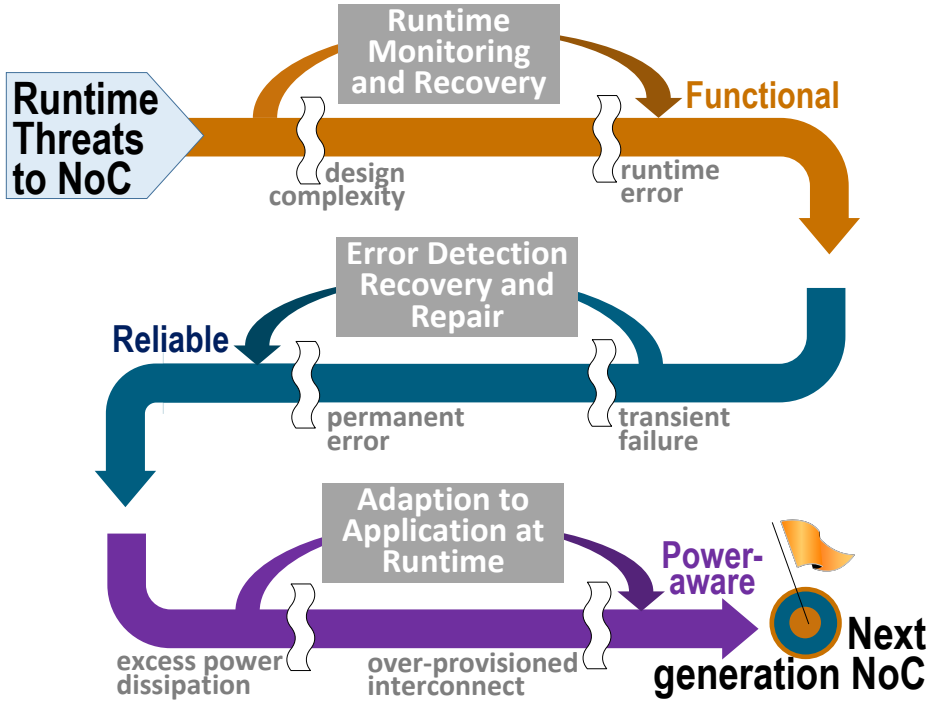


Figure 1.1 Overview of the NoC challenges tackled by the dissertation. This thesis develops the design methodologies to enhance the runtime health of NoCs of the future. It particularly focuses on: i) guaranteeing correctness under unmanageable design complexity, ii) providing dependable communication with unreliable silicon substrate, and iii) enabling adaptable interconnect designs to tackle excessive power dissipation.

In view of the challenges to NoC design that were described above, the current architectures and design methodologies that focus only on performance are no longer adequate. In addition, as the sole medium for on-chip communication when deployed, the NoC also becomes a single point of failure. Therefore, ensuring healthy NoC operation is of great

importance. Figure 1.1 provides a graphical overview of the thesis. Specifically, this dissertation identifies three new barriers that limit advancements of future computer systems from the perspective of NoC design: i) escalating design complexity, ii) waning reliability of silicon devices, and iii) excessive power dissipation. Overcoming these limitations is essential to tap the full potential of future semiconductor technologies. The dissertation provides novel solutions to overcome these barriers by developing NoC-specific design methodologies to equip the communication substrate with the capability to bypass these threats. In addition, the thesis describes runtime mechanisms to promptly detect and pinpoint issues while the system is in operation. These mechanisms allow the system to quickly react to a failure at runtime and adapt to bypass the components affected.

1.1 Network-on-Chip Basics

Continuous technology scaling has enabled computer architects to design larger and more complex CMPs and SoCs for improved performance. As a result, the corresponding increase in communication demands has led to an industry-wide shift towards scalable and efficient interconnects, typically a network-on-chip (NoC). NoCs leverage communication concurrency and resource sharing to provide a distributed and scalable solution. An NoC is a collection of *routers* connected together via *links (channels)* and organized in a certain *topology*. In NoCs, data is transmitted between communicating entities in units called *packets*, which are often further divided into smaller blocks called *flits*. Packets are injected into the network via a dedicated *network interface* (NI) that forms the interface between a core and a network router. Once packets are injected into the network, its routers provide packets with temporary buffering as they are routed to their respective destinations according to some routing protocol. The left side of Figure 1.2 shows an NoC with its routers organized in the popular *mesh* (grid) topology.

The right side of Figure 1.2 shows the micro-architecture of a simple NoC router. Packets received at the router are temporarily buffered at the *input ports*, while the routing logic determines the *output port* that leads to the correct destination. Once the correct output port is determined, packets request access to the *crossbar* that connects all input ports to all output ports. An *arbiter* is responsible for handling requests and grants to the crossbar. Once the packets traverse the crossbar, they are transferred to the input buffers of the downstream router via inter-router channels. The protocol governing the flow of packets or flits through the routers and channels is termed as the *flow control*.

Figure 1.3 depicts the micro-architecture of a more complex *virtual channel* (VC)

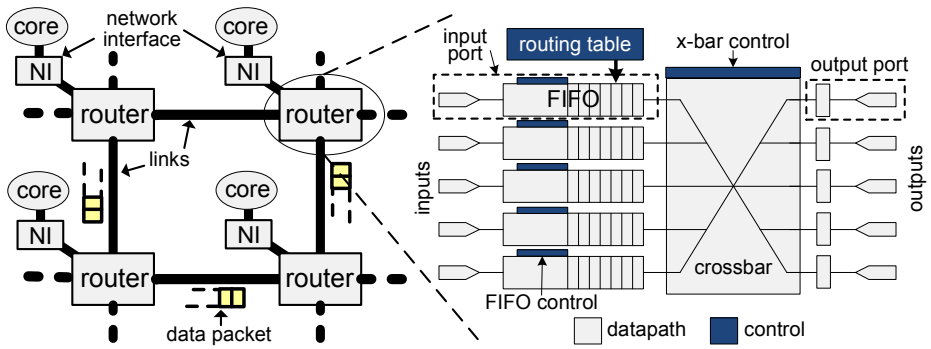


Figure 1.2 Network-on-chip basics. The left side of the figure shows a high-level diagram of an NoC organized in a mesh topology. The right side of the figure depicts the micro-architecture of a simple NoC router.

router. A VC router decouples the dependency between channels and input buffers, allowing multiple interleaved packet flows through the same channel. The primary idea is to deploy multiple buffers and *VC control* units at each input port, and allocate a buffer and a VC control unit to each packet flow before the packet enters the router's pipeline. A *VC allocation* (VA) unit is responsible for managing the buffering resources for these independent packet flows. Similarly, a *switch allocation* (SA) unit manages requests and grants to the crossbar in a VC router.

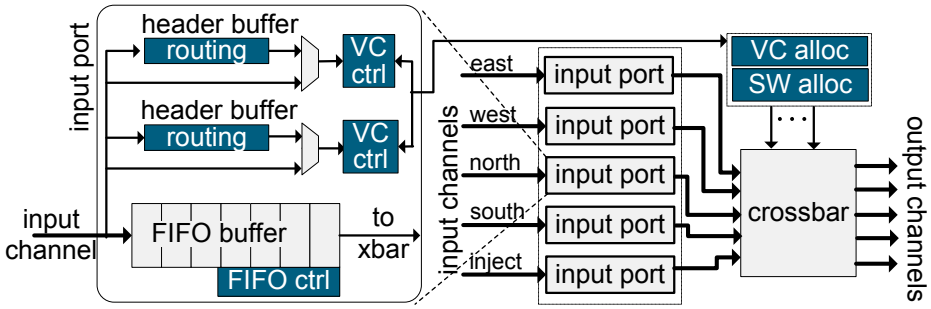


Figure 1.3 A typical virtual channel router. Virtual channels decouple the dependency between channels and input buffers to allow multiple interleaved packet flows through the same channel.

1.2 NoC Verification Bottleneck

The shift towards communication-centric designs has placed increasing demands on the interconnection fabrics, which, in turn, have become progressively large, distributed and complex. Therefore, a greater number of interactions must be verified at design time. This steep increase of the verification space can be perceived in the vast verification effort that

is invested in current microprocessor designs: up to 70% of human and monetary resources dedicated to microprocessor designs are channeled towards verification [58].

Failures that are caught early in the design process through pre-silicon verification techniques do not significantly impact the time-to-market. Therefore, the computer industry relies heavily on pre-silicon verification techniques to filter out most bugs. Pre-silicon validation efforts involve a combination of simulation-based verification techniques and formal methods. Simulation-based verification, though helpful in catching many easy-to-find bugs, is incomplete, as it cannot exhaustively test the countless different execution scenarios within an NoC. In contrast, formal methods, such as model checking, are complete but only effective in verifying small portions of the design, and cannot scale to verify end-to-end system-level correctness.

Despite massive industry efforts in verification, escaped functional bugs that manifest at runtime are a reality. Failures in the field have the most critical impact, which, in the absence of any runtime detection and recovery scheme, often lead to product recalls. In addition to costing huge amounts of money, recalls also adversely affect the competitiveness of the company's product. For example, the Pentium FDIV bug, which caused a divide-by-zero condition, costed Intel $475 million in recalls [77]. Figure 1.4 shows a study of the number of escaped bugs for various generations of Intel processors as reported in processor errata documents [60], [61]. It can be seen that there has been a steady increase in the number of bugs detected per month with each design generation, especially as the designs moved to dual-core (Core 2 Duo) and multi-core (Core i7) architectures. The growing complexity of inter-core communication is the leading cause for this rise, with 10%-13% of the reported design errors related to the communication system of dual and multi-cores. The growth in the bug-rate is expected to continue as the CMPs transition towards more complex NoC-based interconnects, with a lot of hard-to-find bugs contributed by the communication infrastructure.

Recently, the research community has started to explore runtime verification solutions where the system's activity is monitored at runtime and checked for correctness. Runtime verification can greatly reduce the impact of escaped design bugs. The associated cost, however, includes silicon area for runtime monitoring and recovery, dedicated design effort and a performance impact due to continuous monitoring activities. Therefore, naïvely safeguarding against all possible failures scenarios at runtime is prohibitively expensive.

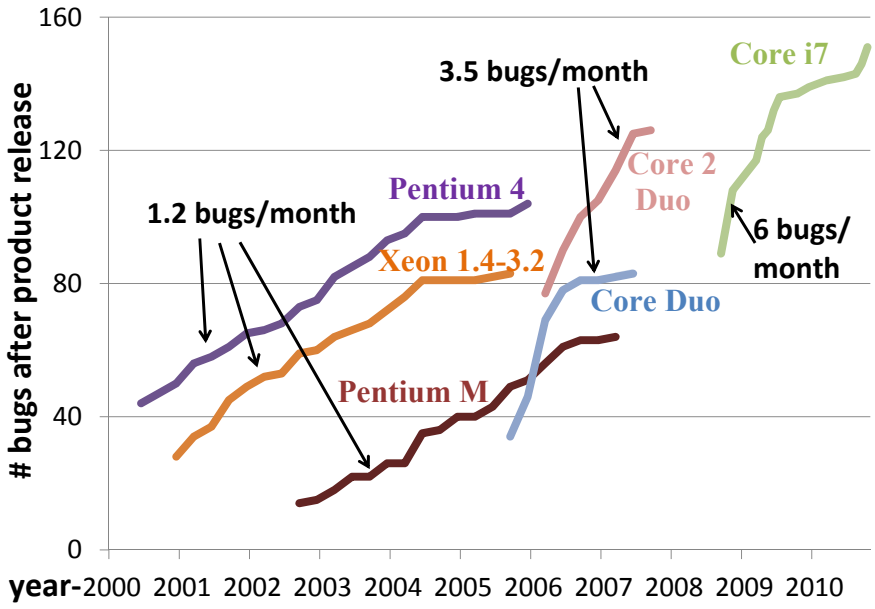


Figure 1.4 Bugs discovered after deployment for a number of Intel processors. The rate of escaped bugs increased drastically with the transition from uni-core to multi-core architectures. Thus, it can be speculated that a significant fraction of the bugs relate to the interconnect sub-system.

1.3 NoC Reliability Challenge

In today's landscape of extreme technology scaling, silicon transistors are highly susceptible to a wide variety of reliability threats. Specific sources of hardware failures include: particle strikes on silicon components, device degradation over time, and voltage and temperature fluctuations. Failures may manifest as one-time upset to device operation (transient or soft faults) or as a permanent hardware failure (permanent or hard faults).

Transient failures do not permanently damage the device functionality and typically can be overcome by re-execution of the erroneous computations. However, without proper detection, transient errors can silently corrupt data produced by digital systems. In addition, naïvely re-executing the entire application is often not possible, especially for mission-critical and real-time systems. The problem is further exacerbated by the increasing rate of these single-event upsets (SEUs), making the re-execution of entire applications an impractical solution.

Permanent failures, unlike transient faults, are persistent changes to the behavior of silicon devices. Permanent failures in the field are typically introduced by transistor wear-out phenomena: electromigration [47], negative-bias temperature instability [8], and time-dependent dielectric breakdown [117]. Commonly, permanent faults are overcome by disabling the malfunctioning component and leveraging redundancy in the design to continue execution. Naturally, the smaller the disabled circuit portion, the more graceful is the

performance degradation with each fault manifestation. A gracefully degrading solution, therefore, requires a fine-resolution fault detection and diagnosis, followed by reconfiguration around the faulty component.

Both transient and permanent faults can lead to undesirable consequences. Both fault types can cause silent corruptions to program's output or can lead to expensive service disruptions [107]. In addition, permanent failures can lead to complete loss of functionality due to their recurring nature [73], often forcing product recalls. As an example, the recent Intel Cougar-Point chipset was recalled due to a wear-out problem in the SATA controller: a recall that industry experts estimated to cost approximately $1 Billion [113].

In the context of NoC sub-systems, all components of the NoC are susceptible to soft-errors: from buffers and logic in the routers [34] to the on-chip links that are additionally affected by cross-talk and coupling noise effects. Further, as wear-out phenomena are accelerated by increased activity and higher temperatures, the heavily utilized on-chip networks become vulnerable to permanent faults. To make things worse, the network-on-chip is often the sole medium for on-chip communication, and may become a single-point of system failure, leading to critical loss of data and failure of software applications or of the entire system. Therefore, effective techniques ensuring reliable operation of NoCs in face of hardware failures are a critical requirement for overall system reliability.

1.4 Power-agnostic NoC Designs

In the past decade, designers have relied on multi-core designs to provide increased performance scaling at reduced power budgets. Unfortunately, adding more cores on a chip is no longer sufficient to provide meaningful performance improvements within reasonable power budgets [37]. Researchers have termed this phenomenon *dark-silicon* as it is no longer possible to run all components in a system at full throttle, and a significant portion of the chip must be completely switched off to avoid overheating.

Performance improvements in the dark-silicon era can be achieved by designing application-specific hardware components that accelerate certain types of computations. However, as the variety and quantity of heterogeneous components that make it into future computers increase, the interconnect that binds them together must also evolve to accommodate this shift in the design paradigm. In addition to the varying bandwidth requirements of heterogeneous components, the interconnect must also adapt to the needs of different applications that stress different portions of the network. The growing heterogeneity, the short time-to-market and the need for NoC designs to be generic enough to run a variety of appli-

cations, have rendered design-time optimizations difficult and ineffective. This often leads to over-provisioned NoC architectures that are designed to handle the most adverse communication patterns at all times. This shift in the design paradigm requires the development and deployment of novel interconnect architectures that can adapt to a wide range of communication demands at runtime, and avoid the alternative of over-provisioning hardware resources.

Particularly problematic is the trend of growing power-dissipation in NoC devices designed for worst-case traffic patterns, which today can consume up to 30% of the entire chip's power-budget. The problem is further exacerbated by leakage effects, causing power dissipation even during periods of low activity. For NoCs at 22nm, leakage can be a majority contributor to the total interconnect power [120], and the share of leakage power is expected to grow in future generations. The result is excessive power dissipation, even when an NoC is not under heavy use. However, heterogeneous workloads stressing only a portion of the design create opportunities for isolation of the lightly-used portions of the system and corresponding NoC region. The isolated components can be switched to a low-power mode using power-gating [57]. Such a solution requires application behavior monitoring and prediction, followed by runtime reconfiguration of components to optimize the NoC resources for the executing workload. By updating the reconfiguration trigger criterion, it is also possible to provide a variety of power and performance trade-offs, depending on the available on-chip power budget.

1.5 Threats to NoCs' Runtime Health

This dissertation aims at circumventing threats to NoCs' runtime health. In other words, we aim to detect and overcome spurious NoC behaviors that manifest at runtime and cannot be adequately addressed at design-time. Ensuring correct NoC operation in the presence of runtime malfunctions is challenging because: i) the faulty behavior can manifest unpredictably at any time or in any component, making prediction and prevention non viable options, ii) the continuous monitoring required to detect malfunctions results in substantial resource and performance overhead, and iii) complete recovery to a correct state after error manifestation is difficult without employing high redundancy in the design. Developing an effective solution without sacrificing performance or dedicating heavy silicon resources requires careful study of the characteristics of each type of failure. The characteristics of each failure's manifestation in silicon components is referred to as a *fault model*. In addition to an accurate fault model, ensuring runtime health also requires a thorough understanding

of the component under threat, *i.e.*, the NoC in our case. Note that NoCs are particularly critical because they are the only communication medium between interacting units in a chip and, therefore, a malfunctioning interconnect can potentially render the entire chip unusable.

In this dissertation we propose solutions to overcome three classes of failures that affect NoCs' runtime health. Specifically, we deem the following threats to NoCs' runtime health as major roadblocks to their evolution: i) **design errors** due to the increasing complexity of on-chip communication infrastructures, ii) **transistor failures** due to the waning reliability of transistors in the fragile silicon era, and iii) **power dissipation** beyond the cooling capacity of chips due to increasing transistor density. We believe that overcoming the issues above is the key to unlocking the full potential of communication-centric computation in future generations of computer systems. In the remainder of this section we discuss the characteristics of each failure class when they manifest in NoCs. We leverage these characteristics to design low-cost and high-coverage runtime solutions.

Design errors. As the name suggests, these are errors that are introduced accidentally into the hardware design, often due to its complexity and the challenges associated with a large and distributed development effort. Design errors cause the hardware not to conform to the specification and can have impacts ranging from minor delays in product launch to recall of the malfunctioning hardware. We particularly target design errors that have escaped all verification efforts before launch. Note that the computer industry spends massive effort in ensuring that released products are bug-free. Therefore, bugs that slip through the verification effort are typically associated with rare runtime occurrences. Even though these bugs rarely manifest, they can affect any NoC component in an unpredictable manner.

Design bugs can be overcome by re-execution of the application from a correct state, while operating the hardware in a barebone but guaranteed-to-be-correct mode. However, any hardware complexity introduced to protect the NoC against design errors has the potential of adding further complexity to the design. Taking these observations into consideration, our runtime verification solutions incorporate the following features:

- they are optimized for error coverage, rather than speed of recovery as design errors manifest rarely but in an unpredictable manner.
- they involve a lightweight recovery phase that operates the NoC in a guaranteed-to-be-correct mode till the error is circumvented.
- they add only simple and easily verifiable hardware logic to the NoC, simultaneously ensuring that the recovery operation does not interfere with the baseline micro-architecture.

Transistor failures manifest either as temporary corruptions of transistor output values or as permanently malfunctioning transistors. The former, called *soft-errors*, are similar in their manifestations to design errors. Soft-errors occur rarely and unpredictably, and they can be overcome by re-executing from an uncorrupted state. The latter, referred to as *permanent faults*, permanently damage transistor outputs. Therefore, the affected components are unusable for the rest of the chip lifetime. To continue operation, diagnosis of such permanent malfunctions is required, followed by a hardware reconfiguration step to disable the failed components. Similar to other failures, permanent faults also occur rarely but unpredictably. Permanent faults can also render the entire chip unusable if not addressed appropriately. The guiding principle for protection against permanent faults is *graceful performance degradation*. In other words, with each new fault, components should be diagnosed and disabled frugally so as to prolong the chip's lifetime. Naturally, overcoming permanent faults requires some resource redundancy in the design. Based on these characteristics, our solutions for protection against transistor failures incorporate the following features:

- soft-error solutions leverage a detection and recovery mechanism similar to our design error tolerance design.
- they provide fine-grained diagnosis and reconfiguration to frugally bypass permanently faulty components.
- solutions protecting against permanent faults are optimized for diagnosis accuracy rather than reconfiguration latency.
- they utilize redundancy built into the NoC architecture, rather than adding reliability-specific hardware.

Excessive power dissipation. Contemporary CMPs and SoCs incorporate a variety of application-specific components, and they run a variety of workloads exhibiting vastly different characteristics within and across applications. This trend compels designers to over-provision resources at design-time. Over-provisioning allows CMP/SoC designs to appropriately tackle the worst possible execution scenarios at runtime. However, over-provisioning of resources leads to excessive power dissipation.

NoCs, being a central component in CMPs and SoCs, also suffer from increased power dissipation due to over-provisioning. In addition, they consume a significant portion of the on-chip power budget. Note that a majority of NoC power dissipation can be attributed to transistors leaking power even when not in use. Finally, the available power-budget varies at runtime depending on the current chip operating conditions. Thus, we propose to develop

power-conscious NoC design solutions with the following characteristics:

- targeting the reduction of leakage power dissipation.
- performing dynamic optimization at runtime, based on workload characteristics.
- providing application-specific power efficiency, *i.e.*, greater power savings for less demanding applications.
- enabling flexible and tunable trade-offs between power and performance.

1.6 A Functional, Reliable and Power-aware NoC Architecture

With the growing relevance of communication-centric microprocessor designs, NoCs have emerged as the *de facto* scalable on-chip interconnect. However, focusing on performance and scalability of NoC designs alone does not completely address the challenges brought upon by silicon scaling. To this end, this thesis identifies and addresses the roadblocks arising from three key trends in extreme-integration silicon technology.

The first trend is the growing complexity of interconnect architectures due to the sheer number of collaborating entities and the never-ending quest for low-latency, high-bandwidth communication. This dissertation first provides evidence that design-time verification tools and methodologies are no longer adequate to validate all the complex interactions in NoC operation, and, as a result, bugs slipping into final silicon are a reality. It then proposes a verification approach that operates completely at runtime to detect and recover from errors in the field [2]. We further improve coverage and overhead of the solution by designing a complementary design- and run- time verification solution [90]. These solutions rely upon a lightweight and guaranteed-to-be-correct checking mechanism to detect execution anomalies, and they utilize interconnect reconfiguration for error recovery. The interconnect operates in a degraded and verified mode for the duration of the bug activity. Once the recovery is complete, the interconnect switches back to a full performance mode.

Similar to design errors, reliability failures also manifest non-deterministically at runtime. Functional bugs are triggered in rare execution scenarios, and they often disappear on re-execution. Similarly, soft-errors are single-event upsets, and they can be easily protected against by detecting and re-executing. Therefore, this dissertation proposes a soft-error protection solution [92] that is similar to our runtime verification approach. Handling permanent failures, however, is more involved and it requires: i) detection of the faulty

behavior, ii) diagnosis of the fault site, and iii) reconfiguration around the faulty component to prevent it from affecting future operations. To overcome permanent failures, this dissertation describes solutions that enable all these mechanisms. In our scheme, detection is facilitated by lightweight execution checkers, while information collected from the detection procedure is analyzed for accurate diagnosis [48]. We also propose a gracefully degrading NoC design that leverages a fine-grained fault localization mechanism and a frugal routing-reconfiguration solution [91]. Finally, we describe a solution to provide first-response operation while the reconfiguration procedure re-organizes the network around the fault [71].

The final part of this dissertation describes the design of a runtime-adaptable NoC architecture that is capable of addressing excessive NoC power dissipation resulting from growing heterogeneity, short time-to-market, and application diversity in modern microprocessors. Our solution [93] ensures healthy runtime operation of NoCs by avoiding overheating. Further, our NoC design adapts at runtime, eliminating the need for costly and time-consuming design-time optimizations. To this end, our solution leverages the distributed runtime monitors that are similar in philosophy to the ones used for functional and reliable NoC design. It further utilizes the routing and topology reconfiguration methodologies developed in the earlier chapters of this dissertation for the purpose of providing runtime adaptivity.

The solutions proposed in this thesis address challenges to NoCs' runtime health by employing a reactive approach, *i.e.*, error detection followed by recovery. In this manner, our solutions can address a wide range of unforeseen problems; particularly, design errors, reliability threats, and excessive power dissipation. Further, we adopt an integrated approach to guarantee NoCs' runtime health, addressing both detection and recovery from errors at runtime. A bird's eye view of the mechanisms proposed in this thesis is shown in Figure 1.5. Figure 1.5a shows the generic execution flow of the schemes proposed in this dissertation. We divide applications' execution into fixed-size time windows, or *epochs*, so as to provide temporal error isolation. During an epoch, distributed checkers at each network router monitor the traffic activity to detect anomalous behavior. If a failure is detected, a reconfiguration procedure is triggered at epoch boundaries to circumvent the detected failures. The reconfiguration procedure is implemented with lightweight and distributed hardware, and it utilizes broadcasts to synchronize the operations of all network nodes. Figure 1.5b shows the hardware additions at each router required to implement these capabilities: i) a failure monitor, ii) reconfiguration logic, and iii) a reconfiguration decision/control unit. The runtime monitoring and reconfiguration capabilities enable our NoC architecture to efficiently tackle threats to correct runtime operation.

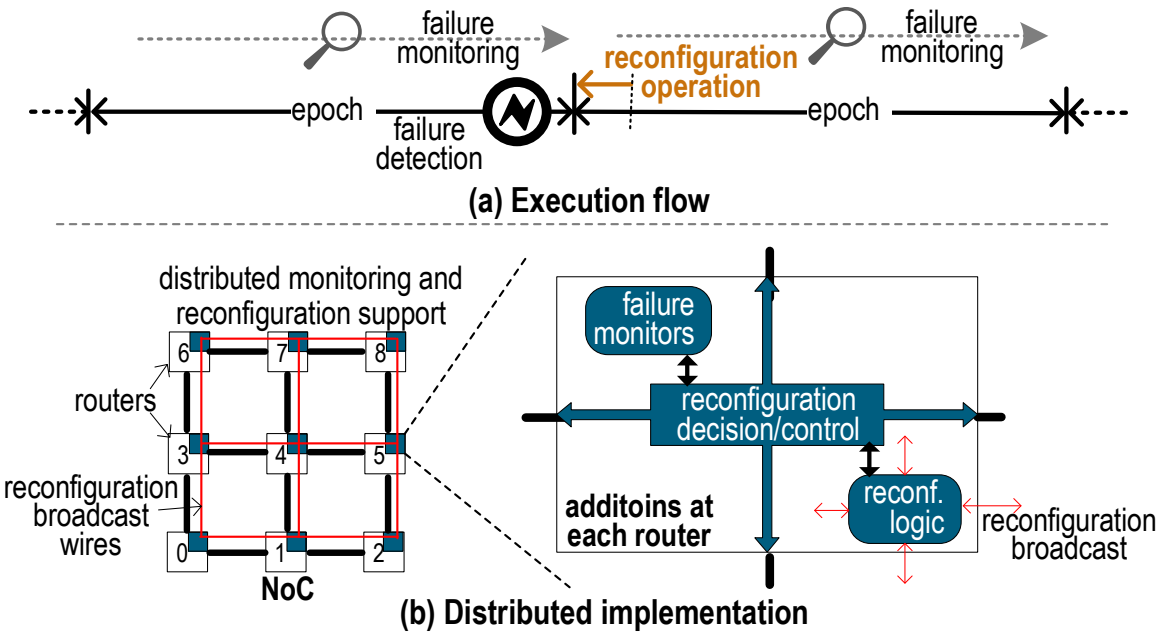


Figure 1.5 Overview of the solutions proposed in this dissertation. a) The figure shows the generic execution flow of our runtime mechanisms. Execution is divided into epochs, and the NoC is continuously monitored for anomalous behavior. A reconfiguration procedure circumvents the failures detected using synchronized broadcasts. b) The figure shows hardware additions at each router required to implement monitoring and reconfiguration capabilities. The added hardware is designed to be lightweight and distributed.

1.7 Dissertation Organization

The main body of this document is organized in three parts, each of which is dedicated to discuss solutions that provide protection against a specific runtime threat class affecting NoCs. This dissertation dedicates one chapter each to innovative NoC designs that provide: i) functional correctness in face of design errors arising from growing communication complexity, ii) resilient operation under both soft- and permanent- faults caused by the waning reliability of silicon, and iii) power-aware execution to keep NoCs' power density under check. The final chapter is dedicated to summarize the work in this dissertation and conclude the discussion.

Specifically, Chapter 2 discusses two solutions that ensure functional correctness in the presence of design errors. The first solution, called SafeNoC [2], employs a pure runtime approach to detection and recovery from an erroneous design state. The second solution, ForEVeR [90], leverages both design and runtime verification approaches to broaden the design error detection scope and reduce overheads. Chapter 3 discusses separate solutions for soft-errors and permanent failures. The soft-error protection scheme [92] exploits similarities between soft-error and design error manifestations, and it is built with minimal changes

to ForEVeR's infrastructure. Chapter 3 also details separate solutions for permanent fault detection, diagnosis, reconfiguration and recovery. The proposed detection and diagnosis scheme [48] leverages lightweight monitors to accurately pinpoint faulty components, while the reconfiguration scheme, uDIREC [91], leverages this fine-grained information to provide graceful degradation with faults. The recovery scheme, BLINC [71], provides a quick first-response on fault manifestations and allows uninterrupted operation of the network even during the occurrence of the fault. The work on tackling the power challenges of NoCs is described in Chapter 4. The proposed scheme leverages the execution information collected from distributed monitors to switch off sparingly used components. Finally, Chapter 5 summarizes and concludes the dissertation.

Chapter 2
Addressing Functional Bugs

Designing a functionally correct microprocessor is a challenging task due to its sheer complexity and the vast scale of modern designs. With the advent of the communication-centric design paradigm, much of this design complexity has shifted to on-chip interconnects, *i.e.*, typically a network-on-chip (NoC) for large multicore and SoC designs. NoCs are implemented as a network of routers and links, carrying messages to their destinations abiding to some routing protocol. The routers themselves often include advanced features, such as pipelining, speculation, prioritization, complex allocation schemes, *etc*. In addition, the routers are organized in a wide range of topologies, supported by complex routing algorithms. With these advanced performance features, it is a challenge to ensure correct functionality under all circumstances for the entire network. Functional bugs, or failure by a design to meet its specification, are filtered through various stages of verification, including techniques like simulation, emulation and formal verification. Even though computer architects channel significant resources to design-time verification, functional bugs slipping into the shipped silicon are a reality. Bugs manifesting in the field are the most devastating, necessitating permanent performance-degrading workarounds, and in extreme cases, forcing semiconductor companies to recall the affected product. This chapter presents the solutions to reduce or completely eliminate the cost of NoC functional bugs manifesting at runtime. We first propose a complete runtime verification approach that overcomes the majority of design errors in NoCs [2], followed by a hybrid pre-silicon and runtime verification approach that improves verification coverage and reduces the overhead of the runtime protection mechanism [90].

2.1 Inadequacy of Design-Time Verification

Pre-silicon verification efforts are used to ensure correctness using a combination of simulation, emulation and formal techniques. Simulation- and emulation- based verification

techniques, though helpful in catching many easy-to-find bugs, are incomplete as they cannot exhaustively test the countless different execution scenarios within a network. Even though incomplete, simulation and emulation are still the prime approach for catching the majority of design bugs early-on during the design phase. Design-time detection and correction provides three advantages. First, bugs caught at this stage can be fixed simply by modifying the design and cause only minor delays in schedule. Second, simple designs or small components of a complex design can be exhaustively tested to guarantee functional correctness. Finally, for complex designs, well designed test cases stress the most common usage scenarios to circumvent the majority of simple bugs. On the other hand, formal methods, such as model checking, are complete but only effective in verifying small portions of the design [43, 18]. Formal methods also suffer from state-space explosion problem for bigger designs. Therefore, formal methods typically do not scale to verify end-to-end system-level correctness properties.

Recently, the research community has started to explore runtime verification solutions where the system's activity is monitored at runtime after product deployment. Runtime verification can reduce the cost of design bugs that escape design-time verification by detecting their occurrence and preventing the corruption of network/processor state, loss of data and/or failure of the entire system. However, runtime verification solutions require dedicated monitoring and recovery hardware, which entails additional design and verification effort. Therefore, a naïve runtime verification solution results in substantial design, area and runtime overhead. To keep these overheads low and still be effective, a runtime verification solution should have the following properties:

- have no performance overhead during normal operation,
- cover a wide variety of failure scenarios,
- incur only a small area and power overhead, and
- entail minimal design and verification effort.

Runtime verification solutions have so far primarily focused on microprocessor designs [10, 83, 128]. In this chapter, we leverage runtime verification techniques to overcome functional bugs in NoCs, while trying to enable all the desirable properties stated above. Our designs are based on the observation that all frequently-occurring bugs are typically caught during pre-silicon verification: the bugs that slip through most often manifest very rarely. This observation allows our runtime verification methodologies to operate with low overhead detection and recovery hardware, yet they are capable of overcoming a wide variety of errors. We also strive for solution's designs that entail only minimalistic design modifications that are easily verifiable and mostly decoupled from the NoC hardware, and

incur no performance overhead in the absence of errors.

To evaluate the effectiveness of pre-silicon techniques for NoC verification, we performed an experimental study on a simple 2x2 mesh NoC design, with 5-port wormhole routers [28]. Since we are aiming for complete NoC correctness, simulation-based verification cannot be considered a suitable solution as it cannot exhaustively traverse an NoC's state space comprising tens of thousand of state elements. Therefore, we focused on formal verification techniques, and wrote properties as System Verilog Assertions (SVA) [1] describing high-level network behavior. We then tried to formally prove those properties using Synopsys Magellan [122]. However, the model checking-based formal verification engine was unable to complete the task due to the state explosion problem. Therefore, we restricted ourselves to properties that can be verified within a single router (component-level verification) to avoid state explosion. However, formal verification failed even in these scenarios if properties were not specified carefully. In general, it was hard to verify liveness properties and bounded properties with a large time bound.

In subsequent efforts, we were successful in verifying router-level specifications by breaking them down into simple sub-properties. An example of a property, "router does not drop any flits", is illustrated in Figure 2.1. For this particular example, we also decoupled the liveness requirement ("flit leaves eventually"), with the property we targeted to verify ("router does not drop any flits"). This analysis led to three important guidelines: i) verifying properties at component-granularity is tractable, in contrast to monolithic end-to-end verification, ii) broad-scope properties can be tackled by breaking them into simple sub-properties, and iii) formal verification is inadequate in proving liveness or large time-bound properties: the bugs that cannot be exposed in the process are good candidates for runtime detection and recovery. We used the observations above in developing the hybrid design- and run- time verification solution described in Section 2.3.

2.1.1 Overview of this Chapter

To counter the shortcomings of design-time verification, we propose novel runtime validation schemes tailored to NoC-centric multi-cores and SoCs. This chapter first presents SafeNoC in Section 2.2, a solution that provides end-to-end protection against NoC design-failures at runtime. SafeNoC leverages the observation that bugs that escape into the final silicon are rarely triggered, and a potentially slow software-based recovery mechanism has little overall performance impact. The chapter then introduces ForEVeR in Section 2.3, which guarantees complete NoC correctness, while keeping the area and reconfiguration overhead at minimum. ForEVeR leverages the observation that components that can be

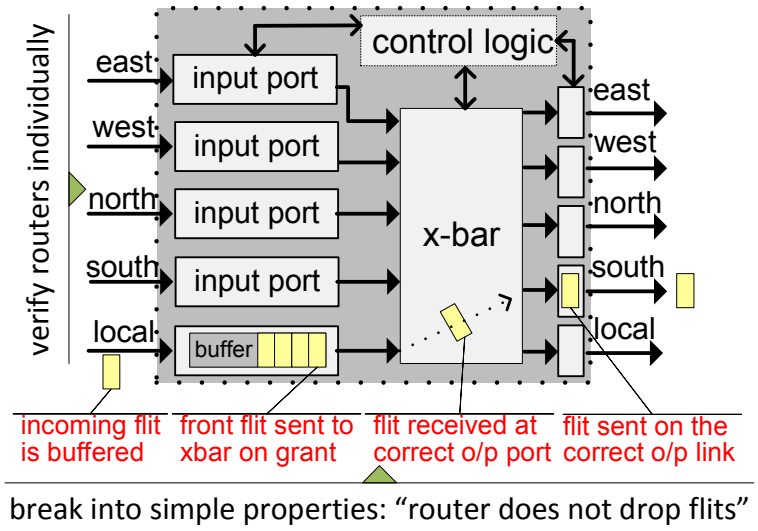


Figure 2.1 Formal verification guidelines. State explosion avoided by: i) verifying routers individually, ii) breaking specification into simple sub-properties and iii) checking liveness requirements at runtime.

fully verified during design-time do not need protection at runtime. Particularly, ForEVeR uses formal verification to guarantee correctness in small components, such as routers, while relying on a runtime detection and recovery mechanism to overcome network-level errors. The hallmark of both these techniques is that they apply minimal modifications to the baseline NoC, mostly conducting the verification activities in a decoupled fashion. Both solutions are independent of the topology organization, the router architecture and the routing scheme of the baseline NoC.

2.2 Runtime Verification with SafeNoC

SafeNoC is a end-to-end runtime detection and recovery mechanism to guarantee the functional correctness of the communication fabric in CMPs and SoCs. To this end, SafeNoC augments the existing interconnect with a simple and lightweight checker network that is guaranteed to deliver messages correctly. For each data message sent over the primary NoC, a look-ahead signature is transmitted over the checker network and is used to detect errors in the corresponding data message. Specifically, if a message is delivered without the corresponding signature present at the destination, SafeNoC triggers a recovery event. On error detection, all the flits in-flight in the NoC at the time of detection are reliably transmitted through the checker network to all the destination cores. There, a novel software-based recovery algorithm reconstructs the original packets from the erroneous flits, leveraging the

available signatures for the same. SafeNoC, thus ensures that all data reaches the intended destinations. Figure 2.2 shows a baseline CMP interconnect overlaid with the checker network. Both the checker router and the NoC router connect to the network interface to which two signature calculation units are also added. A generic 64-bit, 4-stage pipeline, 2-VC, 8-flit buffer per VC, wormhole router with inbuilt ECC functionalities is assumed as the baseline.

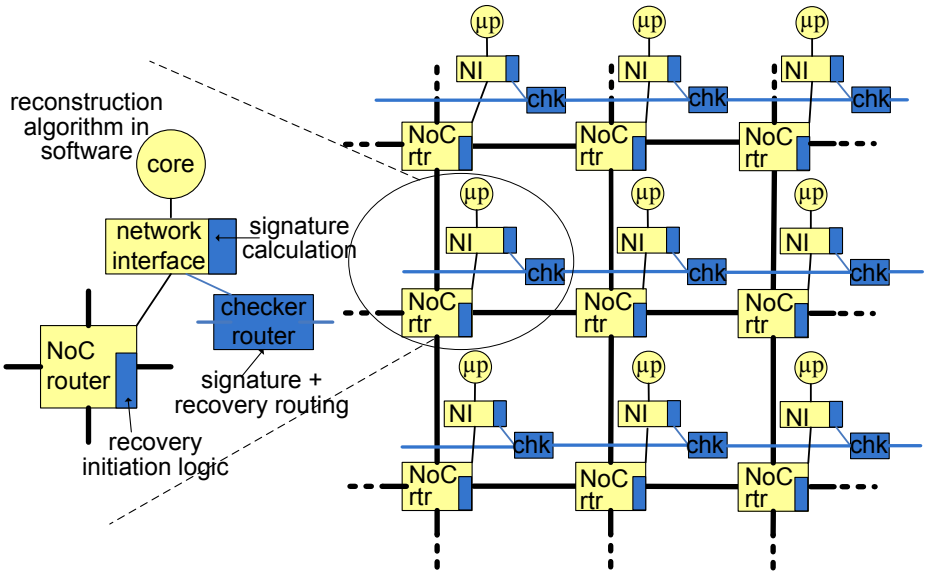


Figure 2.2 High-level overview of SafeNoC. SafeNoC augments the original interconnect with a lightweight checker network. For every data packet sent on the primary network, a look-ahead signature is routed through the checker network. Any mismatch between the received packet's computed signature and its look-ahead signature flags an error and triggers recovery.

SafeNoC is not limited to any particular interconnect topology or router architecture, and it can overcome a wide variety of functional bugs affecting the NoC. In addition, SafeNoC exhibits no overhead in the absence of errors, incurring a penalty of few milliseconds in the worst-case bug manifestation. Unlike traditional end-to-end 'retransmission-based' detection and recovery techniques for NoCs [85] that require large buffers for retransmission of clean data copies, SafeNoC requires small storage of compact signatures. SafeNoC leverages the infrequent nature of bug manifestations for trading-off recovery latency for a low hardware overhead design. SafeNoC's silicon footprint is a considerable improvement over the retransmission-based recovery scheme, as we will discuss in Section 2.5. Moreover, SafeNoC's hardware additions are simple and mostly decoupled from the existing interconnect hardware, and they are formally verified to be functionally correct.

The SafeNoC solution has separate error *detection* and error *recovery* phases. In the error 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 signatures.

2.2.1 Error Detection

SafeNoC's checker network incurs minimal area overhead, is formally verifiable and it delivers look-ahead signatures before actual data packets arrive through the primary network. A checker network that does not deliver signatures before their corresponding data packets would only introduce a performance penalty but would not affect correctness. SafeNoC's optimized checker network minimizes such cases by deploying a simple, single-cycle latency, packet-switched router, organized in a ring topology [67]. The details of the checker network design are presented in Section 2.4.

SafeNoC's bug model assumes that data within a packet's flits is protected by built-in ECC, and functional errors can either manifest by affecting entire packets, such as a deadlock, or by affecting individual flits within a packet, such as misrouting or re-ordering of flits. Within this bug model, 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. 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 signature computation provides low aliasing probability (3.05×10^{-5}), and it can be implemented with low area overhead. Each destination router maintains a timeout counter for every look-ahead packet it receives. If the signature from the newly received packet matches any of the look-ahead signatures, then this packet is considered to have been delivered correctly. In case of a signature time-out, an error is flagged and recovery is initiated.

2.2.2 Error Recovery

The recovery phase consists of four steps in the following order: *network drain*, *packet recovery*, *flit recovery* and *packet reconstruction*. First the *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 Figure 2.3a. The draining stage only leaves problematic packets and flits stuck within the network. The network then enters the *packet recovery* phase, which involves recovery of these problematic packets and flits. All primary routers remain active during this phase, except they do not service new packets. Only one primary router tries to recover packets at any time. To this end, a token is circulated through the checker network to determine the recovering router. When a router receives the token, it checks its input buffers to determine if there is a complete packet, in which case it is retrieved and sent over the checker network, as shown in Figure 2.3b.

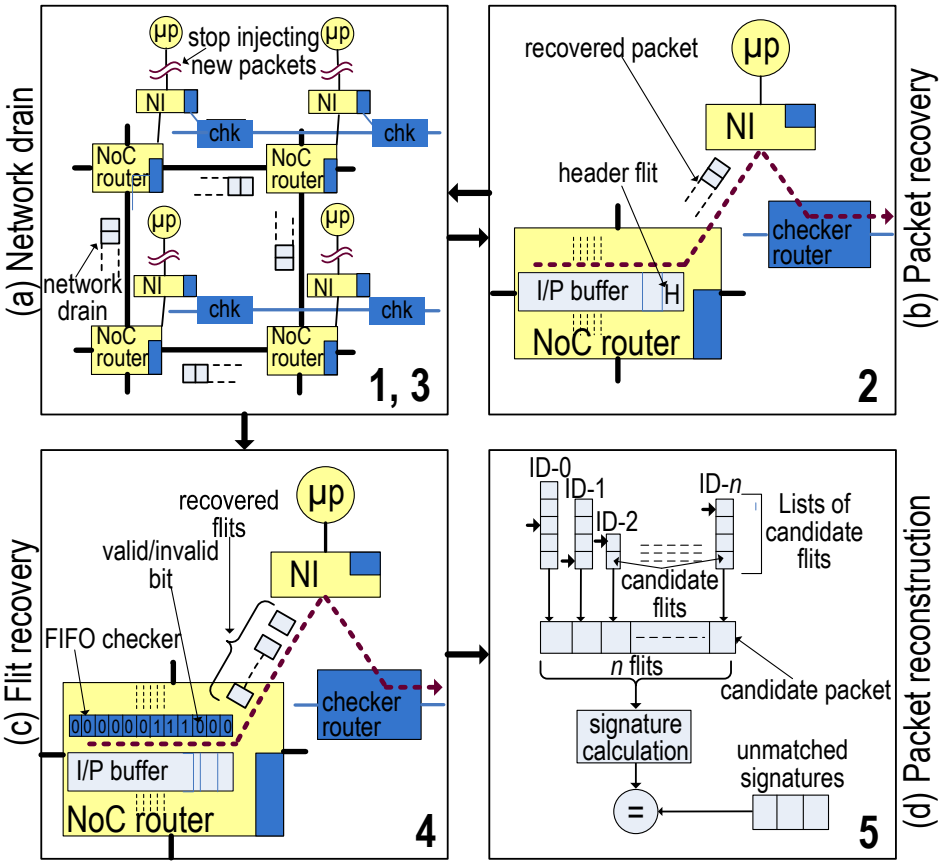


Figure 2.3 SafeNoC recovery process. Recovery proceeds in four steps. The last step is executed in software, while the others are implemented in hardware.

The next step, *flit recovery*, recovers stray flits from the network. A flit is considered stray if it is stuck in a router buffer or if it is delivered to the wrong destination. All stray

flits are candidates for the reconstruction process. Figure 2.3c illustrates this phase: a FIFO checker at every input buffer of each router tracks valid stray flits. Using a token-based protocol, each router, in turn, sends the stray flits over the checker network to all destinations in the NoC. During the last phase, *packet reconstruction*, the processor cores run a software algorithm to reconstruct the original packets using the stray flits (Figure 2.3d). 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. For each flit ID, a candidate is added to the set of current candidates. The current candidates are then assembled into a new packet and its signature is computed. If the signature matches any of the remaining look-ahead signatures, the packet is delivered to the application and all its flits are removed from the candidate groups.

2.2.3 Experimental Evaluation

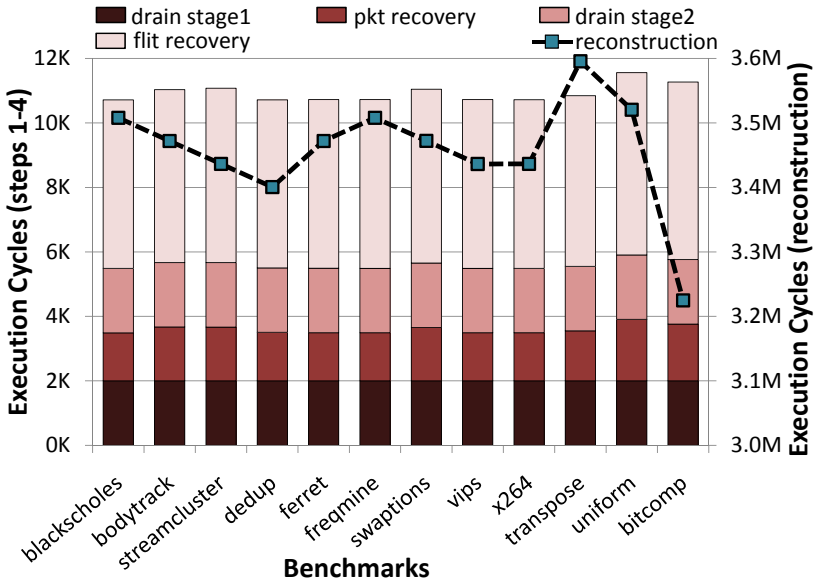
We modeled SafeNoC both in Verilog HDL and with a cycle-accurate C++ simulator. Using Synopsys Magellan [122], we formally verified the Verilog model of the hardware involved in recovery. The impact of recovery on performance was evaluated using the C++ simulator modeling a variety of functional bugs in the baseline system. The model was simulated with two different types of workloads: directed random traffic (16 flits per packet), as well as application benchmarks from the PARSEC suite [14]. The SimpleScalar [19] simulation infrastructure was leveraged to estimate the reconstruction algorithm's execution time. The network drain time was set to 2,000 cycles and the packet delivery timeout to 4,000 cycles.

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

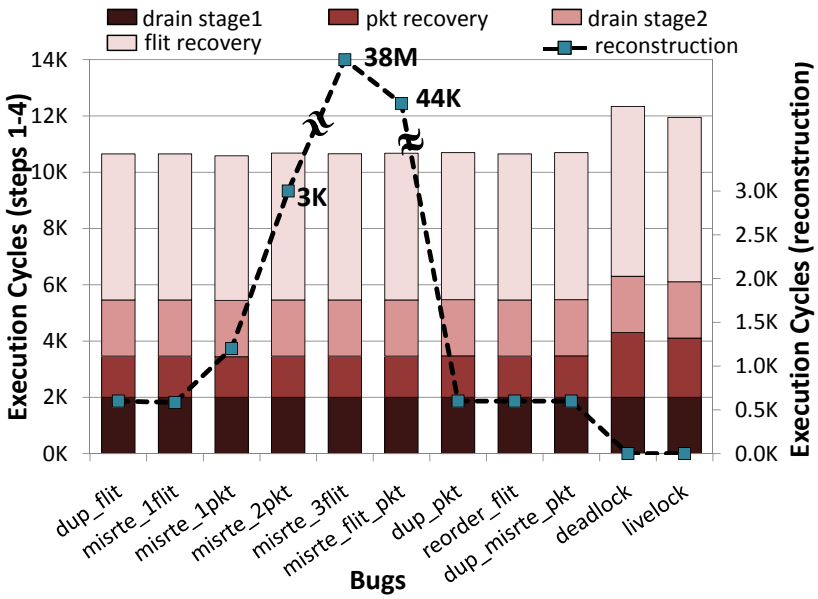
Table 2.1 Functional bugs injected in SafeNoC.

We injected 11 different bugs reported in Table 2.1 into the NoC infrastructure and

evaluated the detection and recovery time. One bug was injected per execution, and each simulation was repeated for all traffic patterns. The simulations were further repeated for 10 different bug trigger times and for 10 different random seeds for statistical confidence. With SafeNoC, all workloads complete, delivering all packets correctly to their destinations. Figure 2.4aa reports the recovery time required by each benchmark, averaged over all random seeds, activation points and bugs, for a total of 11,000 runs.



(a) Recovery time by benchmark



(b) Recovery time by bug

Figure 2.4 SafeNoC recovery time. Execution cycles for the first 3 steps of recovery (bars-left axis, network drain repeated twice) and for packet reconstruction (line-right axis).

On average, SafeNoC spends approximately 11,000 cycles in the first four steps of recovery. The drain time is a preset design parameter, at 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. Therefore, the performance overhead of SafeNoC is dominated by the reconstruction algorithm. In Figure 2.4b, we also 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 affects. For example, 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. The packet recovery time is constant for almost all bugs, at 1,473 cycles, required for the token to traverse all routers, with the exception of *deadlock* and *livelock*, where entire packets are retrieved from the primary network.

2.3 Complementary Design and Runtime Verification with ForEVeR

ForEVeR's approach is based on the insight that although formal methods do not scale to the complexity of an entire NoC, yet they can ensure component-level correctness, which, in turn, could greatly reduce the need for runtime bug detection and recovery. During system development, ForEVeR recommends a methodology for providing complete formal verification of the individual NoC routers. In addition, ForEVeR provides hardware additions to equip the NoC for monitoring and correcting the network execution at runtime. In the case that even individual network routers are too complex to be amenable to formal verification, ForEVeR proposes an additional runtime solution targeting specifically only those aspects of the routers' functionality that could not be verified during system development. The ForEVeR (**For**mally **E**nhanced **Ve**rification at **R**untime for NoCs) solution is independent of topology, router architecture and routing scheme. ForEVeR can detect and recover from a wide variety of functional errors in the interconnect, and it can ensure forward progress in the execution with no data corruptions. ForEVeR comes at a small area cost of 4.8% for an 8x8 mesh interconnect, while incurring a minimal performance impact only when an error manifests.

2.3.1 Methodology

As reported in [16], the functional correctness of an NoC can be organized along four high-level requirements. Three of them can be satisfied by guaranteeing their validity locally at each network router: *no_packet_drop*, requires that no packet is lost while traversing the network; *no_data_corruption* states that packets' payloads should not become corrupted while traveling from source to destination; finally, *no_packet_create* requires that no new packet is generated within the network (packets can only be injected from network's source nodes). If each individual router satisfies these properties, then they hold for the NoC system as a whole, since network links are simple wires and cannot embed functional bugs that corrupt, create or drop packets. ForEVeR leverages formal verification to ensure these three properties and maintain packet integrity. Finally, the last requirement (*bounded_delivery*), specifies that each packet is delivered to its intended destination within a finite amount of time and it ensures that there is forward progress in the transmission. This last requirement cannot be validated locally, since it affects the entire network. The runtime component of ForEVeR detects the violations of this property, and also provides a mechanism for error recovery.

To this end, Figure 2.5 shows a high-level overview of the hardware additions required by ForEVeR, and, in particular, it highlights the components required to enable *bounded_delivery*. Partially verified routers are connected together to form an NoC that is completed by detection and recovery logic. Runtime checkers and recovery logic are used at the router-level to protect complex router components against design flaws (if they cannot be formally verified at design time). In addition, the primary NoC is augmented with a lightweight checker network that is used to transmit advanced notifications to the monitors at the destination nodes. During recovery, the checker network is also used to reliably deliver in-flight data packets to their respective destinations. Note that each component of the router can be classified into i) verified at design-time, ii) monitored at runtime, or iii) providing advanced performance features to be disabled during recovery.

ForEVeR's checker network is similar in design to SafeNoC's checker network, and so is the philosophy of operating it concurrently for error detection and recovery. However, the checker network in ForEVeR is smaller as the notifications it carries do not contain any packet signatures. In addition, ForEVeR eliminates the need of the majority of SafeNoC's structures: i) signature buffers at destinations, ii) signature-packet comparators at destinations, and iii) a software-based reconfiguration. ForEVeR also greatly simplifies the recovery operation, requiring just a single packet recovery phase. Finally, ForEVeR protects against a wider variety of bugs, guaranteeing no dropped or corrupted flits, which is not the case with SafeNoC. In summary, ForEVeR leverages complementary verification

for a more holistic solution at cheaper cost.

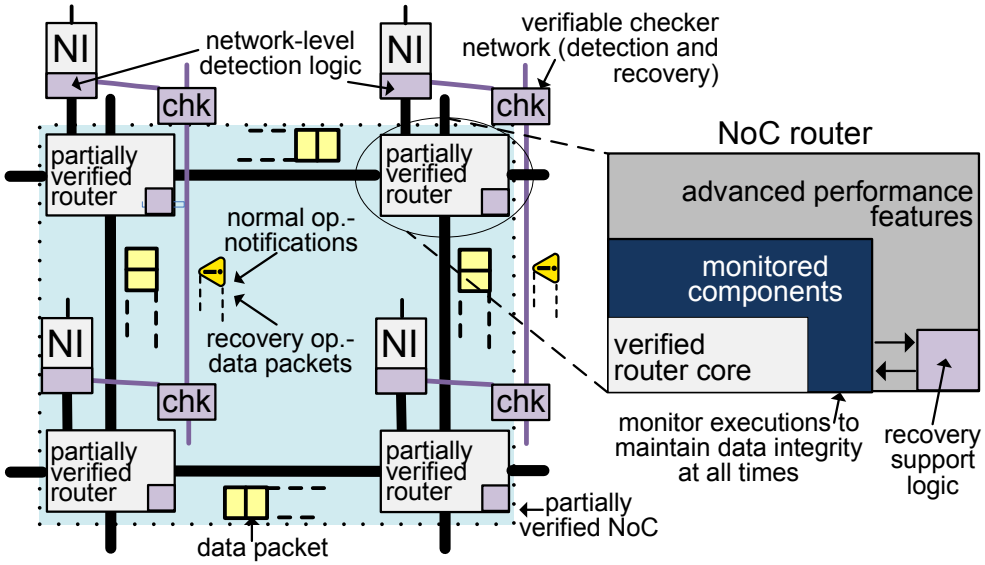


Figure 2.5 High-level overview of ForEVeR. A combination of router-level verification and runtime monitoring, and network-level detection and recovery ensures correct NoC operation.

2.3.2 Router Correctness

A correctly functioning router should ensure that each packet's integrity is maintained while it transfers within the router. This can be achieved by guaranteeing that routers do not drop any individual packet's flits and flit ordering from head to tail is preserved during the transmission in a wormhole fashion. The methodology to achieve router correctness is presented via an example of a fairly complex and generic 3-stage pipelined router that is input-queued and that uses virtual channel (VC) flow control, look-ahead routing and switch speculation. This verification methodology can however be generalized to any network and router architecture, as discussed in Section 2.3.5. The datapath components consist of input buffers, channels and crossbar, and are controlled by input VC control (IVC), route computation unit (RC), VC allocator (VA), switch allocator (SA), output VC control (OVC) and flow control manager. The datapath components are fairly simple and can be completely verified at design-time, while verification of the control components presents a greater challenge. First, we attempt a full formal verification of the routers, and in case some aspects of the router cannot be proven correct, we leverage a runtime solution to monitor and correct any functional bug in those components.

Router-level Formal Verification

The verification process can be efficiently partitioned into three sub-goals: i) ensuring that no flit is dropped (*no_packet_drop*), ii) showing that no flit is created or duplicated (*no_packet_create*), and iii) ensuring that packets maintain integrity as they travel through the router (*no_data_corruption*). For the first sub-goal, it must be verified that all valid flits received at input channels are written into valid buffer entries, that the buffers operate in a FIFO manner, and that each flit after gaining access to the output channel moves from input buffer to the output channel in a fixed number of clock cycles (depending on the router pipeline depth). To accomplish the second sub-goal, we verify that flits are not duplicated as they travel through the various stages of router pipeline (IVC, crossbar and OVC). We also verify that these stages do not create flits out of thin air. The third sub-goal encompasses the behavior of entire packets, rather than individual flits, ensuring that all body flits belonging to any particular packet should follow that packet's head flit in a wormhole order, as the packet traverses through the router's datapath. We pursued the formal verification of the baseline router design, using the structure described above and Synopsys Magellan [122], a commercial formal verification tool. Table 2.2 summarizes the sub-goals, how many properties were proven for each of them and how much computation time they required. Properties were described as System Verilog Assertions and verification executed on an Intel Xeon running at 2.27 GHz and equipped with 4GB of memory.

Correctness goal	**Property to be verified**	**#SVA**	**time(s)**
no_packet_drop	* valid flit written to buffer	4	90
(router datapath and control)	* buffer operates as FIFO	20	660
	* transferred from IP to OP channel	17	170
no_packet_create	* no flit/packet duplication at IVC	4	30
(control components)	* no flit/packet duplication at crossbar	1	10
	* no flit/packet duplication at OVC	2	10
no_data_corruption	* wormhole preserved (leaving IVC)	25	1,800
(complex component interaction)	* wormhole preserved (leaving crossbar)	5	350
	* wormhole preserved (leaving OVC)	5	200

Table 2.2 Organization of router's formal verification with ForEVeR.

Router-level Runtime Verification

ForEVeR provides runtime checkers to protect router components that could not be fully verified at design-time. Such components, typically, control the interaction between multi-

ple router activities [43]: VC allocator, switch allocator and buffer management. Note that the route control unit does not need additional protection because its activity is monitored as part of the network-level solution. If any router-level checker detects the occurrence of a bug, the router is reconfigured to a barebone mode of operation that can be completely formally verified. Recovery is performed by transferring the packets in this degraded mode of operation, which is guaranteed to be correct. After router-level recovery, a network-level recovery step is also performed. Below we describe ForEVeR's runtime router-level monitors for error detection in detail.

1. VC and switch allocator. A design error in the VC allocator may give rise to various erroneous conditions. However, some of these conditions are tolerable as they either do not violate router correctness rules, or they are detected and recovered by the network-level correctness scheme. Assignment of an unreserved but erroneous output VC to an input VC is an example of such an error as, in the worst case, it may only lead to misrouting or deadlock, which can be detected and recovered by our network-level correctness scheme. Starvation is another example that needs no detection or remedy at the router-level. Critical errors, *i.e.*, errors that threaten data integrity, arise when an unreserved output VC is assigned to two input VCs, or an already reserved output VC is assigned to a requesting input VC. This situation will lead to flit mixing and/or packet/flit loss. Similar to the VC allocator situation, a design flaw in a switch allocator may or may not have an adverse affect on ForEVeR's operation. To monitor VCs and switch allocators at runtime, we propose the use of an Allocation Checker (AC) unit, a simplified version of a unit proposed in [94] for soft-error protection. The AC unit is purely combinational and it performs all comparisons within one clock cycle. It simultaneously analyzes the state of VC and switch allocators for duplicate and/or invalid assignments. If an error is flagged, all VC and switch allocations from the previous clock cycle are invalidated. Flits in flight in the crossbar are discarded at the output. To avoid dropping flits during the invalidation/discard operation, an extra flit storage slot per input port is reserved for use during such emergencies. To implement this runtime monitor, VA, SA and crossbar units are modified to accept invalidation commands from the AC.

2. Buffer management. A design error in buffer management can lead to either buffer underflow or overflow. Input buffers can be easily modified to detect and refuse communication during an underflow, thus not loosing or corrupting any data. On the other hand, a hardware checker is used to detect buffer overflow errors. Additionally, each input port is equipped with two emergency flit storage slots. Upon receiving a flit when the corresponding buffer is full, the communicating routers switch to a NACK-free variant of ACK-NACK

flow control, that guarantees freedom from buffer overflows using a simple scheme. The emergency slots are reserved for flits in flight during this event. During this NACK-free flow control operation, a flit awaiting acknowledgement is re-transmitted every two cycles (round trip latency of the links). This scheme, though detrimental for performance, is extremely simple and can be implemented with little modification to the baseline buffer management scheme. In addition, the router operates in this simple and verified mode only during recovery, switching back to its high performance mode after recovery is complete. Note that, to safeguard against all errors, at most two emergency slots per input port are required and this storage can be implemented as a simple shift register. In addition, the cost of this extra storage is amortized across multiple VC buffers in a single input port.

2.3.3 Network Correctness

At the network level, it must be guaranteed that all packets are delivered to their intended destination within a bounded amount of time. Specifically, the network-level solution must detect and recover from design errors that inhibit forward progress in the network (deadlock, livelock and starvation) or cause misrouting of packets. To achieve this, ForEVeR augments the NoC design with a checker network similar in architecture to the checker network of the SafeNoC solution 2.2. Similar to SafeNoC, ForEVeR operates its checker network concurrently with the original NoC, providing a reliable fabric to transfer notifications and packets to be recovered. During normal operation, each packet transmitted on the primary network generates a corresponding notification over the checker network, directed to the same destination.

Network-level Error Detection

All design errors preventing forward progress result in packet(s) trapped within the network. Therefore, the detection mechanism should be capable of detecting such scenarios. Any unaccounted packet at destination will lead to a counter with an always positive value, and hence the detection mechanism will flag an error. Figure 2.6 illustrates the hardware implementation and execution flow of the detection scheme. The detection operation is organized into '*check epochs*' or time intervals of fixed length. The detection algorithm increases the counter at the local destination node for each notification received and decreases it for each packet received. In addition it stores in a separate register, reset at the beginning of each *check epoch*, whether a zero has been observed. Recovery is initiated if one or more network nodes have not yet observed a zero at the end of a *check epoch*. The

implementation requires a counter connected to both the primary and the checker network, a timer to track epochs and a zero-observed storage bit. Finally, note that design errors leading to misrouting of packets can be detected by analyzing the routing information in the header flit at the destination nodes.

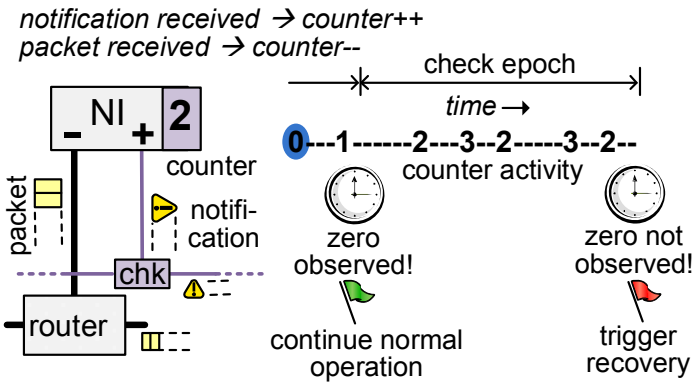


Figure 2.6 ForEVeR network-level (runtime) detection scheme. The counter at destination NI is incremented (decremented) upon notification (data packet) arrival, and recovery is triggered if zero is not observed during a *check epoch*.

Network-level Error Recovery

When an error is reported either by the router-level runtime monitors or by the network-level detection scheme, the NoC enters a recovery phase, consisting of a *network drain* step followed by a *packet recovery* step. The recovery phase is depicted graphically in Figure 2.7. During *network drain*, the network is allowed to operate normally to drain its in-flight packets, while no new packets are injected. At the end of this phase, which runs for a fixed time length, recovery terminates if all destinations have received all their outstanding packets. This situation indicates that recovery was triggered by a false positive detection, which can be caused due to inaccuracies in ForEVeR's detection scheme. The subsequent phase, *i.e.*, *packet recovery*, recovers all remaining outstanding packets. To this end, a token is circulated through all routers in the NoC via the checker network, and NoC routers can only operate when they hold this token. In addition, all routers are prevented from servicing new packets. When a router receives the token, it examines all its VC buffers sequentially to find packet headers. If a header is found, the corresponding packet is extracted and transmitted over the checker network. The token circulates through all routers retrieving packets from one router at a time. Retrieving all packets may require repeating the token loop through all routers, as certain packets may still remain in their respective buffers.

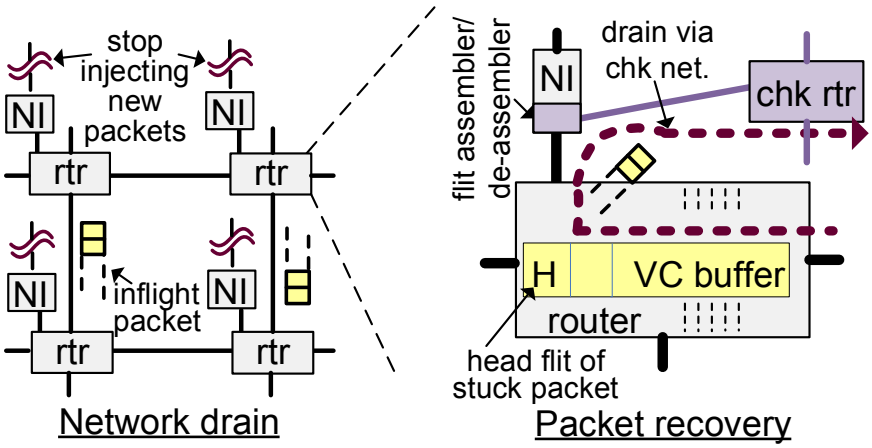


Figure 2.7 ForEVeR network-level (runtime) recovery scheme. All packet injections are suspended during the network drain phase, while remaining packets are recovered via the checker network during the packet recovery phase.

2.3.4 Experimental Evaluation

ForEVeR's experimental setup is similar to SafeNoC: Verilog implementation is used for formal verification and area estimates, while C++ simulator [28] is used to evaluate ForEVeR's operation. We injected 9 different design bugs into the C++ implementation of ForEVeR, described in Table 2.3. Bugs 1-6 are errors that inhibit forward progress, bugs 7-8 are misrouting errors, whereas bug 9 is an error that affects router operation while it is servicing a packet. One distinct bug is triggered during each execution of simulation traces, while varying the trigger time (5 trigger points, 10,000 cycles apart), the location of bug injection (10 random locations) and packet size (4, 6 and 8 flits). ForEVeR was able to detect all design errors with no false positives or negatives and correctly recover from them, executing all workloads to completion and delivering all packets correctly to their destinations. Each recovery entailed an execution overhead, due to *network drain* and *packet recovery*. During *network drain*, the primary NoC was allowed to drain for a fixed period of 500 cycles, a parametric value that we set by simulating the draining of a congested network. Table 2.3 reports the additional average *packet recovery* time incurred for each bug, averaged over all benchmarks, packet sizes, activation times and locations.

Note that routing errors are caught at incorrect destinations, while errors affecting router operation are uncovered immediately by the distributed hardware monitors. On average, ForEVeR spends approximately 2,633 cycles in packet recovery for each bug occurrence. This value is primarily affected by the number of packets that must be recovered; thus bugs affecting a large portion of the network, such as an entire port (*VA_port_strv*), take more time to recover than bugs that influence smaller portions, such as only one VC (*VA_vc_strv*).

Similarly, *deadlock* errors that may affect many packets, require the largest recovery time. We observed the worst case recovery time of 30K cycles across all our simulation runs, including all packet sizes and both uniform and application traffic. A key aspect of ForEVeR design is that it incurs no overhead during normal operation, spending time in recovery only on bug manifestation. Therefore, it can afford a longer recovery time as design bugs manifest infrequently.

Bug name	**Bug description**	**recovery time**
deadlock	some packets deadlocked in the network	4,821 cycles
livelock	some packets in a livelock cycle	3,084 cycles
VA_vc_strv	input VC never granted an output VC	2,827 cycles
VA_port_strv	no input VC in a port granted output VC	3,055 cycles
SW_vc_strv	one input VC never granted switch access	2,123 cycles
SW_port_strv	no input VC in a port granted switch access	2,490 cycles
misroute1	one packet routed to a random destination	1,724 cycles
misroute2	two packets routed to random destinations	1,810 cycles
router_bug	hardware monitors in routers detect a bug	1,764 cycles
average		2,633 cycles

Table 2.3 Functional bugs injected in ForEVeR and average packet recovery time.

Bugs that typically escape into production hardware are extremely rare corner-case situations buried deep in the design state space that were not uncovered by extensive pre-silicon and post-silicon validation efforts. Hence, it is safe to assume that these bugs are extremely infrequent as they have escaped months of verification efforts. Therefore, even though more than 100 bugs were discovered in the latest Intel chips (Figure 1.4) after production, released bug patches (mostly software-based) do not lead to significant slow down in overall performance. In perspective, ForEVeR's recovery penalty of 2.6K cycles on average is insignificant even for a unrealistic bug rate of one error every 5 minutes. For a 1 GHz NoC, exhibiting an error rate of one error every 5 minutes, this translates to a negligible performance penalty, less than one hundred millionth (10^{-8}).

To closely study the relationship between recovery time and number of flits recovered via the checker network, we injected a varying number of packets in the NoC and prevented them from ejection at the network interfaces. The network-level detection scheme flags an error after the second epoch due to un-accounted primary network packets at destinations, thus triggering a network recovery. Concurrently, we noted the time required to drain all stuck packets through the checker network. Figure 2.8 plots our results for varying packet sizes, reporting *packet recovery* time vs. number of extracted flits. As seen from the figure, *packet recovery* time varies almost linearly with the number of stuck flits, requiring less

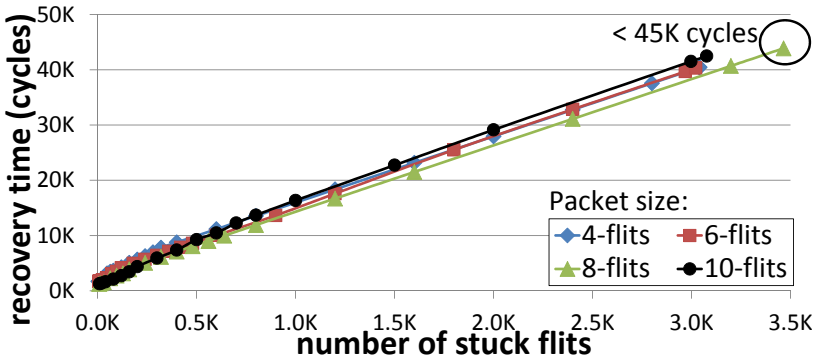


Figure 2.8 ForEVeR's packet recovery time. ForEVeR's recovery overhead increases almost linearly with the number of flits stuck in the primary NoC that must be transmitted reliably through the checker network. Worst case packet recovery time of 45K cycles is observed in this limit study.

than 45K cycles, even in the worst case. Note that in this artificial scenario, we are intentionally jamming the main network with many more flits than usual network occupancy at any instant. Thus, this serves as a limit study, and, in practice, ForEVeR's *packet recovery* time is limited by 30K cycles, as seen by our results on recovery from design bugs.

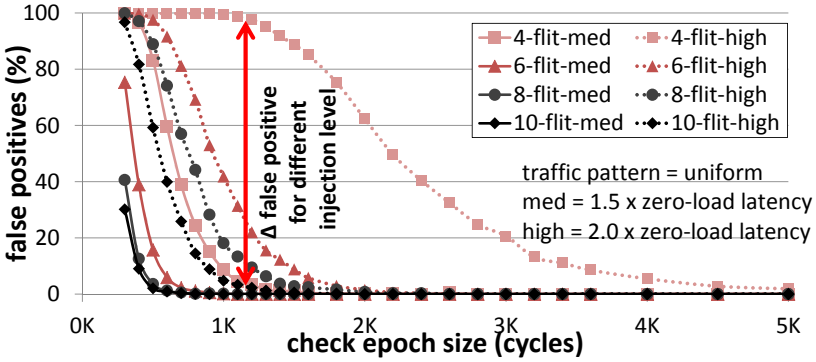


Figure 2.9 False positive rate vs. *check epoch* size, for various packet sizes. The false positive rate drops rapidly with larger *check epochs* and decreasing network load.

False Positives

False positives occur when an unnecessary recovery is triggered in absence of a bug occurrence, and they are due to inaccuracies in the runtime monitors. Note that, a false positive in the detection mechanism does not affects the network's correctness but only its performance. The false positive rate of the detection scheme depends on the duration of the *check epoch*, relative to traffic conditions. Note that false positives are triggered when the destination counter is non-zero for an entire *check epoch*; hence a heavily loaded network will

trigger more false recoveries as unaccounted notifications accumulate at destinations while their corresponding packets are being delayed due to congestion in the network. Intuitively, a longer *check epoch* will reduce the false positive rate by allowing more time for packets to reach their destinations. Figure 2.9 shows the decrease in false positive rate with increasing *check epoch* size. The false positive rate drops to a negligible value beyond a certain *check epoch* size ($Epoch_{min}$), whose value depends on network load. Additionally, a heavily loaded network exhibits a higher false positive rate than a moderately loaded network, and hence a heavily loaded network requires a larger $Epoch_{min}$ to practically eliminate all false positives. Extensive simulations indicate that $Epoch_{min}$ rises to intolerable values only when the network is operated at loads well past its saturation. However, NoC workloads are characterized by the self-throttling nature of the applications, which prevents them from operating past saturation loads [87].

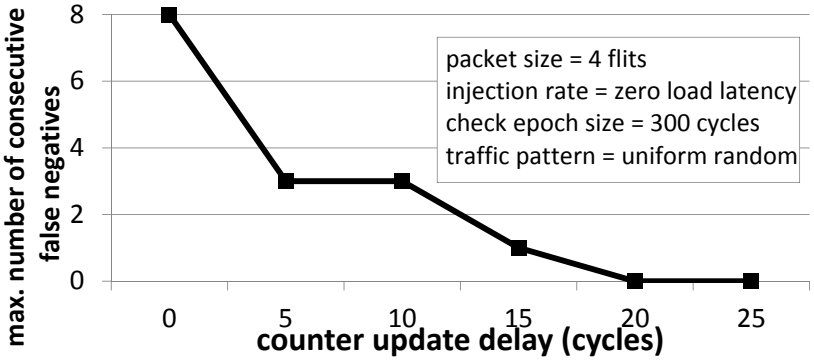


Figure 2.10 False negatives vs. primary network offset for *check epoch* length of 300 cycles, low network load and 4-flit packets. False negatives decrease with increasing primary network offset.

False Negatives

False negatives might cause an error to go undetected for a few epochs. But, since no loss of flits/packets is guaranteed, the data would eventually be delivered in an uncorrupted state to the correct destinations upon error detection. To avoid false negatives in the detection scheme altogether, the checker network is constrained to deliver notifications before the corresponding data packets arrive via the primary network. In ForEVeR's evaluation system, the checker network almost always delivers notifications ahead of data packets, except for very low latency situations, where primary network packets take shorter routes through the primary NoC, while notifications travel longer routes in the ring-based checker network.

To counter these cases, the updating of the monitor counters can be delayed by an amount determined by the maximum latency difference between primary and checker network at zero load (we call this value *counter_update_delay*). We ran low latency simulations using uniform traffic with a small packet size (4 flits) and a *check epoch* of 300 cycles. With this setup the primary network is only lightly loaded, and hence it has a greater chance of creating false negatives. Figure 2.10 plots the maximum number of false negatives observed over 10 different seeds for different *counter_update_delay* values. Note that the rate of false negatives falls quickly and are completely eliminated at a delay of 20 cycles or greater.

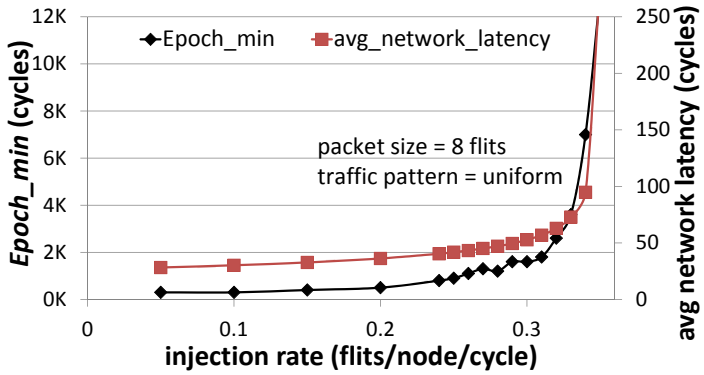


Figure 2.11 $Epoch_{min}$ **and latency with increasing network load** for uniform traffic. $Epoch_{min}$ is within tolerable limits for all but deeply saturated networks.

Optimal Epoch Length

To calibrate the *check epoch*, we ran rigorous simulations using both uniform random traffic and PARSEC benchmarks. After operating ForEVeR normally for a preset length of time, a random primary network packet is dropped to emulate the impact of an error in the primary network; the false positive and negative rate for a range of *check epochs* is then calculated. Figure 2.11 plots $Epoch_{min}$ (necessary to minimize the false positive rate) and the average network latency as network load is varied, under uniform network traffic. $Epoch_{min}$ exhibits a slow increase with rising injection rate up to network saturation, and a steep rise afterwards. From the plot, a worst case $Epoch_{min}$ of 7K cycles is sufficient to eliminate all false positives when the network is in deep saturation, operating at an average latency of about 4 times the zero-load latency. With similar experiments on PARSEC benchmark traces, we observed that the *check epoch* of length 600 cycles is sufficient to eliminate all false positives.

Verification of ForEVeR's Recovery

All components involved in the detection and recovery processes must be formally verified to guarantee correct functionality. Detection leverages the checker network and the counting logic in the network interfaces. In addition to the interface between primary and checker routers for packet draining, the recovery operation also uses the checker network.

To verify the correctness of the checker network, we need to show that it delivers all packets to their intended destination in a bounded amount of time, as discussed in Section 2.3.1. We partition this goal into four properties: *injection*, guaranteeing correct injection of packets into the network; *progress*, ensuring packets advance towards their intended destinations; *ejection*, proving timely ejection of packets; and *data_integrity*, ensuring that data remains uncorrupted throughout.

As discussed earlier, during recovery, the NoC routers operate in a barebone mode with all complex hardware units disabled, thus making the verification task much less challenging. To ensure correct recovery, we have to verify that routers fairly take turns in retrieving valid packets from their respective buffers. To this end, we check the following aspects: i) fairness and exclusivity during extraction (*fairness*) to guarantee that routers take turns in transmitting packets on the checker network. ii) We also verify that complete packets are extracted (*complete_packet*), emptying the buffer completely (*buffer_empty*). We also check that iii) only valid packets are recovered (*valid_packet*). Table 2.4 reports the time required to prove these verification goals using our setup.

checker network correctness		**recovery operation correctness**	
verified property	time(sec)	verified property	time(sec)
injection	8	*fairness*	15
progress	156	*complete_packet*	10
ejection	86	*buffer_empty*	46
data_integrity	10	*valid_packet*	29

Table 2.4 Formal verification of ForEVeR's network-level detection and recovery.

Area Results

A central goal in designing ForEVeR is to keep silicon area at a minimum. The amount of hardware required to implement router-level correctness varies with the designer's ability to verify different router components as formally verified functionalities need no protection at runtime. Thus, we present the area overhead for network-level and router-level correctness separately. Table 2.5b reports additions for network-level correctness, indicating a 4.8%

area overhead over a primary network router. The overhead is due to recovery support at each router, contributing 1.7%, and network interface additions and checker router, which, combined, are responsible for the remaining 3.1%.

Table 2.5a reports the overhead for ForEVeR's router-level hardware monitors and reconfiguration hardware, accounting for 9.2% additional area over the baseline router. Recovery support for router-level correctness costs 7.8%, whereas the monitoring hardware results in 1.4% overhead. In our framework, we were able to formally verify the baseline router completely, and hence we only incurred the network-level area cost (4.8%).

(a) router-level correctness		
design	**area** (μm^2)	**%**
baseline router	77,723	100.00
recovery support	6,071	7.81
monitoring	1,053	1.35
overhead	7,124	**9.16**

(b) network-level correctness (per router)		
design	**area** (μm^2)	**%**
recovery support	1,300	1.67
NI additions	1,550	1.99
checker router	845	1.09
overhead	3,695	**4.75**

Table 2.5 ForEVeR area overhead.

ForEVeR leverages formally verified components within the router to recover from design errors to keep the overhead low when compared to purely runtime verification techniques [85, 2]. Without these verified components there would be a need for substantial extra hardware. Specifically, in case of an error, a full runtime solution would require storage to duplicate all the in-flight data and retransmit that using the same unreliable network or transfer that over a secondary network that is guaranteed to be correct. We present the comparison against runtime schemes in Section 2.5.

2.3.5 Generalization

Generalization to other NoC Designs

Both the detection and recovery schemes of ForEVeR can be generalized to any NoC design/architecture. ForEVeR is agnostic to NoC topology and the routing algorithm employed, as long as the checker network, used both during detection and recovery, can adapt to consistently deliver notifications ahead of time. To this end, checker network's performance can be tuned to the needs of the baseline network by various mechanisms, such as increasing bandwidth or bundling notifications together before transmission. For the baseline design, a low bandwidth checker network sufficed to deliver notifications as required.

Generalization to other Router Designs

ForEVeR can be generalized to all mainstream router designs, as they have similar underlying structure, where control components manage the flow of data through the data-path components. ForEVeR's hardware monitors for router-level detection provide protection to router components that handle complex interactions such as flow control and resource allocators. Although the baseline router implementation is completely formally verified in our experiments, we designed these generalized hardware monitors (flow control checkers and allocation comparators) to be able to extend ForEVeR's detection scheme to more complex router designs that may be outside the scope of formal verification. As for ForEVeR's recovery scheme, only guaranteed basic router functionality is required to safely salvage packets from the routers. To ensure this, basic router components (input ports, buffers, arbiters, crossbar) required for bare-bone functionality are formally verified. These components are common to all router architectures, while many of the other router features that are design specific tend to be performance oriented: these type of features are disabled during recovery.

2.4 Checker Network Design

This section discusses the design principles behind the design of the checker network for ForEVeR. The same design principles apply to the checker network design for SafeNoC, and hence it is not discussed separately. A suitable checker network should have three main properties: i) it should be formally verifiable; ii) it should provide low latency transmission; iii) and finally, it should incur a low area overhead. A formally verifiable network should present a simple router and network architecture, and a simple routing algorithm. In contrast, a naïve solution to consistently deliver notifications before their counterpart data packets arrive through the primary network, may require complex router and network designs that are area-intensive and difficult to verify. Fortunately, exploiting the nature of traffic flowing through the checker network (*i.e.*, notifications), we are able to design a network satisfying both these conflicting requirements. Note that these properties are not a strict requirement for the correctness of our solution. However, not meeting them leads to additional costs in area, development time and performance. For instance, a checker network that does not deliver notifications before their corresponding data packets, may result in an error going undetected over multiple epochs. In such a scenario, error detection latency is affected, but NoC correctness is still guaranteed. The checker network, therefore, is tailored to the characteristics of the primary network so to minimize the occurrence of

such cases.

The checker network is used exclusively to transmit notifications to their destinations. The notifications contain no data, only including destination address (6-bits for a 64-node NoC) and a valid bit. Having to transmit only packets of uniform and small sizes presents many advantages that can be leveraged to design a simple and efficient NoC architecture. First, packet-switched routers, without large buffers or many wires between routers, can be utilized. This eliminates the need of complex switching techniques for buffer and/or wire cost amortization, including circuit switching, store-and-forward, virtual-cut through, wormhole and virtual channel switching, which all require a substantial amount of control and book-keeping hardware for proper functioning. Additionally, such a simple router design can be implemented to have single-cycle latency, as we discuss later in this section. Second, the small fixed-size packets require no packetization hardware at the network interfaces (NIs) and fully utilize the available bandwidth on packet switched networks (no bandwidth fragmentation). Finally, since the notifications are not stored at destinations, multiple notification can be ejected simultaneously, without requiring any additional buffering.

Based on these observations, our checker network should be packet-switched, with channel/buffer allocation and traffic transmission performed at packet granularity. To further simplify the design, we chose a ring topology for the checker network, leveraging a simple, single-cycle latency, packet-switched buffer-less router, based on the solution proposed in [67]. All the nodes in the network are connected in a bidirectional ring. A predetermined allocation ensures that there is no contention for network resources in routing a packet from source to destination; therefore, pipeline registers at the router inputs are sufficient to ensure loss-less transmission. To this end, once a packet is injected into the network, it has priority over other packets that are trying to enter the network, and thus the packet is guaranteed to make progress toward the destination. For each packet, a decision is made to inject it in one ring or the one in the opposite direction based on minimal routing distance. Each packet, once in the network, keeps moving forward through the same ring until it is ejected at its destination. Two packets traveling on separate rings may try to eject at the same node in the same cycle, causing contention for the ejection port. This is resolved by providing a separate ejection port for both ring directions at each node. The valid bit from ejection ports connects to the detection counter, while the destination ID is discarded at ejection. The checker router architecture is shown in Figure 2.12, and the checker network normal operation is shown in Figure 2.15a.

Ring topologies suffer from higher hop-counts when compared to common topologies like meshes, but in our system this is mitigated by three factors: i) the checker routers can

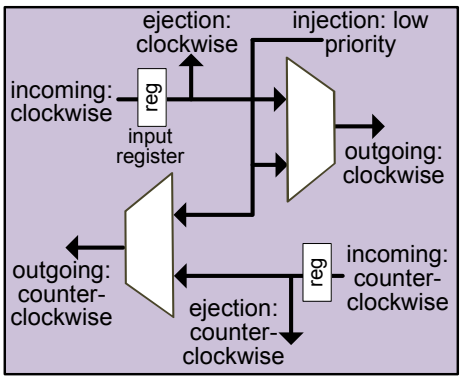
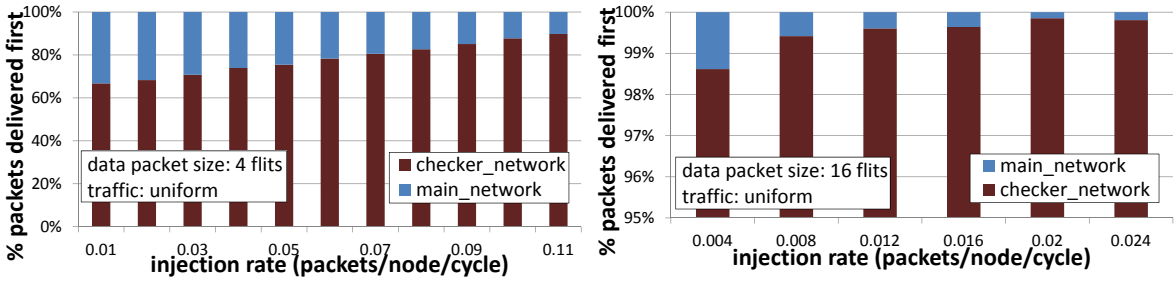


Figure 2.12 Checker network architecture for ForEVeR and SafeNoC. The checker router is packet switched with a single cycle latency. Packets are guaranteed to move forward once inside the checker network. To this end, packets that are waiting injection are given a lower priority as compared to the packets already inside the checker network. Finally, there is no contention at the ejection port.

be traversed in a single-cycle, ii) there is no contention within the checker network, and iii) in contrast to notifications, the main network transfers large data packets (*e.g.*, cache lines) that are typically divided up into long wormholes. As a result, the checker network design presented above consistently delivers notifications before their counterpart data packets arrive via the primary network. In exceptional situations, when notifications lag data packets, our scheme may produce a false negative detection result. In general, however, our scheme has a certain amount of tolerance to having a few notifications lagging behind and this rarely leads to false negatives. When it does, false negatives only increase the detection latency and do not affect the correctness of our scheme as we guarantee no loss of flits/packets, and the data would eventually be delivered in an uncorrupted state to the correct destinations upon error detection.

We ran experiments by injecting uniform traffic with varying packet sizes into the primary network and simultaneously injecting one notification into the checker network for each primary network packet. At the destination, the notifications were matched with corresponding data packets, and the following statistics were logged: i) which network delivered the packet or notification first (main or checker), and ii) the time difference between corresponding deliveries. Figure 2.13 shows the fraction of packets delivered first by each network for varying injection rates. If the main network uses short packets (Figure 2.13a), the majority of the notifications are delivered before the corresponding main network packets. However, there is a non-negligible fraction of packets (10-35%) for which the main network delivers first. Also, note that due to better congestion management in the ring, a larger fraction of notifications are delivered first when the main network is subjected to heavy traffic. Moreover, when the main network packets are larger (16 flits, Figure 2.13b),



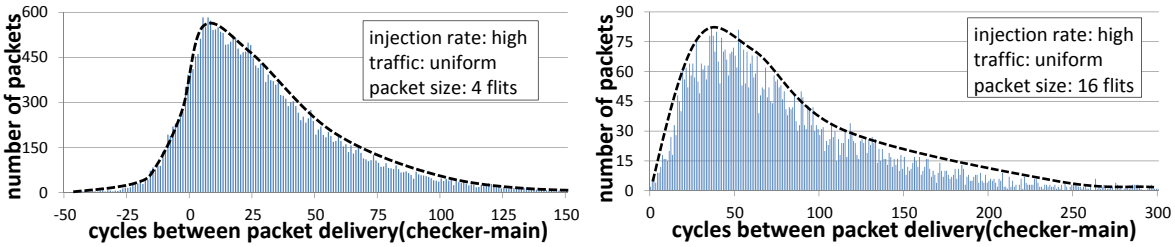
(a) Small packet size (4 flits). **(b) Large packet size (16 flits).**

Figure 2.13 Fraction of checker notifications delivered first. The checker network delivers almost all notifications before their counterpart primary network packets. The effect is more pronounced at higher injection rates and for larger main network packets.

the checker network is comparatively less loaded, and hence more than 99% of the notifications make it to their destination before the main network packets at any injection rate above 0.008 packets per node per cycle. In summary, the majority of the notifications are delivered ahead of their corresponding network packets under all conditions, with an additional marked improvement on notifications being delivered first when packets are large and the network is under heavy traffic load.

Figure 2.14 plots the distribution of notification-main packet deliveries for a range of delivery-time differences. A positive time difference indicates that the notifications arrived first. For small main network packets (4 flits, Figure 2.14a), most notifications are delivered 0-50 cycles ahead of the main network packets. Few main network packets (11% of total) are delivered before their corresponding notifications, and a negligible number of main network packets (<0.5%) are delivered more than 25 cycles before their notifications. Our detection scheme is tolerant to a few main network packets delivered early, when the notifications follow shortly after (<25 cycles later). ForEVeR employs a technique based on delayed counter update to eliminate false negatives. As can be noted from Figure 2.14b, the complete distribution is positive for networks with large packets, with only a negligible number of main network packets (<0.2%) delivered ahead of their notifications.

Recovery operation: As mentioned earlier, during recovery, each main network flit is transmitted in multiple segments through the checker network. The network interfaces house the dis-assembling/re-assembling logic to support recovery. The checker network recovery operation and logic is greatly simplified because only one router is transmitting packets to a single destination at a time. To this end, the checker network's channels include additional dedicated wires for head and tail indicators that are used during recovery operation. The checker packet with head indicator carries the destination address and reserves an exclusive path between the source and the destination. All intermediate valid packets



(a) Small packet size (4 flits). **(b) Large packet size (16 flits).**

Figure 2.14 Distribution of notification-main packet pairs over a range of delivery-time differences at heavy injection. The checker network delivers almost all notifications before their corresponding network packets.

traversing the ring network are ejected at the same destination until a packet is received with a tail indicator. Moreover, all transmissions on the checker network during recovery occur in the same (clockwise) direction to avoid wormhole overlap of two packets. In contrast to normal operation, the address field is not discarded on ejection, as it contains data in body/tail checker packets. In our evaluation system of 64 nodes, the checker network channel is 8 bits wide, with 6 bits for address and 1 bit each for head and tail indicators. Thus, each 64-bit primary network flit is partitioned into 12 checker networks packets (1 head, 11 body/tail) when transferred on the checker network. We expect design errors to manifest infrequently, and thus this serial transmission scheme, while slow, it should not significantly affect overall system's performance. The operation of the checker network during recovery is illustrated in Figure 2.15b.

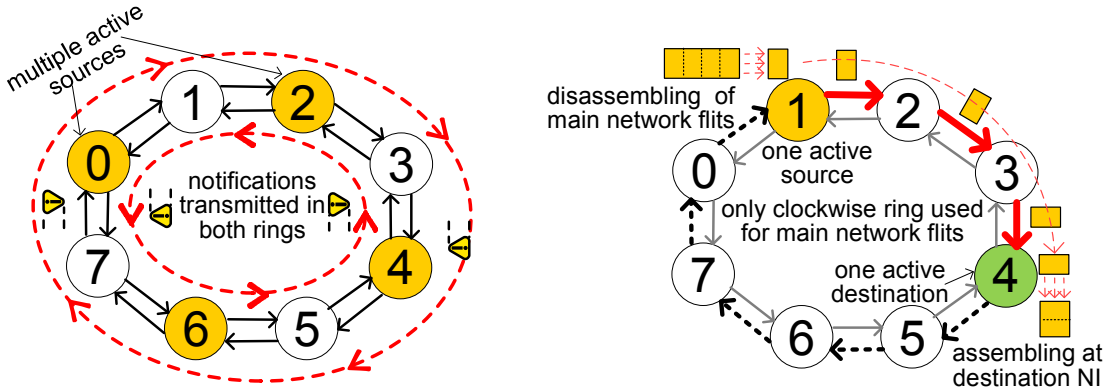


Figure 2.15 ForEVeR's checker network operation. (a) Normal operation: notifications are transmitted in both ring directions and multiple source-destination pairs are active simultaneously. (b) recovery operation: only the clockwise ring is used to transmit disassembled main network flits, that are then reassembled at the destination's NI. Additionally, only one source can be sending packets to one destination at any time.

2.5 Comparison of Runtime Protection Solutions

Table 2.6 outlines the qualitative comparison of SafeNoc and ForEVeR with other pure-runtime schemes. The comparison is with the following runtime solutions: i) ACK-Retransmission [85], which maintains a fresh copy of injected data until it receives a positive acknowledgement from the destination, and ii) Park-06 [94], which utilizes various micro-architectural modifications to overcome soft-errors both in router datapath and control-logic. We compare the four schemes on four aspects: protection against design errors, area overhead, and power overhead. Soft-error resilience is also considered for comparison because soft-error manifestations are similar to design error manifestations, and certain runtime-verification schemes can also be leveraged for soft-error protection. We discuss this aspect in detail in the next chapter. Only ForEVeR can guarantee complete correctness, while others fail to overcome either forward progress bugs or drop/duplicate packets or both. ACK-Retransmission, for instance, cannot recover from bugs like deadlock and livelock, and even for other bugs it uses the same untrusted network to retransmit data upon error detection, possibly incurring the error again and again. Therefore, the delivery of the retransmitted data in ACK-retransmitted cannot be guaranteed.

solution	**type of design errors protected**		**area overhead**	
	forward progress	**duplicate/ drop pkt**	**storage buffers**	**link wires**
ForEVeR	yes	yes	none	low
SafeNoC	yes	no	high	moderate
ACK-Ret	no	yes	very high	none
Park-06	no	no	very high	none
solution	**performance overhead**		**SEU protection**	
	normal operation	**recovery operation**	**data path**	**control path**
ForEVeR	no impact	short	yes	yes
SafeNoC	no impact	intractable	no	no
ACK-Ret	high impact	short	yes	yes
Park-06	low impact	low impact	yes	yes

Table 2.6 Comparison of runtime verification solutions.

ForEVeR does not need to buffer main network packets to support its detection and recovery operation. However, storage of notification packets at injection ports is required in our checker network architecture. The notifications only store the destination address (6-bits), and hence are considerably small as compared to main network packets. In our simulations, the worst-case storage required at checker network injection port was 11 no-

tifications when operated beyond saturation. As on-chip networks do no operate beyond saturation [87], this adds up to only 9 bytes of storage for the worst case, almost the same as one main network flit of 64 bits. Thus, the amount of notification storage required is insignificant.

In contrast, SafeNoC utilizes buffers at destination nodes to store checksums and packets waiting for their counterparts, while ACK-Retransmission maintains large source node buffers for all in-flight packets. Finally, Park-06 [94], requires additional buffering at each router FIFO to overcome deadlocks caused by soft errors, such that the total buffer size is large enough to accommodate the remaining flits of a packet allocated to the FIFO and still have one empty slot. Naturally, the storage requirement grows with larger packets. Moreover, this additional buffering is required at each VC for each input port. We ran stress tests on the NoC using uniform traffic to calculate the maximum buffering required over time for the SafeNoC and ACK-Retransmission solution. For SafeNoC, we observed the maximum number of outstanding checksums and data packets at any time, and for ACK-Retransmission we noted the maximum number of data packets in-flight. Buffer requirements for Park-06 can be calculated using the analytical equations provided in [94]. Figure 2.16 shows the results of our study and plots the worst case buffering required at each node. Note that ForEVeR requires no additional buffers, while all others require substantial buffer space for proper operation. The buffer requirements for ACK-Retransmission and Park-06 grow substantially for large packet sizes, while SafeNoC requires buffering for $\sim$40 flits/node independent of packet size. Note that provisioning for buffers to store 50-100 flits at each node can be prohibitively expensive for a constrained NoC environment. For instance, for data packets of 16 flits, the buffers in SafeNoC, ACK-Retransmission and Park-06 alone result in 12.5%, 37% and 26% area overhead over our baseline NoC, respectively. ForEVeR, however, does use some additional wires on a separate checker network, but since the transmitted notifications contain no data, we can design an area-efficient checker network, as we will discuss in Section 2.4. In contrast, SafeNoC uses a similar checker network to transfer larger (16-32 bit) checksum packets, while the ACK-Retransmission and Park-06 solutions do not use a separate checker network to overcome errors.

On the performance front, neither ForEVeR nor SafeNoC has any impact during normal error-free NoC operation, whereas ACK-Retransmission and Park-06 create additional traffic due to end-to-end and switch-to-switch acknowledgements, respectively. During recovery, all techniques other than SafeNoC can quickly recover from anomalous behavior. SafeNoC runs a packet reconstruction algorithm in software that, in the worst case, can be exponential in the number of flits to be recovered and reassembled.

Although protection against soft-errors is not the focus of this chapter, we briefly compare the solutions on this aspect due to the similarities between soft-error and design error manifestations. ACK-Retransmission and Park-06 can provide the best protection against soft-errors, while SafeNoC has no mechanism to overcome soft-errors. In contrast, as we will demonstrate in Chapter 3, ForEVeR with a few modifications can be leveraged to overcome soft-errors affecting both the datapath and control path of NoC routers. In summary, the low area and performance overhead, combined with the ability to protect against both design and soft-errors, makes ForEVeR the most complete runtime solution.

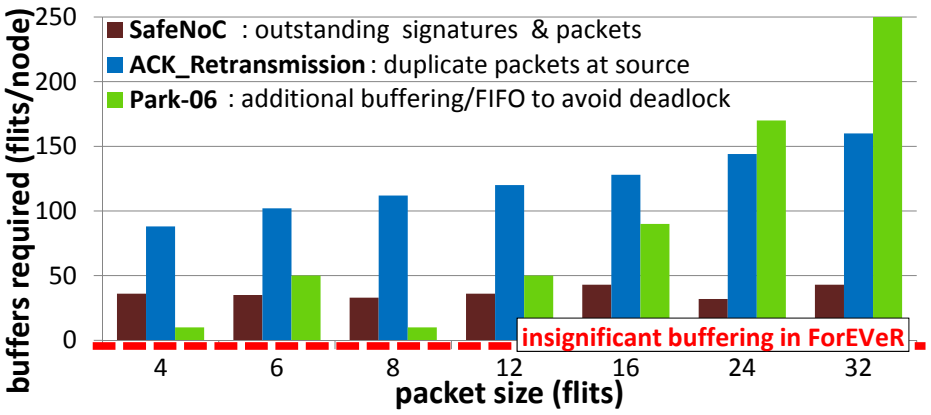


Figure 2.16 Storage requirement of runtime-verification schemes. ForEVeR requires minimal storage only for notifications waiting injection. SafeNoC stores signatures and packets until their counterpart arrives. ACK-Retransmission maintains a backup copy, while Park-06 requires escape buffers at each router FIFO to break deadlock cycles.

2.6 Related Work

NoC simulator distributions [28, 38], and RTL simulation and emulation platforms [54] are widely used for performance and functional verification. However, both simulation and emulation techniques are inherently incomplete, since they cannot check all possible execution scenarios. In contrast, formal techniques can provide the guarantees of complete correctness; however, they either cannot be automated, as in theorem proving, or they are limited by the state explosion problem, as in model checking. This has led designers to use formal verification exclusively for small portions of NoC designs [64] or to verify the abstracted model of the implementation [23, 16, 55]. Other works [16, 126] utilize theorem proving to guarantee NoC correctness and are also able to verify liveness properties like deadlock-freedom [125]. However, these abstracted NoC models, often do not model advance features like virtual channels, flow control techniques, etc. Moreover, properties

proven over the abstracted NoC model cannot guarantee correctness of the actual micro-architectural implementation. In contrast, SafeNoC and ForEVeR guarantee the correctness of all the executions of the NoC implementation, and they also enable designers to deploy aggressively-designed NoCs that are not exhaustively verified at design-time.

To address the limitations of design-time verification, runtime solutions have been recently proposed to ensure the correct transfer of data packets through the interconnect. Several of these works target deadlock, a prominent issue in adaptive routing. Traditionally, the deadlock problem in NoCs is overcome by deadlock avoidance [116, 29, 74], or through detection [79, 75] and recovery [9]. Other works tackle a wider, still incomplete, set of NoC errors through end-to-end detection and recovery techniques, and are surveyed in [85]. The most common among these is the acknowledgement-based retransmission (ACK-Retransmission) technique, where error detection codes are transmitted along with data packets, to check for data corruption at the receiver. An acknowledgement is sent back after each successful transfer. In case of failure, the sender times out and re-transmits the locally-stored packet copy. Apart from large storage buffers and performance degradation due to the additional acknowledgement packets, this approach is incapable of overcoming deadlock, livelock and starvation errors. Moreover, since it uses the same untrusted network for re-transmission, ultimately it cannot guarantee packet delivery. In contrast, both SafeNoC and ForEVeR safeguard against a wide range of functional bugs, including, i) bug manifestations that cause data corruption, such as, duplicated or misrouted packets, and ii) bug manifestations that stall the forward progress of the network, such as, deadlock, livelock, and starvation. In addition, both SafeNoC and ForEVeR are end-to-end solutions leveraging hardware units mostly decoupled from the primary NoC and requiring minimal changes to the primary NoC.

We are not aware of any runtime solutions that deal with design errors in NoCs. However, there has been few such works for processors, [10, 83, 128]. In general, these solutions add checker hardware to verify the operation of untrusted components. Our solutions are similar to such solutions, in the sense that they use a simple and functionally correct checker network to verify the operation of the complex primary network.

Our solutions rely on augmenting the original network with a small and lightweight one that operates concurrently. The idea of using multiple overlaid networks has been proposed for various purposes. [11] and [131] use multiple networks for performance enhancement. Others, such as TILE64 [13], use separate dedicated networks, each supporting a distinct functionality of the NoC. Mostly, these networks share the same topology and are comparable to each other in complexity. To the best of our knowledge, ours is the first attempt to overlay a network with another low-cost and error-free one to ensure the functional

correctness of the original interconnect.

Very few research works have proposed complementary use of formal and runtime techniques. Among them, [12] leveraged hardware checkers in model checking to avoid state explosion by validating abstractions at runtime. However, [12], unlike ForEVeR, cannot be directly applied to ensure NoC correctness. Another work proposed property checking at both design time and runtime [124]. However, as recovery from design errors in not supported, this is not a complete correctness solution.

Finally, ForEVeR's detection mechanism relies on the use of router-level runtime monitors, when formal methods fail to ensure router correctness. Runtime checkers has been proposed for various purposes, like soft-error induced anomaly detection in NoCs [98, 94] or post-silicon debug and in-field diagnosis [17]. In contrast, ForEVeR leverages specialized hardware monitors coupled with recovery support, specifically for NoC correctness.

2.7 Summary

In this chapter, we first presented SafeNoC, a runtime end-to-end error detection and recovery technique to guarantee the functional correctness of on-chip interconnects. SafeNoC augments the interconnect with a lightweight and simple checker network, and it 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, our novel recovery approach collects blocked packets and stray flits from the primary network and distributes them over the checker network to all processor cores, where our reconstruction algorithm reassembles them. SafeNoC can detect and recover from a broad range of functional design errors, while incurring a small performance impact, requiring only between 11K and 39M execution cycles to recover from an error.

This chapter then describes ForEVeR, a solution that improves the error coverage and reduces the area overhead compared to SafeNoC. ForEVeR complements the use of formal methods and runtime verification to ensure complete functional correctness in NoCs. Formal verification is used to verify simple router functionality, augmented with a network-level runtime detection and recovery scheme to provide NoC correctness guarantees. ForEVeR augments the NoC with a simple checker network used to communicate notifications of future packet deliveries to corresponding destinations. A runtime detection mechanism counts expected packets, triggering recovery upon unusual behavior of the counter values. Following error detection, all in-flight packets in the primary NoC are safely drained to their intended destinations via the checker network. ForEVeR's detection scheme is highly

accurate and can detect all types of design errors. The complete scheme incurs only 4.9% area cost for an 8x8 mesh NoC, requiring only up to 30K cycles to recover from errors.

Chapter 3
Addressing Reliability Threats

Reliability has emerged as a first-class design constraint for CMPs and SoCs with the shrinking of silicon feature size. Traditionally, memory elements have been the most vulnerable to transistor failures [109]. However, recent industrial-level studies have shown that transistor failures in logic components are far more prevalent than previously assumed [89, 86]. Protection of memory elements has been well researched; therefore, special attention is required for failures affecting logic components: both datapath and control logic. This chapter focuses on reliability schemes that protect the NoC against failures in both datapath and control logic.

NoCs occupy a significant portion of the on-chip real estate [56, 130] and observe high activity, making them susceptible to both transient and permanent faults. We tackle transient faults by observing that their manifestation is similar to instances of design errors: detection and re-execution is enough to overcome such errors. Therefore, in Section 3.2, we present an extension of ForEVeR that, with only a few modifications, can also overcome transient failures. In case of permanent transistor failures, however, a complete solution should entail detection and diagnosis of the fault site, followed by reconfiguration around the malfunctioning components. This chapter proposes solutions to tackle each aspect of protection against permanent faults. We first describe a low-overhead passive on-line detection and diagnosis scheme in Section 3.3, followed by a unified diagnosis and reconfiguration scheme for frugal bypass of NoC faults in Section 3.4. We conclude by presenting a novel NoC testing approach that ensures uninterrupted availability in the presence of faults by leveraging a quick and minimal-impact reconfiguration solution called BLINC (Section 3.5).

3.1 Detection, Diagnosis and Reconfiguration for Reliable NoC design

There are many different sources of errors in modern digital devices, and thus in NoCs: crosstalk, radiation-induced soft-errors, degradation-induced timing errors, *etc*. Often these effects result in temporary faulty behavior of the system, known as *soft* or *transient* errors. In addition, there are *permanent* faults resulting from manufacturing defects, infant mortality, or hardware aging. While manufacturing and burn-in tests may screen most permanent failures caused by the first two phenomena, permanent wear-out faults cannot be detected before shipment. Moreover, Borkar [15] predicts a 20% transistor failure rate in future CMPs due to variability effects, with an additional 10% over the lifetime of the chip due to aging. This waning reliability of silicon makes the problem even worse. As the sole medium for on-chip communication, NoCs are particularly susceptible to become a single point of failure.

Recent studies [34, 86] have confirmed the alarming rate of transient errors affecting both logic and memory components. As NoCs occupy a significant portion of the on-chip real estate [56, 130], their susceptibility to soft-errors is notable and, additionally, they suffer from single event upsets (SEUs) due to crosstalk and coupling noise in link wires. Soft-error tolerance solutions typically involve monitoring hardware to detect the occurrence of malfunctions, while relying on re-execution from an uncorrupted state for error recovery. For example, [26] employs N-modular redundancy (NMR) to detect and/or recover from soft-errors. The silicon overhead makes NMR impractical for commodity systems; hence soft-error recovery is often based on temporal redundancy, that is, re-execution. Still, naïvely executing duplicates for all computation and communication is not a feasible solution. Hence, designers leverage lightweight monitoring hardware to catch the majority of soft-errors, triggering a re-execution-based recovery procedure only upon an error detection.

In the last chapter, we proposed an NoC design that can overcome design errors. We further noticed that design errors that slip into the final silicon are rare corner-case execution scenarios. Even though a design error is a permanent disagreement between the hardware and its specification, it is unlikely that re-execution of the same application will lead to the manifestation of the same design error again. The reasons are as follows: first, corner-case bugs are triggered only during intricate interactions between components, and it is unlikely that execution scenarios are recreated precisely in large computer systems. Second, the execution trace depends on the operating environment, which is also hard to recreate. Finally, on detection of an error, we change the system to a configuration that

is simple and verified-to-be-correct. Therefore, design errors, similar to soft-errors, are typically one-off events. In this project, we exploit the similarities between soft-error and design error manifestations: i) both are relatively infrequent; therefore, a slow recovery is acceptable, and ii) re-execution from an uncorrupted state is a promising recovery solution for both. By leveraging these properties, we are able to design a lightweight solution, based on our design error protection solution, that can detect and overcome the majority of soft-errors and design errors in both control and datapath logic of NoCs.

Permanent faults pose a greater challenge, as simple detection and re-execution is likely to result in repeated errors. As a result, it is important to accurately *diagnose* the location of faulty components and to disable/bypass them for future executions. Reliability solutions, therefore, try to leverage the redundancy built into the system to enable the system to be operational, even after some of its components have been disabled. The process of disabling faulty components and replacing them with healthy ones is often termed *reconfiguration*. The guiding principle behind all reconfiguration solutions is to achieve *graceful degradation in performance with increasing number of faults*. Fortunately, NoCs provide many concurrent paths between communicating nodes, and thus they inherently benefit from fairly high redundancy. However, it is essential to diagnose the faults at a fine-granularity and to disable only small portions of the NoC's hardware with each fault manifestation. Therefore, a carefully designed combination of diagnosis and reconfiguration solutions is vital for NoC reliability: we propose a novel diagnosis solution in Section 3.3, while we present a frugal routing reconfiguration scheme in Section 3.4.

Even though routing reconfiguration can be leveraged for circumventing permanent failures, it requires lengthy suspension of network operation for reconfiguration changes to take effect. In addition, reconfiguration is triggered on a fault detection, by which time the network could already be in a corrupted state. In other words, routing reconfiguration fails to provide uninterrupted availability in presence of faults: a property that a digital system expects from its communication infrastructure. This property is particularly important for mission critical systems, where a down time of even a few milliseconds can be detrimental. To this end, we propose BLINC (Section 3.5), a novel reconfiguration solution that provides a quick and minimal-impact response to faults. We particularly leverage our BLINC algorithm to develop a transparent reliability solution for NoCs, based on aggressive online testing and failure prevention. In our setup, individual components undergo preventive testing by being taking offline on a rotating basis via our BLINC algorithm. The components are only brought back online if they do not show signs of failure under a strict testing environment.

3.1.1 A case for the need for better NoC reliability

In this section, we present a case study to drive the need for better NoC protection against permanent faults. Due to the continuous scaling of silicon and the saturation of single-thread performance, CMPs and SoCs with many on-chip processors are a growing market segment. Further, these large CMPs and SoCs are increasingly relying on NoC interconnects to provide scalable inter-core communication. With the waning reliability of silicon, permanent wearout faults are affecting silicon components at runtime with a greater probability. In the context of CMPs and SoCs, cores and the interconnect are the primary logic blocks; those that require careful safeguarding against permanent faults.

Researchers have recently proposed novel chip multiprocessor (CMP) architectures that can tolerate up to a few hundred ($\sim$500) processor-logic permanent faults [96, 53] in a 64-node CMP. These reliability schemes do not consider faults in the NoC infrastructure; however, faults in that sub-system may potentially lead to the isolation of one or multiple cores, thus greatly reducing the computational power of the system. The loss of many cores may occur even if the cores themselves are fault-free, simply because they become disconnected due to a faulty interconnect. Data from a few industrial CMP designs [130, 56] reports that roughly 6-15% of the chip area has been dedicated to the interconnect, and roughly 50% to the processor logic. Assuming such an area partition and a uniform distribution of faults over silicon area [65], 500 permanent faults affecting the 50% area occupied by processor logic translates to 60-150 permanent faults affecting the NoC logic that occupies 6-15% of the chip area. Therefore, the NoC should be able to gracefully tolerate up to 60-150 faults to match state-of-the-art processor-oriented reliability schemes. To evaluate the capabilities of current fault-tolerance schemes for NoCs, we modeled a 64-node NoC equipped with state-of-the-art fault-diagnosis [39] and route-reconfiguration schemes [4, 39]. We then randomly injected a varying number of permanent faults at gate outputs of the NoC netlist with uniform distribution over silicon area. The gate-level faults were then mapped to dysfunctional links using the model proposed in [39]. Finally, our model reported the number of isolated CMP nodes in each fault injection experiment.

Figure 3.1 plots the average number of processing nodes unreachable within such an NoC with increasing permanent faults. Each data-point on the graph corresponds to the average of 1,000 different spatially-random fault injection experiments. It is clear from the figure that existing state-of-the-art solutions deteriorate considerably beyond 30 transistor faults, dropping many nodes with just a few additional faults (see slope change). If 60 permanent faults were to occur in the silicon area occupied by the NoC, then almost 2/3rd of the nodes would become isolated, because the network would be partitioned into two or more distinct regions. Without the deployment of a more gracefully degrading NoC relia-

bility solution, the CMPs would potentially lose many healthy processing units with only a few permanent faults affecting the NoC.

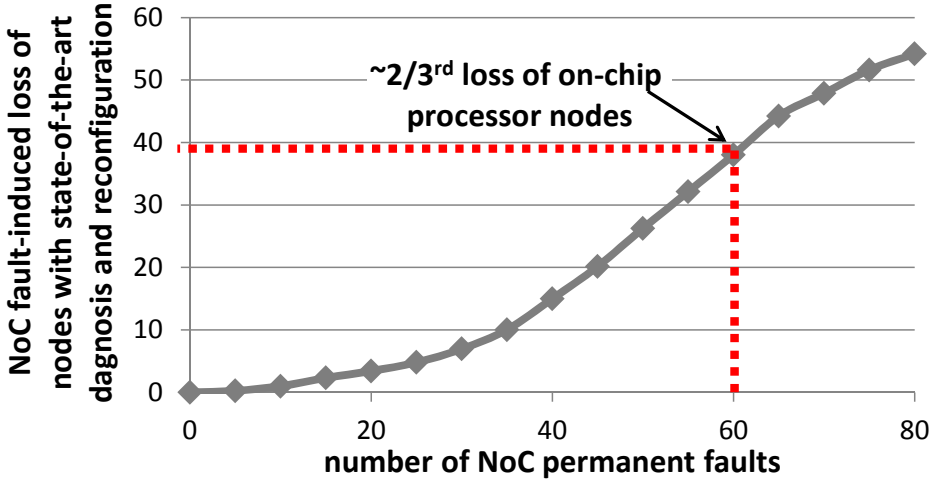


Figure 3.1 Loss of processing capability induced by faults in the NoC for a 64 node mesh network equipped with [4, 100] schemes.

To address this problem, we propose a novel solution in Section 3.4, called uDIREC, which drops over 3× fewer nodes than existing solutions, and thus it minimizes the network-induced loss of processing capability. Further, it is often the case with uDIREC that cores (and other on-chip functionalities) are only discarded when the number of faults is beyond the capacity of the fault-tolerant mechanisms protecting the cores themselves. uDIREC is beneficial for a variety of multi-core configurations (few or many cores) and varied fault rates (few to fault-ridden). For example, configurations with few cores degrade drastically and often become dysfunctional with the loss of one (or few) core(s). uDIREC extends the lifetime of such configurations by dropping the first (few) node(s) only after a significantly higher number of faults have manifested.

3.2 Soft-Error Detection and Recovery with ForEVeR++

The ForEVeR architecture described in the previous chapter, with few modifications, can also detect and recover from transient errors affecting any NoC component type: control, datapath or links. For the rest of this document, we refer to the soft-error resilient version of ForEVeR, as ForEVeR++. ForEVeR++ incurs minimal performance penalty up to a flit error rate of 0.01% in lightly loaded networks. In addition, ForEVeR++ hardware costs only 6% area over the baseline ForEVeR design.

We make the following observations to provide a cheap soft-error protection solution

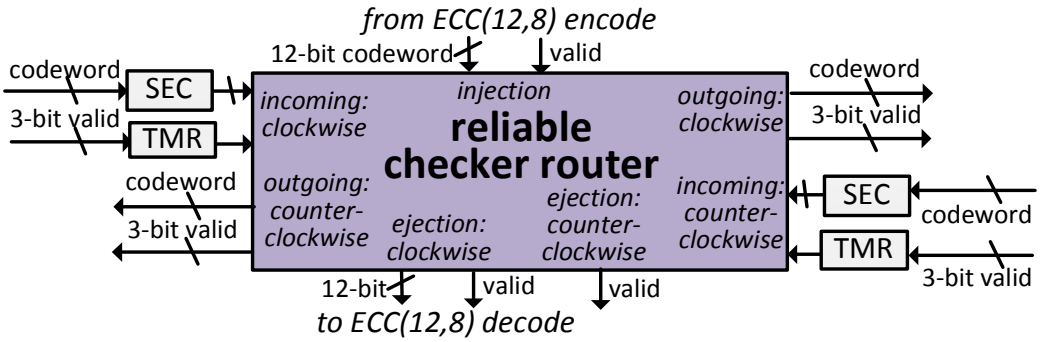


Figure 3.2 Reliable checker network design. ForEVeR++ employs switch-to-switch single error correction ECC (12-bit codeword, 8-bit data) to protect the checker network data, while it deploys triple modular redundancy (TMR) to protect the packet-valid bit.

with baseline ForEVeR: first, the router-level monitors designed to detect anomalous behavior due to design errors, can also prevent the network from entering an unrecoverable state upon a soft-error manifestation. Second, ForEVeR++ can provide low-cost soft-error resilience by utilizing the same recovery hardware as ForEVeR to reliably transmit the soft-error affected packets. Finally, to protect the recovery hardware (*e.g.*, checker network) itself against soft-errors, simple ECC and duplication based techniques can be leveraged without significant overhead. Overall, with these hardware modifications, ForEVeR++ can provide better soft-error coverage compared to the previous state-of-the-art techniques.

The most common soft-error resiliency techniques augment NoC packets with ECC information for error checking [85, 35], while relying on costly data retransmission for error recovery. Retransmission schemes cannot tackle all erroneous scenarios, especially the ones arising from control logic malfunction, *e.g.*, deadlock. A technique to recover from soft-errors in both the router's data-path and control logic was presented in [94]. It uses switch-to-switch ECC and retransmission for data-path errors, while the router control logic is protected with hardware checkers. However, this approach leads to a prohibitive area overhead, especially for networks with large data packets, as shown in Section 2.5. In contrast, ForEVeR++ does not require backup data storage, can overcome the deadlock scenarios, and it can recover from the errors in the control logic.

3.2.1 Methodology and Hardware Additions

Since ForEVeR++ intentionally builds on top of the hardware features of ForEVeR (Chapter 2.3), we borrow the terminology from the previous chapter without re-introduction. Reliability-oriented features in ForEVeR++ are focused on two aspects: i) enhancing the checker network reliability, and ii) ensuring that the integrity of data is always maintained

within the main network routers. The second aspect allows ForEVeR++ to eliminate any backup storage, and instead drain the erroneous packets via the reliable checker network. ForEVeR++ relies on the end-to-end locality-aware ECC [112] for fixing one-to-few bit data corruptions in the primary network packets. Link and datapath related errors, typically only affect single (or few) bit(s), which are corrected by the ECC. Based on the insight that severe data corruptions are only caused by errors in router control logic, ForEVeR++ can preserve data integrity by only protecting the router control logic.

Checker network reliability. For ForEVeR++'s reliable operation, the checker network should always deliver unaltered packets to their correct destinations. To this end, we augment the checker network with a switch-to-switch single-error-correct (SEC) ECC code. ForEVeR++ assumes that due to the small channel width of the checker network, in the worst case, only a single bit can be affected per notification. For the baseline 8-bit wide checker network, a 12-bit wide codeword is required for SEC capability. Additionally, the packet-valid bit is protected by triple-modular redundancy (TMR) to prevent from errors in the ECC output and to avoid transmission of an invalid packet. Thus, the sum total channel width of the reliable checker network is 15 bits (12 codeword, 3 packet-valid TMR). Figure 3.2 shows the architecture of the reliable checker router.

Error detection. Remember from Section 2.3.2, the router control components are augmented with runtime monitors to prevent an erroneous state due to design errors. These checkers can also be leveraged to detect anomalies due to soft-errors. Note that ForEVeR++ detects the anomalous behavior before the network goes into an unrecoverable state (for instance, by dropping a flit) and prevents any data loss/corruption by provisioning small emergency storage. Finally, in a rare scenario, soft-errors can also lead to forward progress errors: a situation easily identified by ForEVeR's network-level detection scheme.

Error recovery. Although the majority of the soft-error bugs can be overcome by restarting the router operation after a soft-error manifestation, we opt for network-wide recovery (Section 2.3.3) initiation at each error detection. This is because the hardware monitors cannot diagnose the source of error (design bug or soft-error), and thus our scheme sticks to a uniform measure in avoiding any unrecoverable state, *i.e.*, trigger a network-wide recovery. During recovery, all main network packets are delivered over the *reliable* checker network to their final destinations.

3.2.2 Experimental Evaluation

The performance of soft-error recovery schemes depends on the soft-error rate (SER). The greater the SER, the larger the performance penalty on network operation. For NoC specific experiments, average network latency with varying flit error rates is a well accepted metric to judge the quality of a recovery solution [85]. Flit error rate is defined as the probability of one or more errors occurring in a flit. Errors in a flit may be caused either by soft-error-induced malfunction of router control logic, or by data bit-flips in router datapath/links. Assuming equal logic, memory and link SER, we make a simplifying estimate that the probability of a flit error being caused by control logic malfunction is proportional to the silicon area percentage dedicated to control components. We conducted an area analysis of the components of the baseline router and observed that approximately 14% of the area is dedicated to control components, while the rest is attributed to datapath and links. Further, datapath corruptions are tackled by ECC with no performance overhead, while overcoming control logic errors involves freezing the router pipelines and draining packets over the checker network.

We ran simulations injecting errors in flits with varying probability. The simulator classified the injected flit errors as *control-induced* with 14% probability. The remainder were tagged as performance-neutral *datapath-induced* errors as they are correctable by the end-to-end ECC. On the other hand, the *control-induced* errors trigger a network-wide recovery, causing a performance hit. Figure 3.3 shows the average network latency with increasing flit error rate for varying loads of uniform random traffic. Naturally, the average network latency suffered with the increasing flit error rate. The effect on average network latency was tolerable (11% worse for 0.1 flits/cycle/node injection rate) up to the flit error rate of 0.01%, beyond which the latency degradation was quick. We also observed that the latency degradation was more drastic for higher injection rates. This is because more main network flits are transferred via the reliable checker network upon recovery initiation. Finally, unlike retransmission schemes based on sending acknowledgement messages during normal operation [85, 5], ForEVeR++ does not introduce any extra traffic into the main network, and hence it does not incur a performance hit in the absence of errors. We conclude that ForEVeR++'s soft-error protection is suitable for networks operating at high load and expecting a flit error rate of less than 0.001% (worst-case latency degradation of 7%), while ForEVeR++ can sustain a flit error rate of up to 0.01% for networks operating at low load.

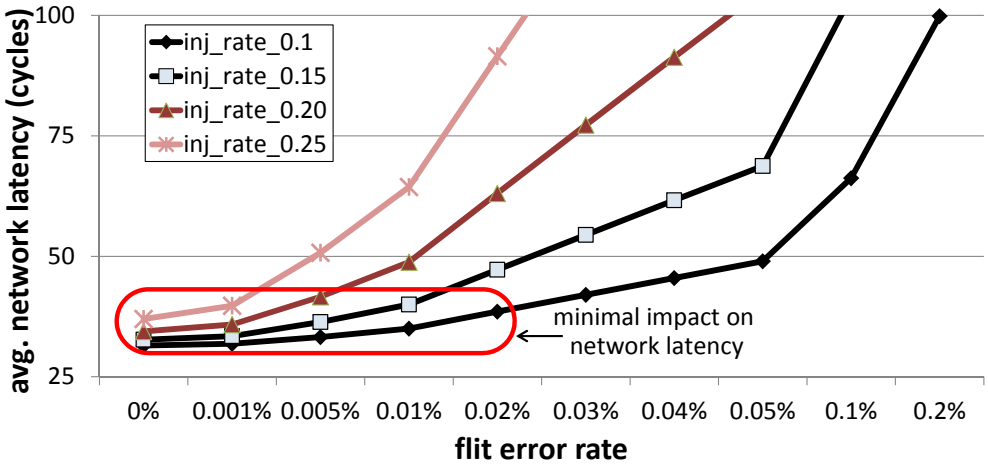


Figure 3.3 Average network latency vs. flit error rate for varying injection rates. A minimal impact on latency is observed up to the flit error rate of 0.01%. As more flits are transferred via the reliable checker network upon recovery initiation for high injection runs, the latency degradation increases with the injection rate.

3.3 Permanent Fault Detection and Diagnosis

A permanent fault in any datapath component along the path of a transmitted data packet can manifest as a corrupted packet payload. Datapath components include routers' external and internal wires, input FIFOs, crossbars and output buffers. The control components, on the other hand, manage the flow of packets and flits from input channels to output channels via the routers' datapath. If a fault strikes the control hardware of a router, it can result in dropped, spurious, duplicated, or misrouted flits or complete packets [7]. Hence, it is vital to protect both datapath and control logic against permanent faults. To this end, comprehensive diagnosis of NoC permanent faults, both data and control, is required.

We present a low-cost and fine-resolution mechanism to detect and diagnose permanent faults in both the datapath and control logic of the NoC. The proposed solution is passive, *i.e.*, it does not inject any test traffic into the network. It comprises of an end-to-end error symptom collection mechanism for locating datapath faults, and a distributed counting and timeout-based technique to locate defective control components. Figure 3.4 shows a generic NoC architecture, briefly describing our detection and diagnosis scheme. Faults in router datapath (highlighted grey) are diagnosed using a software-based scoreboard (highlighted green-checked), while simple counters and timers (highlighted green) are deployed to detect failures in router control components (highlighted blue). Our solution assumes that the network is already equipped with ECC-based forward error correction [35] to detect and correct data corruptions due to soft-errors. To make up for data possibly lost between error manifestation and detection, we assume that the system is equipped with

an orthogonal data recovery scheme, such as the recovery scheme of ForEVeR (Section 3.2). Experimental results show that the proposed method can precisely pinpoint a defective component, while introducing no performance overhead in the absence of faults and only leading to 2.7% area overhead.

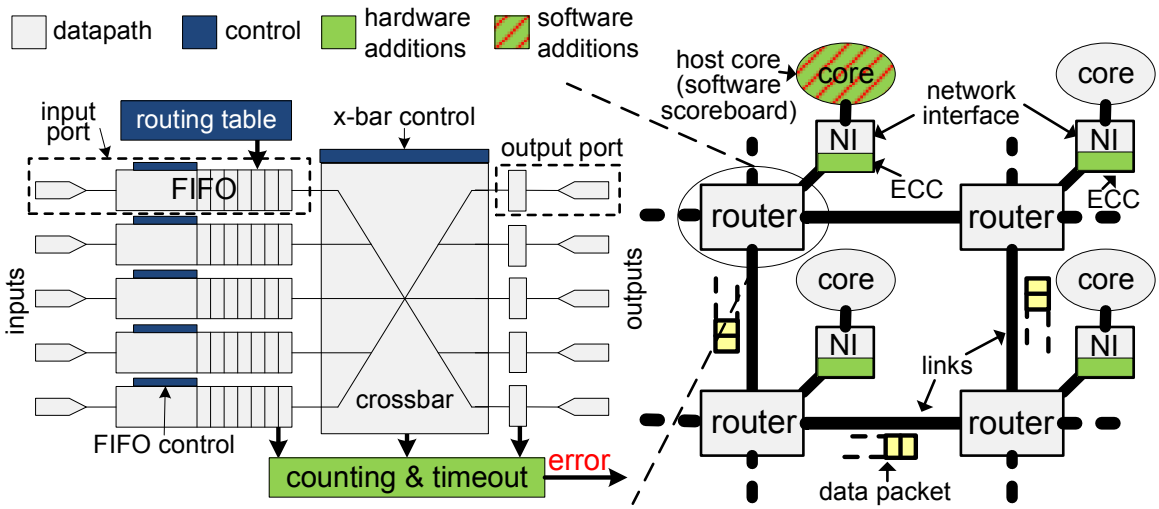


Figure 3.4 High-level overview of our detection and diagnosis scheme. Router logic is classified into datapath (grey) and control (blue) components, with a software scoreboard (green-checked) used to diagnose datapath faults and counters (green) used to detect control failures.

3.3.1 Diagnosis Resolution

The work in [112] proposed an online diagnosis mechanism that analyzes payload data errors to locate faulty links, while ECC is used to correct those data errors. However, the scheme in [112] is limited to the faults in links between routers and does not directly consider faults inside the router logic. Moreover, this method does not address control errors, such as packet drop/duplication.

Our detection and diagnosis scheme extends the proposal in [112] by localizing fault manifestations in any router component (links, datapath or control logic) to the finest resolution possible when observing end-to-end traffic. The extension is based on two innovations: i) the deployment of simple counters and timers to diagnose errors affecting routers' control logic at fine-granularity, and ii) the perception of routers' datapath as an extension of the links connected to it. Elaborating on the latter, since our diagnosis mechanism can only observe errors in an end-to-end fashion, it can differentiate between faults in two components only if there are certain packets that go through one component and not through the other, and vice-versa. Similarly, when considering route-reconfiguration to

circumvent permanent faults, the finest granularity at which components can be disabled is exactly the same. Therefore, faults in the following set of datapath components are not differentiable from an end-to-end diagnosis and route-reconfiguration perspective:

- output port buffer at the upstream router,
- link between routers,
- input port buffer at the downstream router, and
- crossbar contacts with the output (input) port at the upstream (downstream) router.

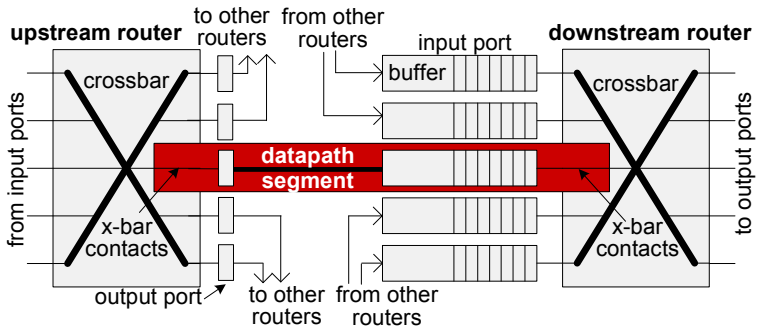


Figure 3.5 A datapath segment includes the crossbar contacts to the output port and the output port in the upstream router, the link, the input port buffer and the crossbar contacts from it in the downstream router.

Throughout the rest of this document, we call the combined set of these datapath components that are non-differentiable from network ends, as a *datapath segment*. Figure 3.5 illustrates the concept and constituent components of a single *datapath segment*. We further observed that a fault manifestation in any of these constituent components can be modeled as the failure of a single unidirectional link at an architectural level. We empirically studied the micro-architecture of wormhole routers to establish that a majority of faults can be masked by re-routing around a single datapath segment. To this end, we synthesized a 5-port baseline mesh router, and conceptually divided all router components into two pools: i) components that only affect one datapath segment's functionality, and ii) components that affect the entire router's functionality. Examples of the former category are crossbars, output ports, input ports and links, while the latter category includes arbiters, allocators and routing table. Components that only affect one datapath-segment accounted for 96% of the router's silicon area: 96% of the faults will affect only one datapath segment, assuming an area-uniform fault distribution. We ascertained our observations by randomly injecting stuck-at faults at gate outputs of every such component, and testing the "unaffected" datapath segments for proper functionality with random test vectors.

After diagnosing faults at a fine resolution, our scheme subsequently reconfigures by leveraging the location of the *malfunctioning* datapath segments or unidirectional links. Our reconfiguration algorithm, described in Section 3.4, fully utilizes this fine-grained diagnosis information to enable graceful degradation with increasing transistor faults. We also present the design of a unified diagnosis and reconfiguration NoC architecture to enable frugal bypass of NoC faults in Section 3.4.

3.3.2 Datapath Faults

We introduce a novel diagnosis method based on a probabilistic scoreboard, similar to the one proposed in [112]. The scoreboard, implemented in software, maintains the probability of routers' datapath components being faulty. Before determining the probability of a datapath component being faulty, it is essential to detect the errors in data transmission. To this end, we add flit-level end-to-end ECC to each packet. Since data errors must be caused by one of the datapath components on the routing path of the erroneous packet, analyzing the erroneous packet's route provides insights into the failure location.

When a data error is detected at the destination, an error symptom packet is sent to a designated "supervisor node" responsible for scoreboarding and diagnosis. This packet contains the source and the destination address of the erroneous packet and the bit position where the error occurred. Based on this information, we use a probabilistic method to calculate the fault probability for each component along the possible paths from the source to the destination. Our scheme accumulates these probabilities in the scoreboard on each fault occurrence. After observing sufficient erroneous traffic in the system, the diagnosis scheme declares the component corresponding to the scoreboard entry with the highest accumulated fault probability as defective.

Calculation of fault probabilities. The scoreboard is a table of floating point fault probability entries for each datapath segment of the network. Every time an erroneous packet symptom is received by the supervisor node, we calculate and accumulate the fault probability for each datapath component that could be responsible for the failure. Specifically, all components lying on the path of the erroneous packet are equally likely candidates for being faulty. Our diagnosis scheme is based on the intuition that in the case of a permanent fault, the scoreboard entry corresponding to the faulty datapath segment will quickly accumulate a value higher than the healthy datapath segments' entries. Hence, our scheme monitors the scoreboard for the highest fault probability value, till the number of erroneous packets exceeds a preset threshold. After monitoring sufficient number of erroneous trans-

missions to gain diagnostic confidence, our scheme declares the component corresponding to the highest fault probability as faulty.

For a deterministic routing algorithm, the routing path is fixed for a given source-destination pair, and hence error probabilities are updated in equal proportion for all *datapath segments* on the path of an erroneous transmission. However, the corresponding calculation is tricky for minimally-adaptive routing algorithms. To this end, we assign separate *usage probabilities* to all NoC datapath segments, with respect to each source-destination pair. These usage probabilities correspond to the fraction of times a particular datapath segment will be traversed, when a packet is transmitted between a particular source-destination pair. Therefore, usage probabilities depend on the path selection algorithm employed at each router and can be easily calculated a-priori for static path selection algorithms. On each erroneous transmission between a particular source-destination pair, we update the scoreboard entry for each datapath segment by that segment's usage probabilities corresponding to that source-destination pair.

3.3.3 Control Faults

A fault in a router's control logic may lead to many different failure scenarios: corruption of data, complete flits/packets being dropped, or spurious ones generated. In addition, such faults can also lead to packets being misrouted, potentially resulting in deadlock or livelock. Other errors that inhibit forward progress, such as starvation, are also possible due to faults in the arbitration logic. Hence, our scheme tries to detect various symptoms of control logic faults and accurately diagnose the faulty datapath segment(s) using the following techniques:

- Corrupted data: based on the diagnosis mechanism described in Section 3.3.2.
- Dropped or spurious flits: by counting the number of flits per packet received by a router's input port.
- Dropped or spurious packets: by keeping a count of packets inside a router.
- Misrouted packet: by detecting packet reception at wrong destination.
- Starvation: using timeouts.

We monitor these symptoms at a per-router or per-datapath-segment basis, and all diagnosis information is relayed back to the "supervisor node". In addition to maintaining a scoreboard for diagnosing datapath faults, the supervisor node also keeps a "status table" to reflect the state (working or faulty) of each datapath segment in the NoC. We update

this table upon receiving the diagnosis information from various nodes. This forms a convenient representation for interfacing with the route-reconfiguration algorithms, as will be discussed later in Section 3.4. As detection of *misrouting* or *starvation* errors is straightforward, we only discuss detection of dropped/spurious flits/packets in detail.

Dropped or Spurious Flits

In an NoC, packets are typically broken down into fragments called "flits". The number of flits per packet can be extracted from the header flit or implied in case of a fixed packet size. Our scheme uses this information to detect dropped or spurious/duplicated flits by counting the number of flits sent from each input port (towards the output port), starting with the header flit and stopping at the tail flit. If this count differs from its expected value, then either a flit was dropped or created within this input port or the connecting output port of the upstream router, *i.e.*, within the corresponding datapath segment. To enable this scheme, only one counter per input port, bounded by the maximum size of any packet serviced by the network, is required.

Dropped or Spurious Packets

In order to detect a dropped or duplicated/spurious packet, our scheme maintains a *packet counter* at each router, incrementing the counter at a packet's tail flit being received and decrementing the counter at a packet's tail flit being sent. This counter value is analyzed to detect dropped and spurious packets. The diagnosis granularity using this scheme is at the router-level. However, we also propose an alternate scheme of counting packets per datapath segment. Although, it improves the diagnosis granularity to the datapath-segment-level, this alternative requires more hardware additions.

Dropped packet. A packet dropped inside a router/datapath-segment means that there are more packets coming into the router/datapath-segment than going out of it. To this end, we maintain a *packet counter* per router/datapath-segment that is incremented on each packet received and decremented on each packet sent. A dropped packet will result in the counter never reading a zero value. Therefore, a dropped packet can be detected if the packet counter does not read a zero value during an entire execution window or "check epoch". This scheme avoids false negatives, but may cause false positives when heavy traffic in the routers leads to continuously occupied routers. We experimentally show that choosing a suitable *check epoch* size can reduce the false positive rate to an acceptable level. This scheme is similar to the end-to-end notification counting scheme of ForEVeR that was

described in Section 2.3.

Spurious packet. A duplicated/spurious packet error is flagged when the *packet counter* reaches a negative value. A negative packet counter value means that more packets have been sent out of the router/datapath-segment than have entered the router. Note that this scheme is susceptible to false negatives: even if one or more packets are duplicated within a router/datapath-segment, the corresponding counter can possibly remain at a non-negative value if the faulty router/datapath-segment is continuously occupied by other packets for an extended period of time. However, experiments show that false negatives are extremely rare for realistic traffic loads, and detection typically occurs within a few cycles.

3.3.4 Experimental Evaluations

We assessed our scheme's diagnosis accuracy for datapath faults using Monte Carlo simulations with a simple C++ simulator. However, we used a cycle-accurate NoC simulator [28] for evaluating our scheme's accuracy in diagnosing dropped or spurious packets. Our baseline NoC is an 8x8 XY-routed mesh network with 4-flit packets; routers are a simple 3-stage pipeline with no virtual channels and 5-entry deep buffers per input port. Our scheme was analyzed with two different types of workloads: random traffic and application traces from the PARSEC benchmark suite [14].

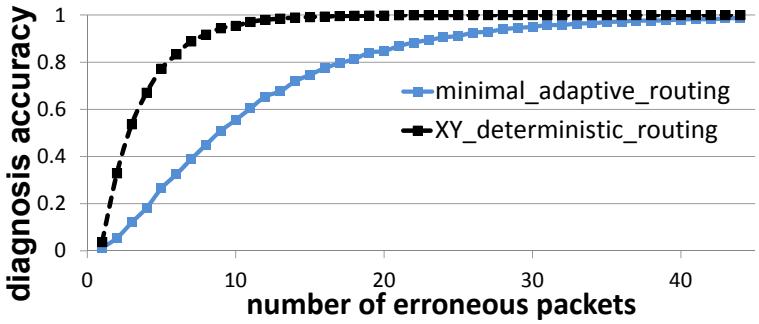


Figure 3.6 Diagnosis accuracy with increasing number of erroneous packets observed by the supervisor node. Higher diagnosis accuracy is achieved for deterministic routing as compared to minimal adaptive routing for the same number of erroneous packets observed.

Datapath faults

Intuitively, the location of the faulty component is diagnosable with high confidence if a sufficiently large number of erroneous packets are observed. Figure 3.6 shows the diagnosis accuracy of our method for a selected faulty datapath segment, while varying the

number of erroneous packets observed. The figure shows results for both XY and minimal-adaptive routing. Over 98% diagnosis accuracy is achievable after fewer than 15 erroneous packet observations with XY routing and around 40 erroneous packet observations with minimal-adaptive routing. The difference can be attributed to the exact knowledge of routing paths with XY routing, while relying on a probabilistic estimation of routing paths for minimal-adaptive routing.

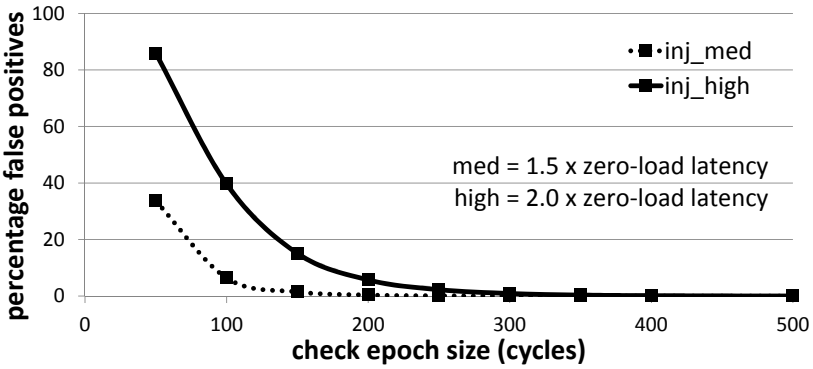


Figure 3.7 False positive rate for the dropped-packet detection scheme with increasing *check epoch* size at medium and high injection. The false positive rate drops for longer *check epochs* and lighter network loads.

Dropped Packet

The scheme proposed for the detection of dropped packets can exhibit false positives. The false positive rate depends on the duration of the *check epoch* and traffic conditions. Note that false positives are triggered when the packet counter is non-zero for an entire *check epoch*; a heavily loaded network will see more false positives as packets accumulate at router buffers. Intuitively, a longer *check epoch* reduces the false positive rate by allowing more time for packets to clear routers' buffers. Figure 3.7 shows the decrease in false positive rate with increasing *check epoch* size for datapath-segment-level diagnosis. As seen from the figure, a heavily loaded network exhibits a higher false positive rate than a moderately loaded network, and hence requires a larger *check epoch* to limit the false positives. Finally, false positives drop to a negligible value beyond a certain *check epoch* size, referred to as $epoch_{min}$).

Figure 3.8 plots the $epoch_{min}$ value necessary to eliminate all false positives and the average network latency, under uniform traffic at various loads. $Epoch_{min}$ exhibits a slow increase with rising injection rate up to network saturation and a steep rise afterwards. From the plot, the worst case $epoch_{min}$ of 1K cycles is sufficient to eliminate all false positives

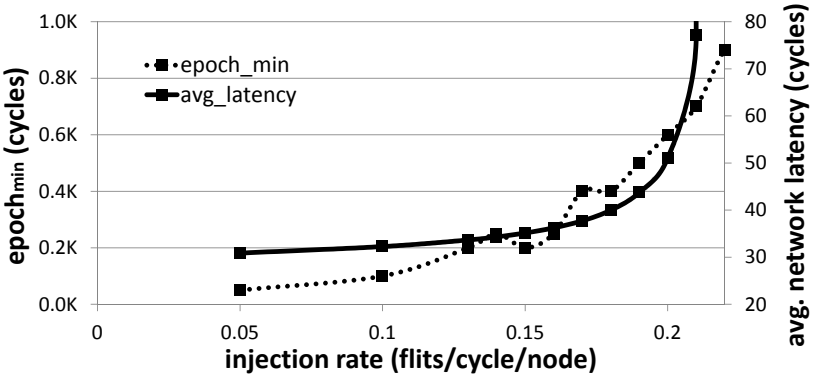


Figure 3.8 The Figure shows the variation of $epoch_{min}$ and latency with increasing network load for the dropped-packet detection scheme. An $epoch_{min}$ size of 400 cycles is sufficient to eliminate all false positives for networks at onset of saturation.

when the network is in deep saturation, operating at an average latency of well over 3 times the zero-load latency. A similar result was observed for 9 different PARSEC benchmark traces (1 million instructions each), where a *check epoch* size of 300 cycles was sufficient to eliminate all false positives for every benchmark. Our simulations indicate that $epoch_{min}$ rises to high values only when the network is operated at loads well past saturation. Such a scenario is unlikely: NoC workloads are characterized by the self-throttling nature, which prevents them from operating past saturation [87].

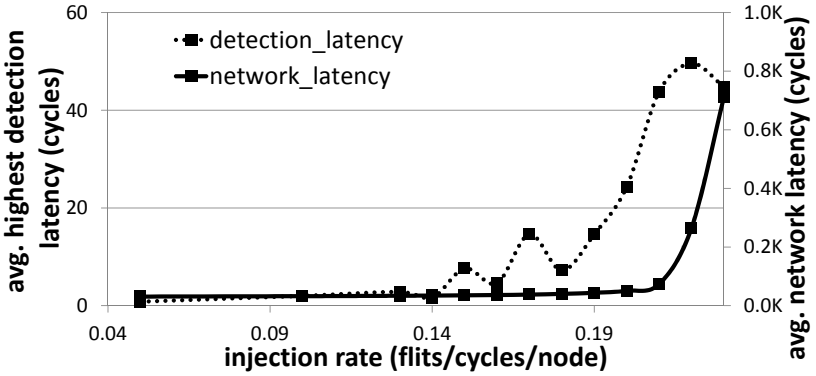


Figure 3.9 Spurious-packet detection scheme under uniform random traffic. The Figure plots the average of highest detection latency observed during multiple runs. The figure also shows average network latency during those runs with increasing network load. The detection latency is within 20 cycles before the onset of network saturation.

Spurious Packet

Our detection scheme for spurious packets can exhibit false negatives, while false positives are not possible. Hence, a different methodology was required to evaluate this scheme. After operating the network fault-free for a preset length of time,withpacket is intentionally created with a certain probability and written to the buffer of a randomly chosen datapath-segment of a random router. To gain statistical confidence, we injected 10,000 such faults, one after the other, and repeated each simulation 10 times with different seeds. We define detection latency as the time required to detect the error after the packet was created. If detection latency for a specific case is more than 10,000 cycles, we flag it as a false negative. Figure 3.9 plots highest detection latency across all 10,000 faults, averaged over all random seeds, with increasing network load. The figure also plots the average network latency to indicate the network operational condition. Again, average detection latency increases slowly up to a certain network load, after which there is a steep increase. We observe that worst case average detection latency is within 1K cycles even for networks operating at deep saturation. At this load, the average network latency is greater than 8x of the zero load network latency. For the network loads shown in the graph, no false negatives were observed. Similarly, for the 9 PARSEC benchmark traces, a maximum detection latency (across all faults and different executions) of 178 cycles was observed for the datapath-segment-level diagnosis.

3.4 Frugal Reconfiguration with uDIREC

Once the fault locations are diagnosed at a fine granularity, it is essential to effectively utilize the information towards a gracefully degrading reliability solution. To this end, we propose a novel solution, called uDIREC, which, upon fault manifestations, drops over 3$\times$ fewer nodes than existing solutions, and thus minimizes the network-induced loss of processing capability. uDIREC builds on the findings presented in Section 3.3.1, which states that majority of the router logic and wire faults can be localized to a single *datapath segment*. Further, as discussed earlier, the functionality of a datapath segment can be entirely masked by disabling and re-routing around a single unidirectional router-link. Our fine-resolution diagnosis scheme (Section 3.3) provides the opportunity for graceful degradation by allowing us to disable only a single unidirectional link on each fault. However, existing route-reconfiguration solutions [4, 100] fail to exploit these healthy unidirectional links, as they can operate with bidirectional links only. Therefore, these reconfiguration schemes unnecessarily consider a fault in one direction to be fatal for the entire bidirectional link,

and *cannot* benefit from the fine-grained diagnosis information. uDIREC's contributions can be summarized as:

- **A fine-grain fault model for NoCs** that enables the frugal bypass of faulty unidirectional links, without disabling other healthy link(s).
- **A novel routing algorithm** to maximally utilize unidirectional links in fault-ridden irregular networks that result from the application of our fine-grain fault model to faulty NoCs. The routing algorithm is guaranteed to discover only deadlock-free routes without requiring additional VCs.
- **Software-based reconfiguration** handles in-field permanent faults in NoCs. It places no restriction on topology, router architecture or the number and location of faults. Internally, it utilizes uDIREC's novel routing algorithm to discover a new set of deadlock-free routes for the surviving topology.
- **uDIREC** (for **u**nified **DI**agnosis and **REC**onfiguration). The integration of the fine-grained diagnosis and reconfiguration scheme enables a low-cost and frugally-degrading reliability solution. Experiments on a 64-node NoC with 10-60 interconnect-related faults show that uDIREC drops 60-75% fewer nodes and provides 14-40% higher throughput over other state-of-the-art fault-tolerance solutions.

3.4.1 Fault Model

We call the traditional fault model, the coarse-grain fault model *Coarse_FM*: it constrains the residual network to have bidirectional links only. Therefore, reconfiguration solutions based on the *Coarse_FM* [39, 4, 100] are often inspired by irregular routing algorithms designed for networks with bidirectional links only [25, 110]. Up*/down* routing [110] is a classic example of such a routing algorithm. Up*/down* works by assigning directions to all links in the network: *up* or *down*. Links towards the root node (connecting to a node closer to the root) are tagged as *up* links, while links away from the root are tagged as *down* links. Links between nodes equidistant to the root are tagged arbitrarily. All cyclic dependencies are broken by disallowing routes traversing a *down* link followed by an *up* link. With this solution, we propose a novel and refined fault model where all link- or router-level faults are mapped to datapath segments, *i.e.*, to *unidirectional links*. We call this fine-grain fault model *Fine_FM*. uDIREC leverages the additional links reported as fault-free by the *Fine_FM* to improve the reliability and performance of the faulty networks.

- **Improved Reliability:** A solution based on the *Fine_FM* can provide better connectivity than the *Coarse_FM*, as the number of working links shrinks faster for the latter.

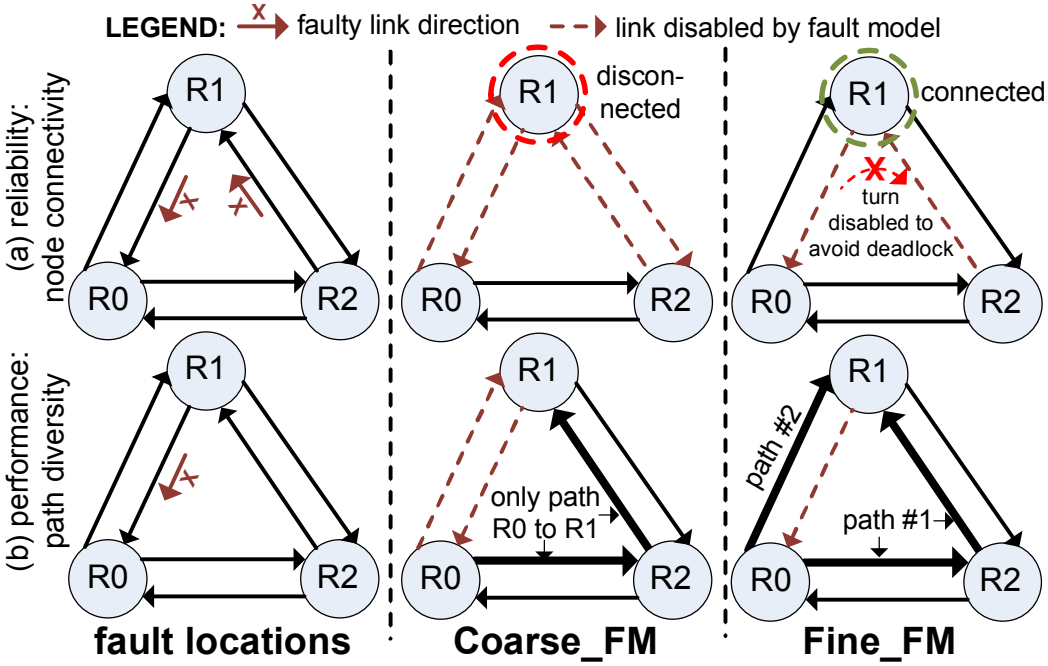


Figure 3.10 *Fine_FM* **provides additional working unidirectional links**. They can be utilized to (a) connect more nodes (reliability), and (b) provide path diversity (performance).

In Figure 3.10a, *R1* is isolated when using the *Coarse_FM*. However, the *Fine_FM* allows deadlock-free routes between any pair of nodes by simply disabling the turn *R0*→*R1*→*R2*.

- **Improved Performance:** The *Fine_FM* can improve path diversity over the *Coarse_FM*, as working unidirectional links can be used to transmit packets. In Figure 3.10b, the unidirectional link *R1*→*R0* is faulty. With the *Coarse_FM*, only one path remains from *R0* to *R1*, via links *R0*→*R2* and *R2*→*R1*. However, with the *Fine_FM*, an additional route can be utilized from *R0* to *R1*, via the unidirectional link *R0*→*R1*.

For the experiments, we have modeled the diagnosis scheme described in Section 3.3, which can localize most fault manifestations to the resolution of a single datapath segment. The same diagnosis information is provided to all evaluated route-reconfiguration schemes (uDIREC and prior works). However, prior works such as Ariadne [4] and Immunet [100], restricted by the *Coarse_FM*, cannot utilize this fine-grained fault localization information due to the limitations of their underlying routing algorithms.

3.4.2 Routing with Unidirectional Links

The constraint that all network links are bidirectional enables a desirable property: if a path between two nodes exists, irregular routing algorithms based on spanning tree construction can enable at least one deadlock-free route between them. In contrast, finding deadlock-free routes between any pair of nodes in a connected network is not always possible if the network has unidirectional links. Since our routing algorithm must enable only deadlock-free routes, it is possible to have to sacrifice a few connected nodes to achieve this goal.

Connectivity and Deadlock Freedom

We consider a network to be connected only if transmission is possible between any pair of nodes in either direction (two-way connectivity). Additionally, we assume that VCs are only used for separation of traffic into classes and/or avoiding deadlock in client protocols, but we do not require them to overcome routing deadlocks [51]. Based on their connectivity characteristics, we can divide all networks with unidirectional links into three categories:

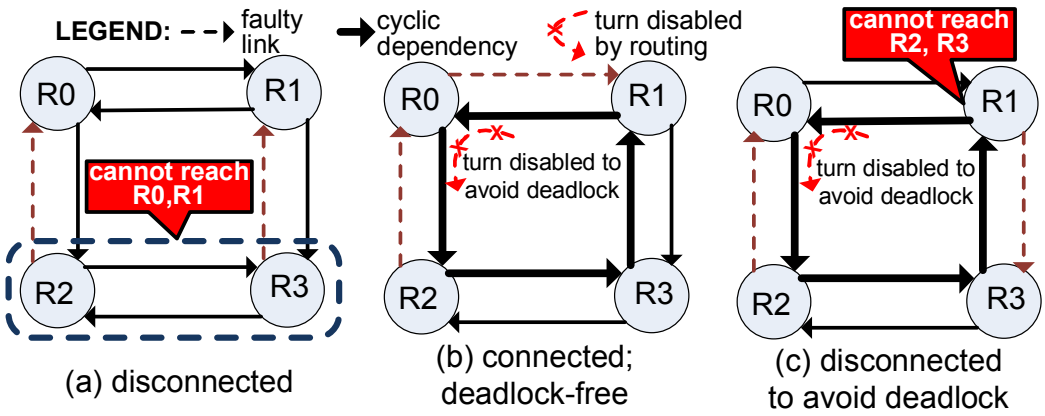


Figure 3.11 (a) Network disconnected. (b) Deadlock-free connectivity possible by disabling turn *R1→R0→R2*. (c) Network connected but deadlock freedom not possible without sacrificing connectivity.

(i) **Disconnected**: The network in Figure 3.11a is disconnected as traffic cannot traverse from *R2 & R3* to *R0 & R1*.

(ii) **Connected and deadlock-free**: In Figure 3.11b, a connected network has a cyclic dependency in the anticlockwise direction. Such a cycle can cause deadlock, as packets residing in routers' buffers can be indefinitely waiting for packets in front of them to free up buffer space. Formally, as a sufficient condition for deadlock freedom, all such dependency cycles should be eliminated [29]. Thus, we use an approach based on disabling certain

connections between links, called 'turns' in [50], so that the packets cannot form a cyclic dependency. For example, in Figure 3.11b, the *R1→R0→R2* turn is disabled to break the anticlockwise cycle. In other words, the routing algorithm prohibits messages to go from *R1* to *R2 & R3* via *R0* to avoid packets from deadlocking. Connectivity is maintained even after disabling this turn.
(iii) **Disconnected to avoid deadlock**: The network in Figure 3.11c is connected as messages can be exchanged between any pair of nodes. However, if the turn *R1→R0→R2* is disabled to break the anticlockwise cycle, connectivity is lost. Specifically, *R1* cannot send messages to *R2 & R3*, as the only path connecting the concerned nodes goes through this disabled turn. Even when disabling any other turn to break the cyclic dependency, connectivity is jeopardized.

uDIREC's routing algorithm is designed to maximally utilize resources in faulty networks that result from the application of the *Fine_FM*. uDIREC deploys this routing algorithm on each fault manifestation, to quickly discover reliable routes between the still-connected nodes. The routing algorithm works by constructing two separate spanning trees with unidirectional links: one for connections moving traffic away from the root node (down-tree), and the other for connections moving traffic towards the root node (up-tree). Each node is then assigned a unique identifier corresponding to each tree: identifiers increase numerically with increasing distance from (to) the root in the down-tree (up-tree), while equidistant nodes are ordered arbitrarily. This leads to a unique ordering of nodes (lower order = closer to root) in each tree. Thereafter, the *up* link is defined as the unidirectional link towards the node with the lower identifier in the up-tree and the *down* link is defined as the unidirectional link towards the node with the lower identifier in the down-tree.

Lockstep Construction

The two spanning trees, however, cannot be constructed independently of each other. Because of the use of unidirectional links, such an approach could lead to a mismatch in the node ordering between the trees, and links could consequently receive inconsistent tags: *up* or *down*. An example of such situation is shown in Figure 3.12, where mismatched node orderings lead to link *R1→R2* being tagged *up* in the up-tree and *down* in the down-tree. Thus, the construction of the two trees must be in lockstep, guaranteeing matched ordering by construction.

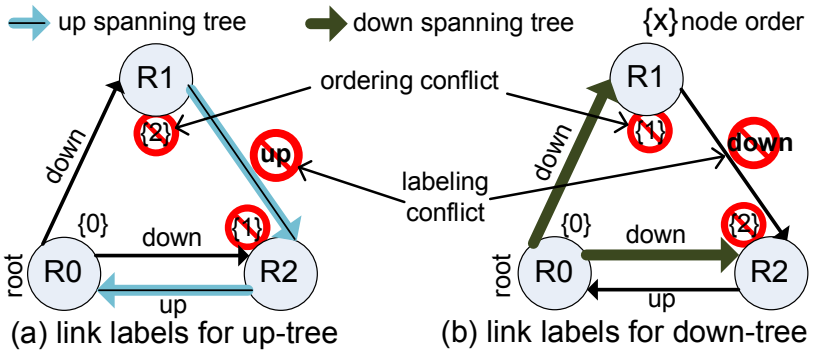


Figure 3.12 Independent construction of up-tree and down-tree causes inconsistent labeling of link *R1→R2*. (a) Up-tree: link is towards a node that is closer to the root; hence tagged *up*. (b) Down-tree: link is between nodes at the same level; hence tagged arbitrarily as *down*.

Matched Ordering by Construction

uDIREC's routing algorithm builds the two trees using a breadth-first search, but advances the construction of the two trees in lockstep, expanding to new nodes only if a node order matching on both trees exists. To this end, each leaf node reached in the network expands the two trees to its descendants only when the node itself is reachable by both the up-tree and the down-tree. Otherwise, the up-tree (down-tree) construction is halted until both tree constructions reach that node. All nodes that are reachable by both the up-tree and the down-tree can communicate among themselves by enabling deadlock-free routes. All other unreachable nodes timeout after waiting for one or both tree(s) to reach them. As shown in Figure 3.14a, starting from the root node (*R0*), both the up-tree and the down-tree expand to *R2* using *bidirectional link R0↔R2*; hence *R2* can expand to its descendants. At the same time, the down-tree expands to *R1* and halts at *R1* for the up-tree to catch-up. In the next iteration, *R2* expands the up-tree to *R1*, cancelling the halting status of *R1*. At this time, both trees reach all nodes, and hence the network is connected and deadlock-free. For deadlock-freedom, all routes traversing a *down* link followed by an *up* link (*down→up* turn) are disallowed. Finally, a route search algorithm finds the minimal route(s) between each source-destination pair.

Property 1. uDIREC's routing algorithm provides deadlock-free connectivity among all nodes reachable by both up-tree and down-tree.

Proof's outline. With consistent node ordering, any deadlock causing cycle will contain at least one *up→down* turn and one *down→up* turn [29]. Since uDIREC's routing algorithm guarantees consistent tagging of links, all dependency cycles are broken by disallowing just *down→up* turns. Additionally, all the nodes connected by the spanning trees can communicate deadlock-free, as any node can reach all other nodes by going to the root node first,

```
ROOT = pick_root()
newly_reached = up-tree = down-tree = ROOT;
/*Begin up-tree and down-tree construction*/
do:
| for(NODE in newly_reached):
| | up-tree += nodes_with_link_to(NODE)
| | down-tree += nodes_with_link_from(NODE)
| newly_reached = new_overlap(up-tree,down-tree)
while(newly_reached != NULL)
disable_unreached_nodes()
order = order_nodes_reachable_by_both_trees()
apply_down_up_turn_restrictions(order)
find_minimal_routes_with_turn_restrictions()
```

Figure 3.13 uDIREC's deadlock-free routing algorithm to determine deadlock-free routes in networks with unidirectional links. To guarantee a matched node ordering, nodes expand the trees to their neighbors only if both up-tree and down-tree have reached them. The resulting matched-ordering governs the turn restrictions, and the evaluated minimal routes adhere to these turn restrictions.

thus following *up* links first and *down* links afterwards. In any such routing path, there will be no disabled *down*→*up* turn.

Property 2. Routing configurations produced by uDIREC's routing algorithm perform at least as well as configurations generated by up*/down*, in any fault scenario.

Proof's outline. In the worst case, to provide consistent marking of links, both up-tree and down-tree can be build only via bidirectionally working links, as in up*/down*. Therefore, uDIREC's routing algorithm is always able to connect at least as many nodes as up*/down* and is able to provide at least as many deadlock-free routes as up*/down*.

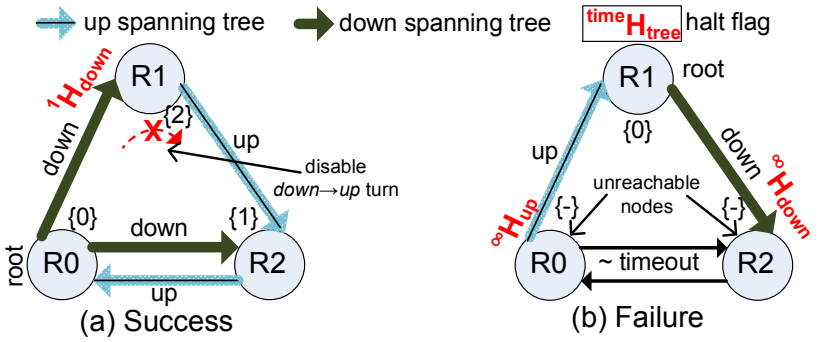


Figure 3.14 Growing the up-tree and down-tree in lockstep. The choice of root affects connectivity. (a) Success with root *R0*: both up-tree and down-tree connect all nodes with consistent labeling. (b) Failure with root *R1*: up-tree (down-tree) halted at *R0*(*R2*).

Root Node Selection

The structure of both trees greatly depends on the root node selection. However, as shown in Figure 3.14, this aspect may also affect the connectivity characteristics of the network. In this example, if instead of *R0* (Figure 3.14a), *R1* (Figure 3.14b) is chosen as root, uDIREC's routing algorithm is unable to find deadlock-free routes to any other node in the network. With *R1* as root in Figure 3.14b, the up-tree uses the link *R0*→*R1* to expand to *R0* and the down-tree takes the link *R1*→*R2* to expand to *R2*. Both trees halt at their frontier nodes (*R0* for up-tree; *R2* for down-tree) waiting for their counterpart trees. The algorithm terminates with *R1* connected to no other node, as the down-tree (up-tree) never reaches *R0* (*R2*) in this configuration. Therefore, optimal root selection can improve the connectivity characteristics of the network when using uDIREC's routing algorithm.

3.4.3 Reconfiguration

uDIREC designates any one node in the multi-core system as the "supervisor", which implements the reconfiguration algorithm in software. uDIREC takes advantage of the fact that our fault diagnosis scheme already stores the topology information in a software-maintained scoreboard at any one node. For simplicity of implementation, uDIREC designates the same node as the "supervisor". With this setup, only the supervisor node is entitled to make diagnostic decisions on the health of the NoC, and hence information about fault locations and the surviving topology is already present at the supervisor. In this manner, uDIREC avoids hardware overhead incurred by software-based reconfiguration solutions [110, 84] to reliably collect the topology information at a central node. Upon a new fault detection, the supervisor node transmits a reserved message to all routers/nodes in the system, informing them about recovery initiation. On receiving this message, all routers suspend their current operation and wait for routing function updates from the supervisor, while the nodes stop new packet injections. In the meantime, the supervisor computes deadlock-free routes for the surviving topology, using the routing algorithm described in the previous section. The computed tables are finally conveyed back to each router/node, which then resume normal operation.

As described in Section 3.4.2, network connectivity depends strongly on the choice of the root node. To this end, uDIREC's reconfiguration algorithm discovers the largest connected topology via an exhaustive search of the root node that maximizes network connectivity. Each node, in turn, is appointed as the temporary root node and the number of nodes it connects is calculated. The optimal root-selection is finalized when one of the

```
/* Root Selection by connectivity evaluation */
ROOT_win = -1; max_connectivity = 0
for(ROOT in all_nodes):
| connectivity = eval_uDIREC_connectivity(ROOT)
| if(connectivity == num_nodes):
| | ROOT_win = ROOT; break;
| if(connectivity > max_connectivity):
| | ROOT_win = ROOT;
| | max_connectivity = connectivity
/* Route Construction for winner root */
apply_uDIREC_routing_algo(ROOT_win)
```

Figure 3.15 uDIREC's reconfiguration algorithm. All nodes are tried as root, and the root that provides maximum connectivity, is chosen to build the new network. Within each root trial, uDIREC's novel unidirectional routing algorithm is leveraged to determine deadlock-free routes and determine connectivity.

following two conditions occur: (i) a root node that provides deadlock-free connectivity among all nodes is found; or (ii) all nodes have been considered as root node. At the end of the selection, a winner-root is determined, *i.e.,* the node that could connect the maximum number of nodes when chosen as root.

Reconfiguration duration. Considering all nodes as root is inefficient, and algorithmic optimizations are possible, but uDIREC trades reconfiguration time for simplicity of the algorithm. This is because permanent faults (even when up to tens or hundreds) are not frequent enough for reconfiguration duration to affect overall performance. Tree-based routing algorithms can be efficiently implemented in software, and typically take only hundreds of milliseconds to complete ($\sim$170ms [110]). Even though uDIREC requires multiple iterations of the tree-based routing algorithm, we expect the reconfiguration overhead to be within a few seconds at worst. Assuming an aggressive life-span of 2 years for high-end servers and consumer electronics, and 150 NoC faults in the worst case, an NoC would suffer 1 fault every 5 days. Therefore, an overhead of few seconds per fault manifestation is negligible.

Reconfiguration-induced deadlock. Reconfiguration can cause routing deadlocks even if both the initial (before fault manifestation) and final (after reconfiguration) routing functions are independently deadlock-free [76, 97]. uDIREC avoids such deadlocks by identifying the packets that request a turn that is illegal according to the updated routing function. These packets are then ejected to the network interface of the router, in which they are buffered during reconfiguration. After reconfiguration, these packets are re-injected into the network upon buffer availability. Other state-of-the-art reconfiguration techniques

[4, 100] utilize a similar technique to overcome reconfiguration-induced deadlocks.

Early diagnosis. Our diagnosis scheme pinpoints the fault locations in the router datapath before the faults grow to fatal proportions. The first few faults in the router datapath are corrected by the end-to-end ECC in the process of obtaining diagnostic information about fault locations. This presents the opportunity to keep using the network (for sometime) even after the fault has been diagnosed, as the initial few faults in the router datapath are within the correction capacity of the ECC. The pre-emptive diagnosis enables the salvaging of the processor and memory state of the about-to-be-disconnected nodes while the network is still connected. Traditionally, computer architects have relied on check-pointing support [99] for this purpose, while a recent research proposal [32] adds emergency links to this end. Using this technique, it is possible to greatly simplify this additional reliability-specific hardware. uDIREC, however, does not guarantee the integrity of packets that are traversing the network after fault manifestation and before fault detection, and relies on orthogonal recovery schemes [85, 5, 90] for that. However, the property of early diagnosis can greatly reduce the likelihood of fatal data corruptions and reduce the reliance on such recovery schemes.

Optimal root and tree. The choice of root node also affects the network latency and throughput characteristics [105]. In addition, the performance of tree-based routing algorithms is sensitive to the way trees are grown (breadth-first vs. depth-first), and the order in which nodes are numbered [105]. Further, the surviving set of on-chip functionalities may also differ with different root selection and tree growth schemes. However, the corresponding analysis is beyond the scope of this project and we do not consider these metrics in the root selection or tree-building process. uDIREC chooses the optimal node solely on the basis of number of connected nodes, breaking ties by comparing statically assigned node IDs, while the trees are build in a breadth-first fashion.

3.4.4 Implementation

In a typical software-based reconfiguration scheme, such as the ones used for off-chip networks [110, 106], the surviving nodes first collaborate to choose the root node, following which, the surviving topology information is communicated to this root node. Finally, the newly computed routing tables are delivered to all the surviving nodes. Dedicated hardware resources are required to reliably communicate this information to and from the root node. Fortunately, uDIREC completely eliminates the need of collecting the NoC health status at a central node on each failure. The tight integration of the diagnosis and reconfiguration

scheme in uDIREC makes it possible to utilize the topology information already available with the supervisor node.

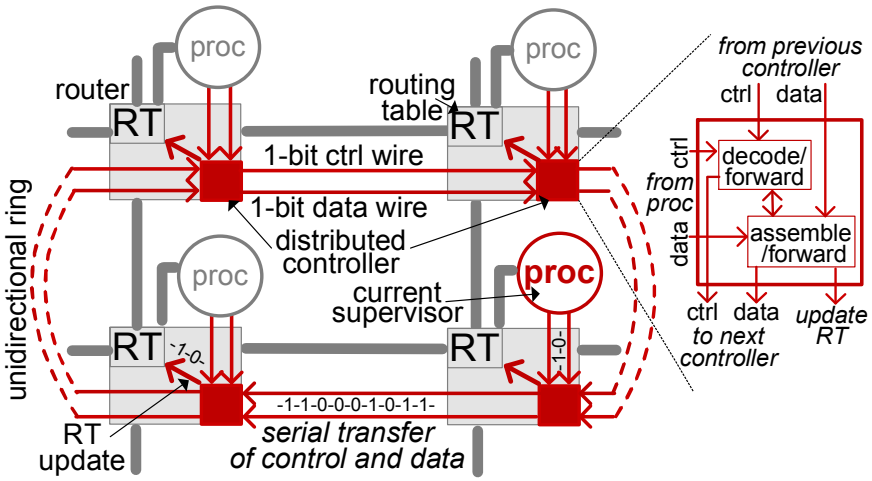


Figure 3.16 Distribution of the routing tables from the supervisor. One control and one data wire, organized in a unidirectional ring, are used to transfer routing tables to each NoC router in a serial fashion. Distributed controllers, one per router, snoop on the bit-stream broadcasted on the control wire and *decode* it to identify the data bit-stream relevant to the current router. The flagged bit-stream is then *assembled* and written into the routing table. As permanent faults are rare, this serial distribution poses insignificant performance overhead.

The overhead of routing table distribution can be drastically reduced by noting that permanent faults are rare occurrences and all reconfiguration-specific transmissions can be done serially over a single wire. To this end, uDIREC utilizes a combination of one control wire and one data wire, that are managed by distributed controllers (at each NoC router) organized in a ring topology. The control wire is utilized to notify all NoC routers of recovery initiation, and to set up the data communication between the supervisor and any particular router. Once the communication is set up, the data wire is used to transmit the routing table for that specific router, bit-by-bit. The process is repeated for all routers in the NoC: upon completion, an end-of-reconfiguration signal is broadcasted again via the same control wire. Specifically, a *decode* unit is included within each controller that snoops on the bit-stream broadcasted on the control wire and identifies data bit-streams relevant to the current router. Once a relevant data bit-stream is identified, the *assemble* unit gathers the information transmitted on the data wire and writes it to the routing table. Note that for a 64-node mesh NoC, the routing table at each node is 64 (destination nodes)$\times$4 (directions) bits. Thus, the routing information for the entire NoC is 2KB only, which should take insignificant time to transmit when compared to the 100s of milliseconds required for software-based route evaluation. The hardware implementation of the scheme just de-

scribed is shown in Figure 3.16. The 2-bit wide links are used only during reconfiguration, and are otherwise disabled with power-gating, greatly reducing both the risk of wearout faults and the power overhead. Further, due to their small area footprint, simple resilience schemes to protect them (TMR or ECC), do not add significant overall overhead.

We synthesized the baseline 64-bit channel mesh router using Synopsys DC [121], targeting the Artisan 45nm library. We also estimated the link wire area for the same using Orion2.0 [63] considering dimensions from an industrial chip [56]. The baseline router has a total area of $0.138mm^2$ (logic=$0.08mm^2$,wire=$0.06mm^2$). In comparison, the distributed controller logic required to implement the routing table distribution scheme is trivial, and the two additional distribution wires (unidirectional ring) lead to an area overhead of only 0.34% compared to this baseline NoC. For this study, we assume that the router wires and the two distribution wires are of the same length, and a unidirectional ring has $1/4^{th}$ the number of channels compared to a mesh.

3.4.5 Experimental Results

We evaluated uDIREC by modeling an NoC system in a cycle-accurate C++ simulator based on [28]. The baseline system is an 8x8 mesh network with a generic 4-stage pipeline with 2-message classes, 1-VC per message class routers. Each input channel is 64-bits wide and each VC buffer is 8-entry deep. For comparison, we also implemented Ariadne [4], which outperformed all previous on-chip reconfiguration solutions, *i.e.*, Vicis [39] and Immunet [100]. Ariadne [4] reports 40% latency improvement over Immunet, which falls back to a high-latency ring to provide connectivity, and 140% improvement over Vicis, which occasionally deadlocks. Since both Ariadne and Immunet guarantee connectivity if routes (using only bidirectional links) between pairs of nodes survive, they show identical packet delivery rates. They also deliver a higher fraction of packets compared to Vicis, especially at high number of faults when Vicis tends to deadlock. uDIREC achieves substantial improvements over Ariadne, and thus uDIREC's improvements over Vicis and Immunet are expected to be even more impressive. Note that Ariadne assumed a perfect diagnosis mechanism, so for a fair comparison, we have paired it with our diagnosis scheme, discussed in Section 3.3.

The uDIREC framework is analyzed with two types of workloads (Table 3.1a): synthetic uniform random traffic, as well as applications from the PARSEC suite [14]. PARSEC traces are obtained from Wisconsin Multifacet GEMS simulator [78] modeling a fault-free network and configured as detailed in Table 3.1b. After fault injections, we ignore messages originating from and destined to the disconnected nodes, and this could

lead to some evaluation inaccuracies for parallel collaborating benchmarks running on a partitioned multi-core. However, the traces are intended to subject the faulty NoC to the realistic burstiness of application traffic, and provide a simple and intuitive comparison. The metrics provide valuable insights considering that a particular fault manifestation in uDIREC and prior work(s) could lead to vastly different configurations in terms of number/location/functionality of working cores/IPs.

(a)

traffic	uniform PARSEC
packet	1flit (control) 5flits (data)
simulation	1M cycles
warm-up	10K cycles

(b)

processor	in-order SPARC
coherence	MOESI
L1 cache	Private: 32KB/node ways:2 latency:3
L2 cache	Shared: 1MB/node ways:16 latency:15

Table 3.1 (a) Simulation inputs. (b) GEMS configuration.

Fault injection. Our architectural-level fault injection technique randomly injects gate-level faults in network components with a uniform spatial distribution over their silicon area. Each fault location is then analyzed to map it to dysfunctional link(s), modeling our fault diagnosis scheme (Section 3.3). The links that are marked as dysfunctional are bypassed using the evaluated route-reconfiguration schemes. We injected a varying number of permanent faults (0-60 transistor failures) into the NoC infrastructure, and analyzed uDIREC's reliability and performance impact. For each number of transistor failures, the experiment was repeated 1,000 times with different fault spatial locations, selected based on a uniformly random function.

Scope of experiments. We restricted the scope of our experiments by injecting faults only in the NoC infrastructure because the system-level performance and reliability depends on the vast number of unrelated factors: fault location, fault timing, memory organization, programming model, processor and memory reliability schemes, and architecture specific characteristics. As a result, the surviving system functionality might not be directly comparable. Hence, we have provided a generalized evaluation across a wide range of faults, consisting of insightful performance (latency, throughput) and reliability (number of dropped nodes, packet delivery rate) metrics for fault-tolerant NoCs.

Reliability Evaluation

As faults accumulate, networks may become disconnected. Performance of parallel workloads running on a multi-core chip, with a faulty network, directly depends on the number

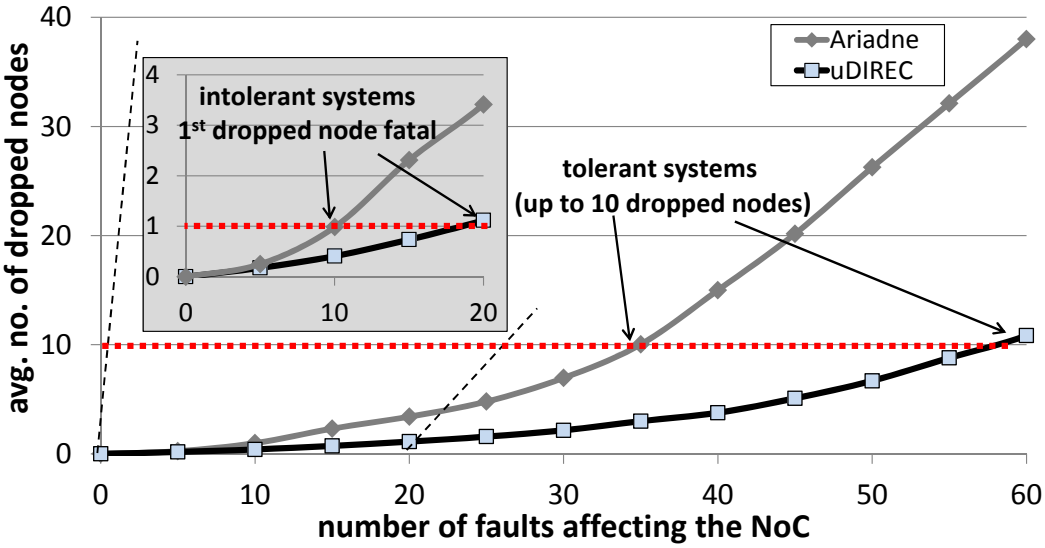


Figure 3.17 Average number of dropped nodes. Compared to Ariadne, uDIREC drops 3× fewer nodes and approximately doubles the number of faults tolerated before the same number of nodes are dropped.

of connected processing elements (PEs) and other on-chip functionality, e.g., memory controllers, caches. Thus, the ability of an algorithm to maximize the connectivity of a faulty network is critical. In this section, we study: a) average number of dropped nodes, *i.e.*, nodes not part of the largest connected sub-network, and b) the packet delivery rate for uniform traffic, as faults accumulate in the NoC.

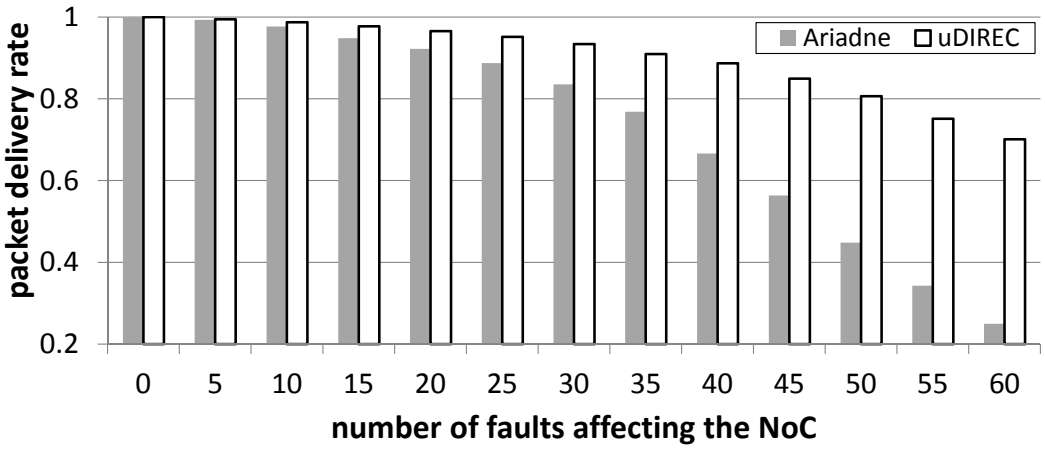


Figure 3.18 Packet delivery rate. Higher network partitioning in Ariadne causes steep decrease in delivery rate beyond 10 faults. In contrast, uDIREC degrade gracefully.

It can be noted in Figure 3.17 that Ariadne consistently drops over 3× more nodes than uDIREC. Even with just a few faults (5-20), uDIREC shows substantial improvement over Ariadne, dropping 1 node against Ariadne's 3 at 20 faults, as shown in the zoomed section

of Figure 3.17. Further, Figure 3.18 reports the number of packet delivered as a percentage of packets generated in the network. As packets, with source and destination in disconnected subnetworks, cannot be delivered, this metric is an indication of partitioning in the NoC. From the figure, both uDIREC and Ariadne deliver the majority (or all) of packets up to 10 faults. Beyond 15 faults, Ariadne starts partitioning into multiple sub-networks, and hence its delivery rate drops substantially below that of uDIREC. At 25 faults, uDIREC delivers 7% more packets than Ariadne, and the gain goes up to $\sim 3\times$ at 60 faults. uDIREC's ability to deliver a large fraction of packets even at a high number of faults makes it an excellent solution for fault-ridden NoCs.

Performance Evaluation

After reconfiguration, the NoC should keep functioning adequately with only a graceful performance degradation. In our evaluation, we report two performance metrics: average network latency and saturation throughput, after the network is affected by transistor faults. Latency and throughput measures are reported for the largest connected sub-network, assuming nodes disconnected from each other cannot work collaboratively. First, we report the average zero-load network latency, that is, the steady-state latency of a lightly loaded network (0.03 flits injected per cycle per node).

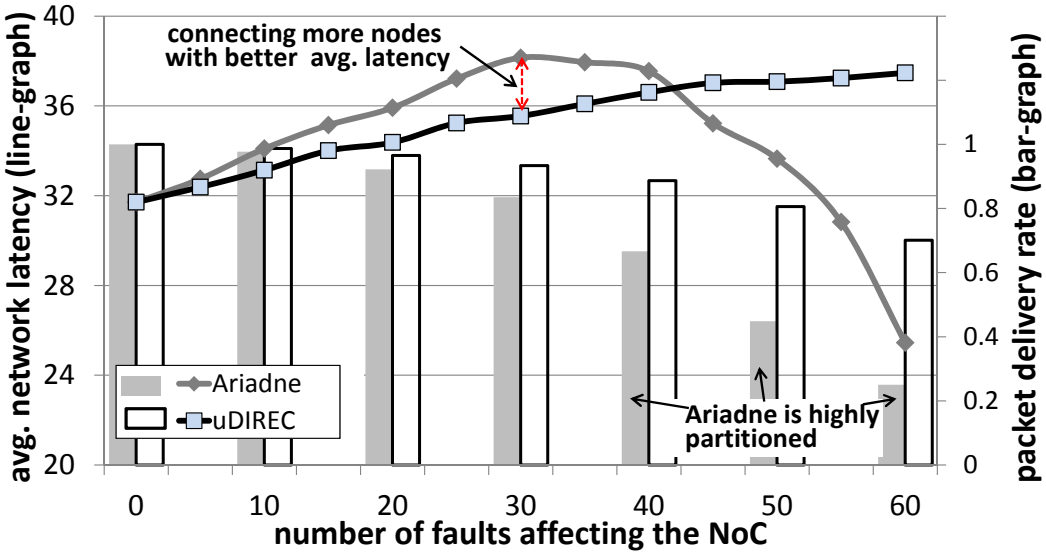


Figure 3.19 Zero load latency. Initially, latency degrades more gracefully for uDIREC as it provides path diversity. Beyond 40 faults, Ariadne becomes highly partitioned and hence latency drops steeply. Packet delivery rate is much lower for Ariadne, confirming its excessive partitioning.

Analyzing Figure 3.19, both uDIREC and Ariadne initially show an increase in la-

tency because the number of paths affected increases with increasing faults, while very few nodes are disconnected from the network. uDIREC, however, degrades more gracefully and at 30 faults, uDIREC on average has 7% lower latency than Ariadne. Beyond that point, Ariadne's latency starts falling. This effect is easily understood by analyzing Figure 3.17 beyond the 30-faults mark: a substantial number of nodes are dropped by Ariadne. As a result, packets now travel shorter routes to their destinations, and thus the average network latency is reduced. The crossover between the two latency graphs is at ∼40 faults, and as noted from the packet delivery rate chart copied over in Figure 3.19, the difference between the delivery rate of the two techniques is large (35% more in uDIREC) and grows rapidly beyond that point. We observed similar latency trends in simulations using PARSEC benchmark traces.

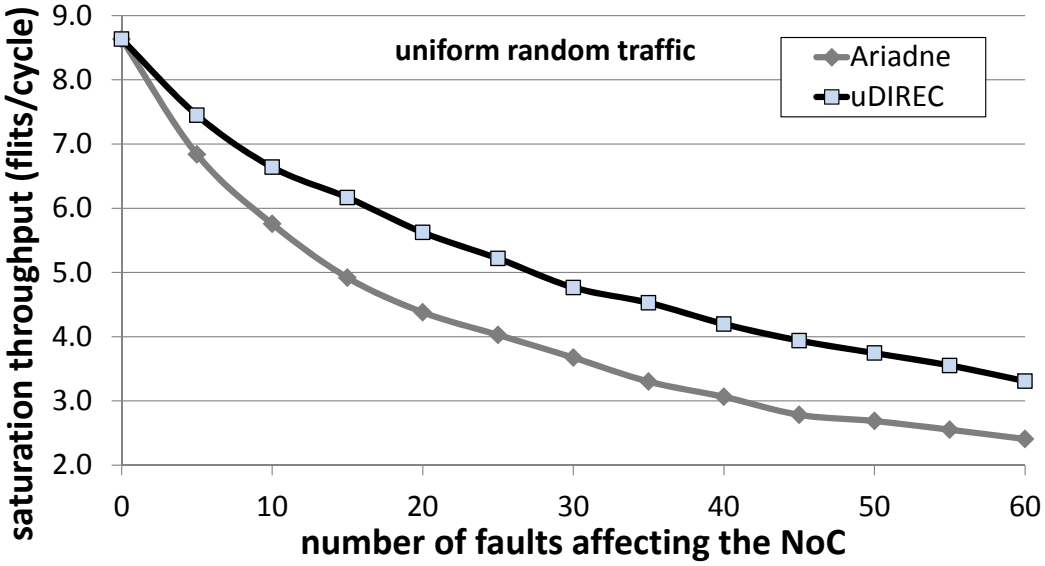


Figure 3.20 Saturation throughput. uDIREC consistently delivers more packets per cycle as it uses additional unidirectional links to connect more nodes and enable more routes.

In addition to latency, we also measured saturation throughput of the largest surviving sub-network. Figure 3.20 plots the packet throughput delivered by the network. uDIREC consistently delivers more packets per cycle as it uses additional unidirectional links to connect more nodes and enables more routes. uDIREC delivers 25% more packets per cycle than Ariadne at 15 faults. This advantage further increases to 39% at 60 faults.

Performance, energy and fault-tolerance (PEF) metric [68]. Traditional NoC metrics, such as energy-delay product (EDP), do not capture the importance of reliability and its relation to both performance and power. To this end, [68] proposed a composite metric which unifies all three components: latency, energy, and fault-tolerance. They defined PEF

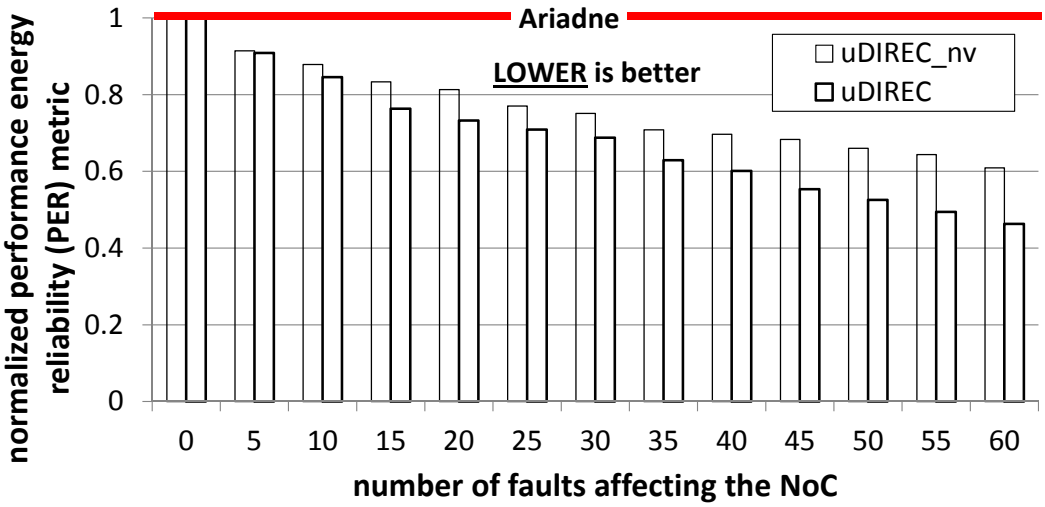


Figure 3.21 Performance-energy-fault tolerance (PEF) metric [68]. uDIREC monotonically improves with increasing faults. uDIREC (uDIREC_nv) shows 2× (1.6×) improvement at 60 faults.

as in Equation 3.1. In a fault-free network, *Packet Delivery Rate*=1; thus, PEF becomes equal to EDP. We assume total network energy to be proportional to the number of active routers in the network, as we use identical routers in our setup. Therefore, we estimate per packet energy to be proportional to the fraction of number of active routers by packet throughput, as shown in Equation 3.2. Figure 3.21 shows PEF values for uDIREC variants normalized against PEF values for Ariadne. Note that a *lower* value of PEF is *better*. The relative difference in the PEF values between uDIREC variants and Ariadne monotonically increases with the increasing number of faults. At 15 faults, uDIREC (uDIREC_nv) has 24% (17%) lower PEF than Ariadne, while showing more than 2× (1.6×) improvement at 60 faults. The reported PEF values confirm the benefits of using uDIREC across a wide range of fault rates. At few faults, the additional paths provided by uDIREC lead to reduced latency, while at higher number of faults, uDIREC delivers a greater fraction of the packets to their intended destinations.

$$PEF = \frac{(Average\ Latency) \times (Energy\ per\ Packet)}{Packet\ Delivery\ Rate} \tag{3.1}$$

$$Energy\ Per\ Packet \propto \frac{Number\ of\ Active\ Routers}{Packet\ Throughput} \tag{3.2}$$

3.5 Minimal-Impact Reconfiguration with BLINC

Current solutions for permanent fault circumvention, including the one described in the Section 3.4, often require suspending the normal network activity while executing a centralized and global-impact reconfiguration algorithm. Network activity is suspended during reconfiguration because new routes may conflict with old routes, possibly triggering a deadlock situation. In addition, the change in routes can lead to loss of data. These techniques are based on the assumption that fault occurrences are rare events, and orthogonal data recovery mechanisms can be leveraged without much impact on correctness or performance. Thus, they only strive to provide optimal or quasi-optimal post-fault routing configurations, at a high reconfiguration latency cost. For instance, reconfiguration takes between a 1K and 10K clock cycles for an 8×8 mesh with dedicated hardware [127, 4]. When reconfiguration is conducted in software, it takes substantially longer [42]. Overall, these solutions fail to provide non-stop operation with no data loss under faults; something that a digital system expects from its communication infrastructure.

To this end, we propose a **B**risk and **L**imited-**I**mpact **N**oC routing re-**C**onfiguration (BLINC) algorithm. BLINC deploys a topology-agnostic routing algorithm, which provides maximal connectivity and deadlock-freedom. The algorithm leverages a novel representation of the network topology, which allows to quickly perform reconfiguration locally, affecting very few routers. The representation consists of routing metadata stored at each router in a distributed fashion and updated upon each reconfiguration event through neighbor-to-neighbor updates. The metadata is used to compute alternative (*emergency*) routes, which affect only a few routers in most cases and can be quickly deployed upon a failure. BLINC maintains the deadlock-free network connectivity at all times, if at all possible.

We use the BLINC algorithm to develop a transparent reliability solution for NoCs, based on aggressive online testing and failure prevention. In this framework, individual *datapath segments* (Section 3.3.1) are taken offline and tested to evaluate if they are close to failing, in which case they are disabled. BLINC allows the quick movement of datapath segments offline and back online, and it provides a first-response routing solution when a failure is deemed imminent. These capabilities allow this framework to operate uninterrupted and without data loss through testing and fault reconfiguration. Experimental results show an 80% reduction in the average number of routers affected by a reconfiguration event, when compared to state-of-the-art techniques. BLINC enables negligible performance degradation in the non-stop detection and reconfiguration solution, while solutions based on current techniques suffer a 17-fold latency increase.

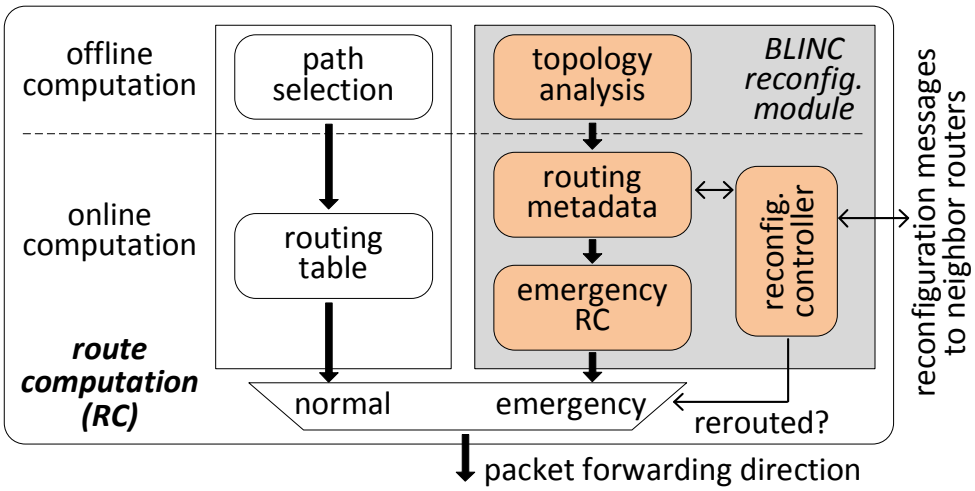


Figure 3.22 Route computation. BLINC deploys two route computation components: routing table (white) and reconfiguration module (gray). On fault occurrence, invalid routes are immediately replaced by emergency routes, while a procedure running in the background figures out an optimized routing configuration. The optimized configuration for the new topology is updated after the background procedure completes.

3.5.1 Methodology

In distributed routing, forwarding directions are selected locally at each router. Thus, rerouting entails recomputing the routing tables for all routers in the network [100, 39, 4]. We observe that the re-computation effort can be limited by utilizing pre-computed routing metadata, so to quickly pinpoint the affected routes. The immediate rerouting response from BLINC is fast and deadlock-free, but not necessarily minimal. However, a new minimal routing configuration is computed in background in software to reflect the topology change. Once this process is complete, the minimal routing tables are transferred to the nodes, replacing the temporary emergency routes.

Figure 3.22 shows the components required for the BLINC algorithm. The components shown with the white-background are part of baseline routers: routing tables are generated offline and ready when the network becomes operative. Upon a topology change, the gray-background reconfiguration module quickly calculates valid alternative routes, so that packets affected by the change can be safely sent through alternative (*emergency*) routes. Note that the majority of packets still utilize the original optimal routes.

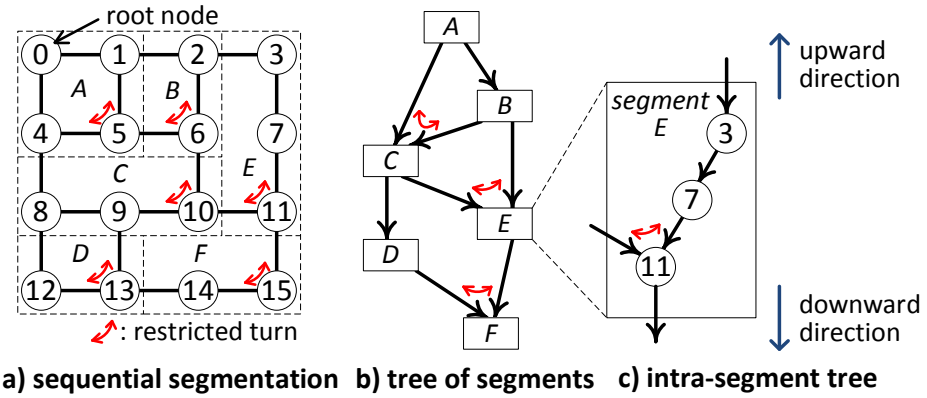


Figure 3.23 Network segmentation example. a) The segmentation process identifies a segment that starts and ends at the root node (nodes 0-root, 1, 4, and 5). Additional segments stem from nodes already a part of other segments (nodes 2 and 6). b) The corresponding segment-to-segment tree structure reflects this construction from root to leaves. c) Nodes within each segment are organized in an intra-segment tree. Note how each segment includes two connections to a parent segment.

Network Segmentation

It is possible to design alternative routes based on local information if a few constructs of segment-based routing [84] are leveraged. In segment-based routing, the entire network is partitioned into segments. The segmentation process starts by selecting a root node, and then identifying a segment as a sequence of nodes that start and end at the root node. Each subsequent segment is identified by building a sequence of nodes that start and end at nodes already included in the segmented portion of the network. Figure 3.23a shows an example of segmentation. Note that the definition of segments in segment-based routing corresponds to topology segments, rather than *datapath segments* as defined in Section 3.3.1. To avoid confusion, we will predominantly use the term *link* instead of *datapath segment*.

BLINC's off-line routing solution augments segment-based routing with an additional high-level tree structure, showing the connectivity between segments. The higher-level tree (Figure 3.23b) is built by traversing the network following adjacency, segment-by-segment starting from the root segment. Any two adjacent segments have a parent-child relationship if the segment under consideration (child segment) has links connecting to one or more segments already in the tree (parent segments). For example, once segment A and B have been considered, segment C is found to be a child of both. Nodes in the tree are ordered from root to leaves based on the order in which they are included in the tree (to improve readability, Figure 3.23b shows the segment order by using letters instead of numerals). Moreover, BLINC also introduce an ordering of the nodes within each segment by building *"intra-segment trees"* (see Figure 3.23c). Once all nodes are ordered, deadlock-free rout-

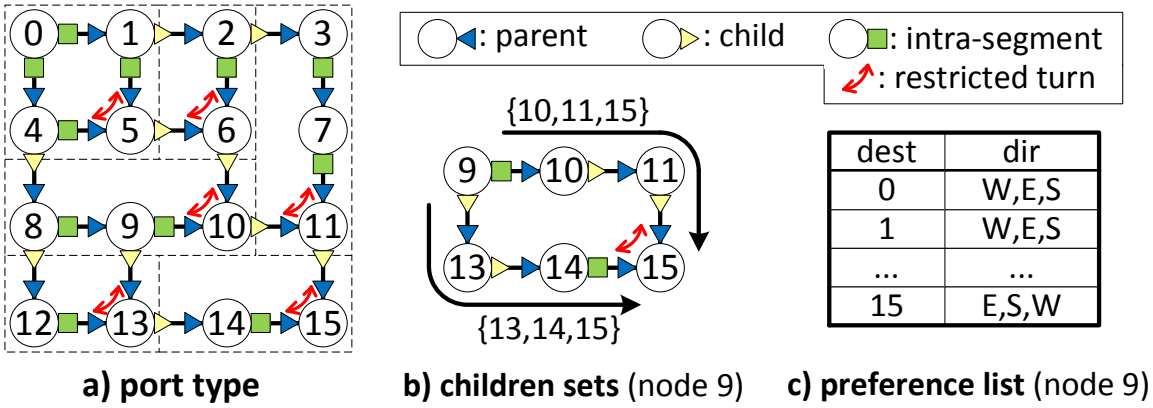


Figure 3.24 Routing metadata. a) Port types are assigned based on the network segmentation process. b) Each port in a node has an associated children set through child and intra-segment ports. c) Preference direction lists associated with each node are optional.

ing can be enforced by forbidding the turns around the highest order node. Note how the forbidden turns indicated in Figure 3.23a follow the approach just described: in segment E, the highest order node is 11 according to Figure 3.23c, and hence turns 10-11-7 and 7-11-10 are disabled.

To gather an intuitive understanding of our approach, notice that each segment is connected to segments closer to the root through two links. Thus, upon a link failure within a segment, it is possible to use the two links to reach the two disconnected portions of the segment. The tree structure remains unchanged, and it is sufficient to find a different route through one or more segments. As an example, in Figure 3.23a, assume that packets going from node 4 to node 13 traverse nodes 8 and 9. If the link 8-9 fails, it is possible to find an alternate route via 5, 6, 10 and 9.

Routing Metadata

Routing metadata is computed during network segmentation and is embedded at each router. It includes three types of information:

- **Port type**: A router port can be connected to a lower-order node (parent port), or to a higher-order node in the same segment (intra-segment port), or to higher-order node in a different segment (child port). Figure 3.24a indicates the type of all ports for a example network.
- **Children set**: the set of reachable nodes along downward routes for each port. Figure 3.24b shows the children sets for node 9.

- **Preference list** (optional): An ordered list indicating the preferred output directions. If available, this list is used to improve the quality of the emergency routes generated. Figure 3.24c shows an example of a preference list.

The following storage is required to store the routing metadata at each router: 2 bits to encode the port type, a bit array to indicate the children set for each port, and 6 bits per destination to encode the preference list (at most 3 directions, 2 bits to encode each direction). Thus, for an 8x8 mesh, at least 264 bits per router are required (384 additional bits if the preference list is provided).

3.5.2 Reconfiguration Algorithm

Upon a link failure, BLINC leverages the metadata described above to quickly generate alternative routes for the affected packets. Figure 3.25 illustrates the process with a high level schematic: each segment can be represented as a chain of nodes, and thus the segment affected by the failure will find itself disconnected. The localized reconfiguration process will re-establish connectivity for all the nodes by exploiting the additional routing paths that had earlier been disabled to avoid deadlock. Indeed, each segment contains exactly one disabled turn, which at this point will be re-enabled. Then all the children sets within the segment must be updated, so that each node is reachable from the segment boundary. This entails adding some children to some nodes and removing children from other nodes. In the example of Figure 3.25, Y was originally reachable via X only, but after the failure, it becomes reachable via Z instead. Finally, the additions and subtractions to the children sets are propagated outside the segment, until a common ancestor is reached. The reason for keeping the children sets updated is that, during a topology change, packets whose routing is affected by the change will use the children sets to determine their new paths. Note that a new optimal routing configuration is generated in the background in software; once computed, it overwrites all emergency routes. A reconfiguration example, showing all algorithm steps (described below), is presented in Figure 3.26.

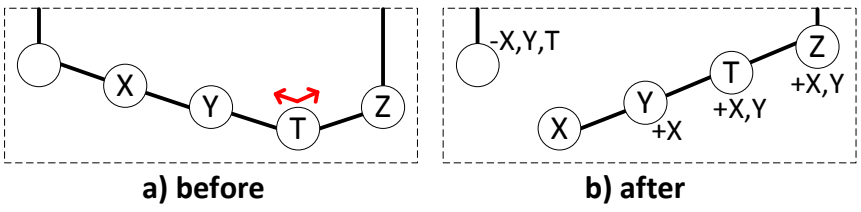


Figure 3.25 Children set update on reconfiguration.

Step 1. Disabling the link: Nodes stop sending packets through the faulty link.

Step 2. Re-enabling the turn: Before the fault, the node with the disabled turn (T in Figure 3.25) was the leaf in the intra-segment tree. After the fault, the portion of the segment between the turn-disabled node and the faulty link becomes isolated (portion between X and T in Figure 3.25). To reconnect that portion, i) the turn at T must be re-enabled, ii) the port types between the ends of each link of the isolated portion must be swapped (as shown in Figure 3.26.2 and then iii) an *"added-children set"* must be computed for each port in the isolated portion, which includes all the nodes downstream towards X.

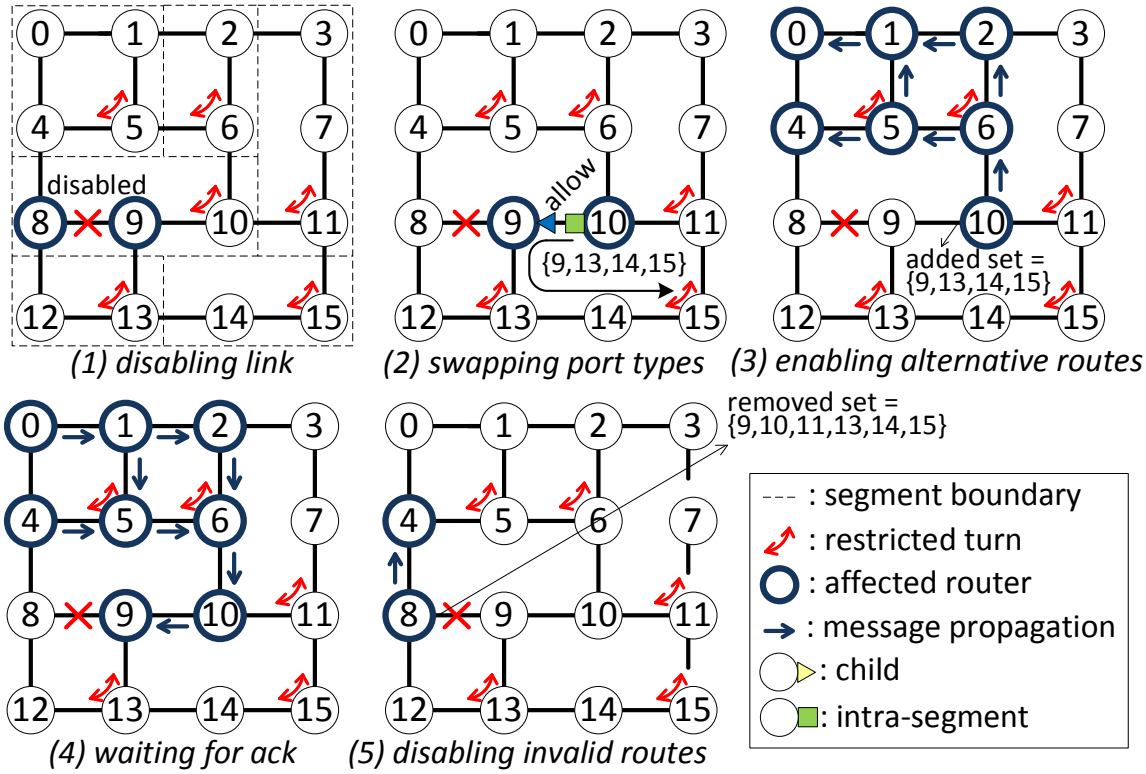


Figure 3.26 Reconfiguration example. The turn restriction in the segment with a fault is eliminated and the node next to the faulty link (node 9) is the new intra-segment leaf. An added-children set is created to indicate the new routes to reach the downward nodes after the fault, and it is propagated towards the root node. Once this step completes, acknowledgment messages are propagated back down. A similar process is followed to propagate removed-children sets starting from the other end of the fault (node 8). Finally, invalid routes are disabled.

Step 3. Enabling alternative routes: Once the added-children set for the turn-restricted node (T in Figure 3.25) is computed, the set is propagated toward the root node, across segment boundaries, to instruct every node of the new route to reach the destination next to the faulty link. For each node towards the root, the current children set of that node is compared against the incoming added-children set, and then the latter is reduced to include

only the nodes that are not already present in the children sets of the node under consideration. The process stops when the added-children set becomes empty. In Figure 3.26.3, the added-children set is propagated from node 10 all the way to the root node and node 4.

Step 4. Waiting for ack: The last recipients of the added-children set generate an acknowledgment message that is propagated all the way back to the node adjacent to the link failure.

Step 5. Disabling invalid routes: Finally, the other portion of the segment (the one connected to the parent port side of the faulty link) generates a *"removed-children set"* to indicate that it cannot reach the other end of the link. The removed-children set is propagated towards the root in a similar fashion to the added-children set: the only difference is that nodes are eliminated from the removed-children set when they already exist in the union of the children sets of all router's ports other than the port receiving the remove-children message.

3.5.3 Experimental Evaluation

BLINC was evaluated with a cycle-accurate NoC simulator [28]. The baseline network design uses wormhole, 3-stage pipelined routers with buffers for eight 64-bit flits per input port, connected in a mesh topology. Packets are 10 flits long, injected using random traffic at a 0.05 flits/cycle/router rate. Finally, the fault model is similar to the uDIREC scheme (Section 3.4.5). We evaluate three fault rates: 1%, 5% and 10% links were made faulty. 10 distinct faulty topologies were constructed out of a baseline mesh for each fault rate, and one more failure was added on top of the baseline at 10 different sites. In total, each fault rate is evaluated with 100 fault situations (10 baseline topologies x 10 failure locations).

Number of affected routers. The left part of Table 3.2 reports the average number of routers affected by a reconfiguration event over a range of fault densities and network sizes. The affected number of routers increases slowly with network size, showing that BLINC localizes the fault manifestation to a small region. Compared to existing methods [4, 118], BLINC achieves more than 80% reduction in the number of affected routers, across a wide range of fault densities.

Reconfiguration latency. Reconfiguration latency was computed assuming each node takes 5 cycles to process an add/remove children-set message and 1 cycle to propagate the acknowledgement messages. The reconfiguration latency is reported on the right part of Table 3.2. While reconfiguration latency is minimally sensitive to network size, it does show a steady increase with growing fault density. We believe this is due to the naturally occurring longer segments in faulty networks, which in turn, impose more hops in

the transmission of reconfiguration messages. Overall, BLINC's reconfiguration latency is 98% shorter than previous hardware-based techniques [4].

method	% faults	# affected routers			reconf. latency (cycles)		
		6×6	8×8	10×10	6×6	8×8	10×10
BLINC	0%	7.0	9.0	9.9	21.0	26.0	30.1
	1%	6.8	9.3	10.3	21.0	28.1	31.1
	5%	7.1	9.1	10.0	23.9	28.7	30.6
	10%	6.6	8.9	10.0	24.9	30.0	34.4
ARIADNE [4]	-	all routers			1.3K	4.1K	10K
OSR-Lite [118]	-	all routers			-	∼569*	-
Wachter et al.[127]	-	all routers			-	-	∼0.2K-208K

Table 3.2 Average number of affected routers and reconfiguration latency

3.6 Uninterrupted Availability with BLINC

To showcase the value of BLINC's approach, we evaluated its deployment in a fault detection and reconfiguration solution that provides uninterrupted availability. The methodology tests network resources aggressively to detect early signs of an upcoming fault. Each link (or datapath-segment), in turn, is taken offline for testing, which is performed through transmission of testing packets generated by a test pattern generator, such as the one in [52]. This approach can detect a majority of router faults [39]. For this methodology to be valuable, i) the network should be available and connected while a link is being tested and ii) the testing approach should be capable of detecting early signs of link failure (*e.g.*: increased delay, etc.), so that the network can reconfigure around the upcoming fault with no loss of packets. The first requirement can be accommodated by the BLINC algorithm: as shown in the previous section, it can provide emergency routes with minimal overhead. The latter requirement has been solved in the context of microprocessor designs [114, 132] but, to date, no solution of this kind has been developed for NoCs. BLINC can simply reconfigure the NoC to avoid the target link before the testing phase, and then reactivate the original routing after the test completes. If a link is found at risk of experiencing failure, emergency routing is maintained until a new segment-based routing solution can be computed in the background.

The testing flow is characterized by two parameters: the length of the test duration for each link (L) and the network testing rate (f), as illustrated in Figure 3.27. One complete testing cycle entails testing each link in turn. Note that the network should remain

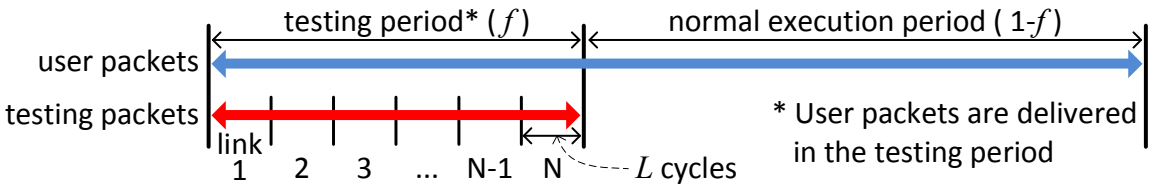


Figure 3.27 Online testing flow. The network should remain completely available even during testing, so that an aggressive testing frequency does not degrade network performance.

completely available even throughout testing.

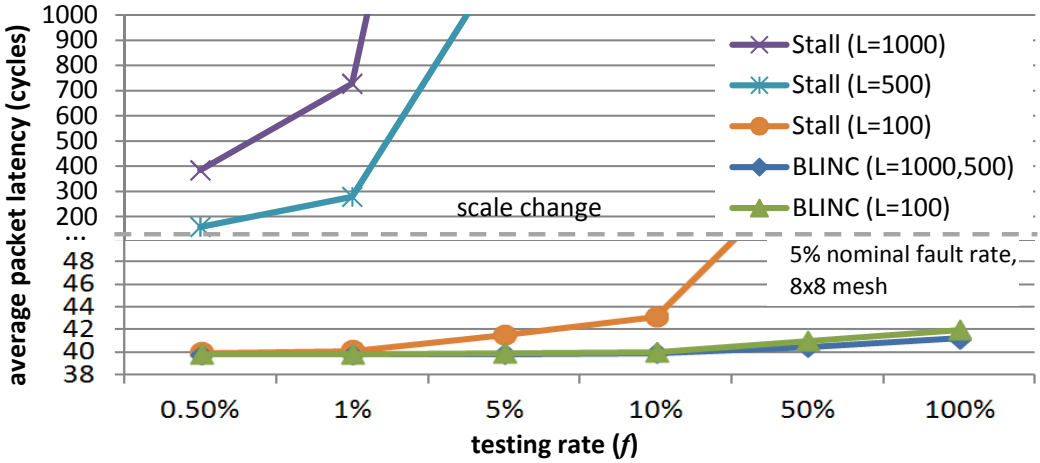


Figure 3.28 Average packet latency under online testing. BLINC reconfigures routing quickly, supporting online testing with only negligible performance degradation.

Figure 3.28 reports our findings: it plots average packet latency under a range of testing rates and test durations. Viable testing rates and durations were derived from [72, 52, 132]. For instance, the rightmost data point in the plot corresponds to one complete test period every 112,000 cycles. We compare these measurements against a routing solution (called *Stall*) that does not benefit from the BLINC solution. In Stall, packets are simply stalled in their buffers whenever they are trying to use a link undergoing testing. We only compared against Stall, because other solutions [100, 39, 4, 127] have reconfiguration latencies longer than the test durations employed. Analyzing Figure 3.28, average packet latency is minimally affected (6% in the worst case) by the ongoing testing and reconfiguration process when using BLINC, regardless of test durations. Moreover, Stall cannot provide uninterrupted availability beyond a test duration of 500 cycles, even at very low testing rates (the latency increase is 17-fold for L = 1,000 and f = 1%).

3.7 Related Work

Ensuring reliability in NoCs has been the subject of much previous research, focusing on a variety of aspects. Works such as [94, 85] focus on NoC protection against soft faults. Other methods enhance NoC reliability against permanent faults by enabling one or a combination of the following features: i) detection of erroneous behavior [85, 48], ii) diagnosis of fault site [48, 39, 101, 27, 69], iii) recovery from erroneous state [85], iv) system reconfiguration to bypass the permanent faults [100, 39, 4, 70, 21] or v) architectural protection for router logic [26, 39, 68].

Protection against soft-errors. The most common reliability techniques augment NoC packets with ECC information that can be checked on an end-to-end or a switch-to-switch basis [85, 35]. ECC codes can also correct a few bit errors, beyond which the system relies on costly data retransmission using a backup copy. Retransmission schemes cannot tackle all erroneous scenarios, especially the ones arising from control logic malfunction, *e.g.*, deadlock. [5] proposed to retransmit clean copies of data directly from the memory subsystem, alleviating the need of large backup buffers. However, the required changes to the cache coherence protocol present a design and verification challenge. Further, their solution also cannot recover from deadlock scenarios. A technique to recover from soft-errors in both the router's data-path and control logic was presented in [94]. It uses switch-to-switch ECC and retransmission for data-path errors, while the router control logic is protected with hardware checkers. Deadlocks due to soft-errors are broken by using additional buffers at each router port. However, this approach leads to a prohibitive area overhead (as was discussed in Section 2.5), especially for networks with large data packets. In contrast, ForEVeR++ does not require backup data storage for soft-error recovery, and it can also overcome forward progress errors such as deadlock. Additionally, ForEVeR++ also protects both the router data-path and control logic against soft-errors, at low design complexity.

Detection and diagnosis of permanent-faults. Forward Error Control (FEC) methods use ECC in order to detect and correct data faults during transmission by adding data redundancy to the packets [35]. However, these methods quickly lose their efficiency in presence of permanent faults, since the correction strength of ECC is quickly exhausted due to the recurring nature of errors arising from permanent faults.

Addressing both data and control faults, authors of [26] demonstrated a reliable router using N-Modular Redundancy (NMR). However, NMR imposes a prohibitively high area overhead with low returns on reliability. A diagnosis method was introduced in [39] where the authors proposed embedding Built-In-Self-Test (BIST) hardware for fault diagnosis and

a rerouting mechanism to bypass the defective parts. Their results indicate that each router incurs a 43% area overhead, with BIST contributing to the majority. Moreover, this method entails periodic intervals of dedicated NoC testing. Our scheme provides fine-grained fault diagnosis information covering both datapath and control-path faults. Additionally, it incurs low area overhead, and ensures no performance degradation or network operation interruption in the absence of errors.

Reconfiguration around permanent-faults. Our route-reconfiguration scheme, uDIREC, is orthogonal to architectural approaches that extend the lifetime of NoC links (ECC [85], reversible transmission [6], partially-faulty links [88]) or components ([26, 68]). When the number of faults affecting a link/component are beyond repair using such approaches, the corresponding link/component can be switched off, and traffic re-routed around it using uDIREC. With uDIREC, we specifically investigate fine-resolution diagnosis and route-reconfiguration to cope with permanent faults.

During route-reconfiguration, a new set of deadlock-free routes are generated to replace the current ones, whenever a new fault is detected. The unpredictable number and location of fault occurrences cause reconfiguration solutions, which are designed for a bounded number [49, 51] or constrained pattern [22, 46, 41] of faults, to be unfit for NoCs. Therefore, we compare uDIREC only against solutions that do not put constraints on number and location of faults. Table 3.3 presents a qualitative comparison of the algorithms in this domain. All previous reconfiguration algorithms, either off-chip [110, 84, 106] or on-chip [100, 39, 4, 21], are limited to the granularity of a bidirectional link, if not any coarser. Therefore, they are limited by the shortcomings of the *Coarse_FM*, failing to capitalize on performance and reliability benefits of using the *Fine_FM*. This is confirmed by the fact that uDIREC drops less than $1/3^{rd}$ of the nodes when compared to the best performing prior-art [4].

Except for Vicis [39], which uses a costly BIST (10% overhead [33]) unit for diagnosis, no other previous solution presented a unified approach to diagnosis and reconfiguration. Typically, standalone route-reconfiguration schemes assume an ideal accuracy diagnosis scheme, which either localizes a fault to an entire link/router [110, 84, 106, 100], or to a *datapath segment pair* [4].

uDIREC uses simple hardware additions to assist its software-based reconfiguration, incurring the lowest area cost among the route-reconfiguration schemes. Implementing off-chip reconfiguration schemes requires dedicated reliable resources for the collection of the surviving topology and the distribution of routing tables, to and from a central node, respectively. Ariadne [4] reports that the software-managed reconfiguration algorithms for off-chip networks, lead to 23.2% area overhead, if implemented on-chip without any

solution	diagnosis support	resolution		node-drop rate	reconf area(%)
		diagnosis	reconf		
off-chip	NO	–	bi-link	high, >3×	23
Immunet	NO	–	bi-link	high, >3×	6
Vicis	**YES**	sgmt-pair	bi-link	high,dlock	1.5
Ariadne	NO	–	bi-link	high, >3×	2
uDIREC	**YES**	**segment**	**u-link**	**low, 1×**	**<1**

Table 3.3 uDIREC's comparison with other route-reconfiguration solutions. uDirec provides unified fault diagnosis and reconfiguration at fine granularity, resulting in better reliability characteristics. Moreover, the hardware structures required for uDIREC, are small and simple. The area numbers for schemes other than uDIREC are as reported in prior-work [4].

modifications. Immunet [100], on the other hand, ensures reliable deadlock-free routes by reserving an escape virtual network with large buffers, which, in the worst case, reconfigures to form a unidirectional ring of surviving nodes. Vicis [39] leverages a bulky BIST unit at each router. In addition, the routing algorithm that Vicis implements is not deadlock-free, and it often deadlocks at high number of faults.

Quick and minimalistic reconfiguration. Global reconfiguration solutions [100, 39, 4, 127] can tolerate an arbitrary number of faults before the network gets segmented, but suffer from high reconfiguration overhead. In contrast, local reconfiguration solutions [133, 36] leverage bypass rules to quickly reroute around faults using local connectivity information. However, due to the localized nature of these solutions, they can only sustain a few faults without causing livelock or deadlock.

Quick routing reconfiguration has been investigated mostly for off-chip interconnection networks, such as local area networks (LANs). [20] and [111] propose dynamic, progressive reconfiguration procedures to this end. However, they only discuss reconfiguration principles without investigating the details and hardware required to support the reconfiguration procedure. In the on-chip networks domain, OSR-Lite [119] proposes a fast reconfiguration solution utilizing resources to support two routing-computation logic sets based on [102], with only one of them active at a time. Upon a fault occurrence, a central manager calculates the new replacement routes, while the old ones are still in use, then the two are swapped. While OSR-Lite is reported to be faster than hardware solutions, the dedicated central manager is a single-point of failure. [123] improves [119] with a disconnection-rescuing algorithm, but it still misses potential connections due to its limited routing capability. [44] proposes a time/space-efficient reconfigurable routing algorithm, but it does not show its applicability to fault tolerance. In Table 3.4, we present a comparison of relevant routing reconfiguration techniques. Our proposed solution, BLINC, is quick and provides high tolerance against a wide range of faults.

method	context	computation	impact	speed	fault tolerance
BLINC (our solution)	on-chip	hardware	**local**	**very fast**	**high**
ARIADNE [4]	on-chip	hardware	global	fast	**high**
MD [36]	on-chip	hardware	**local**	**very fast**	low
Sem-Jacobsen et al. [111]	off-chip	software	**local**	slow	**high**
OSR-Lite [119, 123]	on-chip	software	global	moderate	moderate

Table 3.4 BLINC's comparison with other route-reconfiguration techniques

3.8 Summary

This chapter described solutions to protect an NoC against both soft-errors and permanent faults. Protection against soft-errors requires detection capabilities, followed by a recovery procedure. For permanent faults, however, diagnosis of the fault site is essential, in addition to fault detection. Once a faulty component is identified, the reconfiguration procedure either replaces the malfunctioning component with a healthy substitute, or it disables the faulty component permanently, while leveraging the redundancy in the system to still continue operation, although at a lower performance level.

We first presented ForEVeR++ that leverages the ForEVeR infrastructure (Chapter 2) for protecting the NoC against soft-errors. ForEVeR++ protects the NoC against soft-errors affecting all component types (control, datapath and links). To this end, ForEVeR++ leverages the ForEVeR monitoring hardware to detect soft-error manifestations, in addition to design errors. Recovery of the soft-error affected packets is guaranteed by building resiliency features into our checker network. ForEVeR incurs a minimal performance penalty up to a flit error rate of 0.01% in lightly loaded networks, while incurring an area cost of only 6% over the baseline ForEVeR design.

We then introduced a novel method to comprehensively detect and diagnose permanent faults in on-chip networks. We present a novel representation of the router microarchitecture, partitioning it into *datapath-segments* for the purpose of fine-grained diagnosis. The high-resolution diagnosis information is then utilized by our route-reconfiguration solution (uDIREC) to replace the defective components sparingly. Our diagnosis method imposes no performance overhead during error-free network operation. Additionally, the system suffers no down time on error-detection, while our diagnosis scheme is passive, introducing no dedicated test traffic into the network. Our solution is able to achieve 98% diagnosis accuracy by monitoring only 15 erroneous transmissions. In addition, the area overhead is kept as low as 2.7%.

We then presented uDIREC, a solution for reliable operation of NoCs providing graceful performance degradation, even in presence of a large number of faults. uDIREC

leverages a novel deadlock-free routing algorithm to maximally utilize all the working links in the NoC. Moreover, uDIREC incorporates a software-based fault diagnosis and reconfiguration algorithm that places no restriction on topology, router architecture or the number and location of faults. Experimental analysis shows that, for a 64-node NoC at 15 faults, uDIREC drops 68% fewer nodes and provides 25% higher bandwidth over state-of-the-art reliability solutions. A combined performance, energy and fault-tolerance metric that integrates energy-delay product with packet delivery rate, reports a 24% improvement at 15 faults, which more than doubles beyond 50 faults, showing that uDIREC is beneficial over a wide range of fault rates.

Finally we presented BLINC, a brisk and local-impact NoC routing reconfiguration algorithm. BLINC utilizes a combination of online route computation procedures for immediate response, paired with an optimal offline solution for long term routing. To achieve its goal, BLINC employs compact and easy-to-manipulate routing metadata. Our evaluation shows more than 80% reduction in the number of routers affected by reconfiguration, and 98% reduction in reconfiguration latency, compared to state-of-the-art solutions. We also discussed how BLINC enables uninterrupted availability for networks-on-chip, by allowing individual network links to be taken offline for testing at high frequency. BLINC maintains stable network performance with only a 6% increase in latency during testing, in contrast with a 17-fold latency increase for a baseline approach that stalls one link segment at a time.

Chapter 4

Addressing Excessive Power Dissipation

Recent studies have shown that power-efficient performance improvements can be achieved by designing application-specific hardware components that accelerate certain types of computations. Research in this area has focused on integrating and utilizing several cores implementing a diverse set of solution points in the performance, application specificity and power dissipation space into heterogeneous Systems-on-Chip (SoCs). As the diversity and quantity of heterogeneous components that make it into future CMPs and SoCs increase, the interconnect that binds them together must also evolve to accommodate this shift in the design paradigm. Specifically, the interconnect should be able to deliver reasonable performance under a power budget across a diverse set of workloads. It should also be able to monitor its own health by keeping the power consumption under check: possibly by turning off unused components. A few approaches to designing Networks-on-Chip (NoCs) for heterogeneous platforms have recently been proposed in literature, focusing on tunable design and synthesis processes that optimize architectural parameters for a target set of applications. This chapter proposes a new design framework that enables the NoCs to be reconfigured at runtime, based on changing application and hardware configurations. This design methodology allows heterogeneous system designers to quickly deploy on-chip interconnects providing near-optimal communication in a wide range of architectures running a variety of workloads. This differs greatly from previous approaches that require the application's characteristics to be known at design time and remain static throughout its execution.

From the perspective of NoCs' runtime health, this chapter focuses on avoiding excessive power dissipation. The NoC architectures presented in this chapter reconfigure their topology and routing dynamically to save power, optimizing based on the applications' communication characteristics. During reconfiguration, unused or under-utilized datapath-segments (remember the definition of datapath segment from Section 3.3.1) or even entire routers within the NoC are switched off, and routing is re-configured to better manage active paths. The reconfiguration mechanism is implemented in a low-overhead distributed

hardware that allows for quick reconfigurations without affecting network performance. All the heuristics/implementations presented guarantee that the network never enters an erroneous state, such as deadlock or disconnection. Using such a framework, designers will be able to provide fine-grained optimization mechanisms for pushing power-conscious operation to the limit.

4.1 Power-Aware NoC with Panthre

NoCs are scalable and flexible, however, they are crippled by excessive power consumption [56, 130]. Particularly problematic for NoC structures is leakage power, which is dissipated regardless of communication activity or lack thereof. At high network utilization, static power may comprise more than 74% of the total NoC power at a 22nm technology node [120], and this figure is expected to increase in future technology generations. At low network utilization, leakage power is an even higher fraction of the total power budget for the NoC. With growing system integration, larger and larger portions of the NoC will be only lightly used at any point in time, with the lightly used set varying with each application and even within a single application over time.

Power-gating [57] is a promising solution to minimize the dissipation of leakage power. However, conventional power-gating schemes [57] that opportunistically put components to sleep during periods of no activity are ineffective for distributed and shared resources, like an NoC. The problem is two-fold: i) even when lightly utilized, NoC components often do not observe long idle-periods, failing even to compensate for the energy spent in the power-gating event itself, and ii) packets that encounter sleeping components in their paths accrue latencies due to wake-up delays. Power-gating at a finer granularity than entire routers [81] provides more sleeping opportunities. However, it further worsens the problem of accumulated wakeup latencies, as it puts components to sleep more aggressively. Early wakeup with lookahead routing was proposed to compensate for wakeup latency [80]. However, for a typical 2-stage pipeline router, lookahead can only hide a small fraction of the wakeup latency, which is typically many cycles. In our evaluations with multi-programmed workloads, we have identified that such conventional schemes often lead to significant application slowdown and net energy loss.

A workload stressing only a portion of the network creates opportunities for power-gating the remaining, lightly-used portions of the network. However, deterministic routing algorithms provide fixed routes among source-destination pairs, and in practice do not allow for isolation of any network component. A possible solution is to use an adaptive

routing algorithm and deflect packets toward active units when they encounter a sleeping component on their regular path. However, this approach requires additional resources to maintain deadlock freedom, which must be kept active at all times. Moreover, accruing multiple deflections leads to increased packet latency.

Our solution, called Panthre (for Power-aware NoC through Routing and Topology Reconfiguration), overcomes these issues by modifying routing paths periodically so to *exclude* lightly used portions of the topology. When Panthre's decision engine determines that the set of power-gated components must be updated, Panthre executes a route reconfiguration procedure that avoids the new set of power-gated components, while providing deadlock-free routes for all packets. This step eliminates deflections and the need for dedicated resources to support deadlock-freedom. Panthre leverages the rich set of alternate paths that are available in NoC fabrics to keep traffic away from sleeping components. In addition, it pro-actively adapts to application's demands by power-gating only those network components that are under-utilized. Similar to any route-reconfiguration approach, the smallest granularity of route manipulation with Panthre is a datapath segment, *i.e.*, a collection of components within an NoC router as defined in Section 3.3.1. Therefore, Panthre applies power-gating at the granularity of one datapath segment. This approach presents great saving potential for leakage power, as the five constituent datapath segments of a mesh router account for 99% of the router's static power (Section 4.2.1).

Naturally, Panthre leads to an increase in traffic on the links kept active by channeling traffic away from sleeping components. Therefore, it is essential for Panthre that substantial low-usage links exist in the NoC. To this end, we conducted a study, whose findings are plotted in Figure 4.1. The plot shows the contribution of network links to total network activity. Our testbed consisted of an 8x8 mesh CMP running a network-light multiprogrammed mix of applications from the SPEC CPU2006 suite. Links are sorted by increasing utilization during the execution, and the plot on the right indicates what fraction of network traffic (Y axis) was carried out by a given fraction of sorted links. The plot on the left is an enlargement of the contribution by the bottom 30% of active links: more than 20% of the links share only 5% of the traffic load on average, and only 10% of the traffic travels through the bottom 30% of used links. Beyond the 30 percentile of utilization, this disparity is no longer obvious, thus Panthre's goal is to identify and leverage the 30% least used links, so to maximize power savings without a significant load increase on active links.

Panthre deploys a simple and distributed framework for traffic activity collection and subsequent exclusion of low-usage components. Even though frequent decisions to power-gate components are made locally at each router, Panthre's novel reconfiguration solution

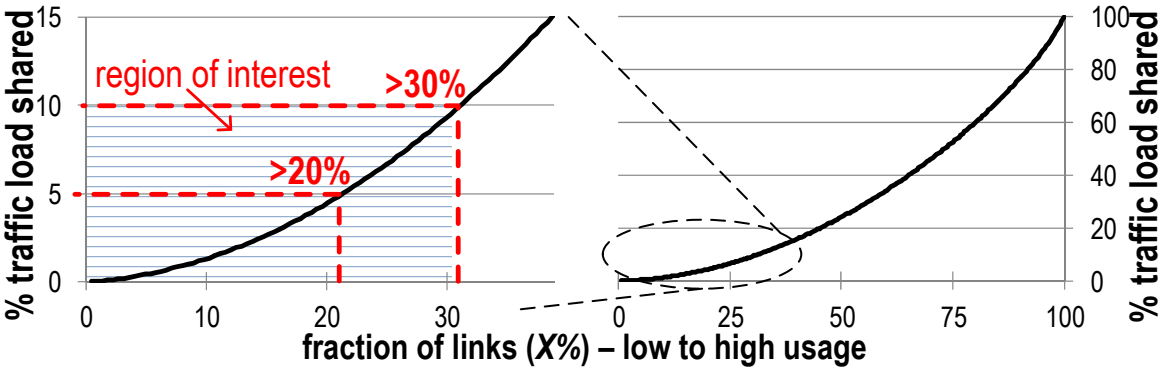


Figure 4.1 Fraction of traffic load shared by the least utilized links. The fraction of traffic transferred by the bottom 30% of used links is very small, thus Panthre targets this pool for power-gating without burdening other links with high load.

ensures uninterrupted operation with guaranteed connectivity and deadlock-freedom for the NoC topology globally. Panthre, by construction, is also free of reconfiguration-induced routing deadlocks [76], thus eliminating the need for expensive dedicated buffering, virtual channels (VCs), and a retransmission protocol [24, 104].

Panthre is the first solution that applies topology and routing reconfiguration in a distributed manner to maximize power savings by adapting routing decisions to changing application communication needs. Panthre's routing reconfiguration abilities are in stark contrast to previous proposals, which require statistics collection, decision making, route update and reconfiguration operation, all to be executed at a central node [104]. Such a process requires dedicated channels to communicate with the central entity and typically takes a long time, often requiring to suspend network operation. As a result, such centralized schemes provide little adaptivity and can only be applied at a coarse-granularity (entire routers), and only when communication patterns are known well in advance.

In our evaluation with multiprogrammed benchmarks running on a 8x8 mesh network, Panthre was able to reduce NoC power consumption by 14.5% on average for communication-light workloads, while causing less than a 2% application slowdown. In contrast, power-gating with lookahead [80, 81], leads to 9-11% application slowdown if implemented at a router level, while causing as much as 20% performance loss if fine-grained power-gating is applied. Finally, Panthre leakage power savings are 36.9% on average when 10-16 nodes of a 64-node CMP are communication-idle.

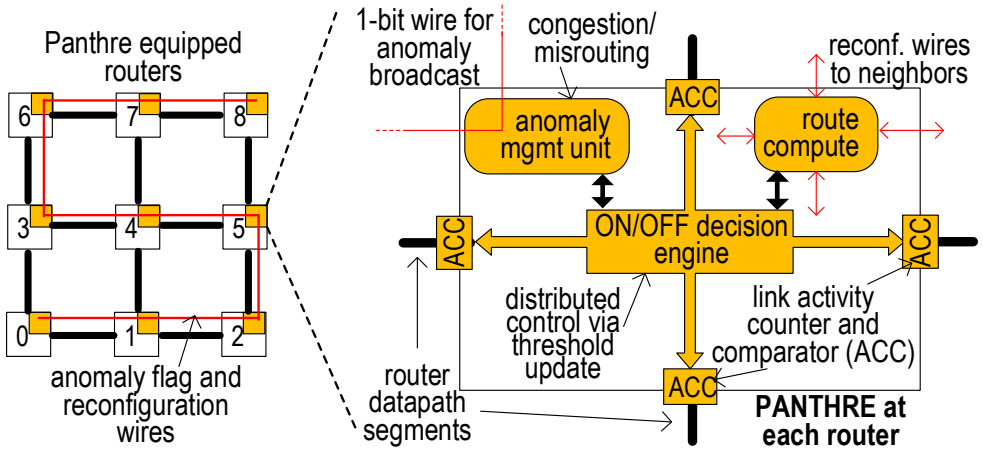


Figure 4.2 Overview of Panthre. Panthre consists of four components at each router: a) a usage activity count and compare (ACC) framework to identify lightly used components, b) an ON/OFF decision engine that determines the set of components to be power-gated, c) a Panthre-enabled route computation unit that can execute in the background without interrupting regular NoC operation, and d) an anomaly-based feedback algorithm that tracks application's needs dynamically and sends updates to the ON/OFF engine.

4.2 Panthre Design

Panthre consists of four components at each router: i) a component usage collection framework (ACC), ii) a lightweight ON/OFF decision engine that determines the set of links to power-gate based on local usage data, iii) a route compute module that updates the routing tables using broadcasts after each decision event, without interrupting normal NoC operation, and iv) a feedback-based anomaly-detection and management unit that communicates updates to the ON/OFF decision engine so that Panthre can adapt dynamically to changing communication patterns in the application over time. The four components of Panthre are highlighted in Figure 4.2. These four components are implemented using fast and lightweight distributed hardware, with minimal information communicated globally via a few single-bit wires. The lightweight distributed hardware allows Panthre to adjust to application's communication needs very quickly and without ever interrupting the normal network operation.

4.2.1 Fine-Grained Power Gating

Panthre provides fine-grained power-gating by allowing components to be excluded at the granularity of a single unidirectional link. Upon careful examination of routers' datapath, we identified that powering-down a unidirectional link between two routers is equivalent to powering down the corresponding crossbar contact and the output port at the upstream

router, the unidirectional link itself, and the input port, input buffer and crossbar contacts at the downstream router. We call this combined set of components, a *datapath segment*: it represents the smallest granularity at which routing-based reconfiguration can be applied for the purpose of power-gating. The concept of a datapath segment is illustrated in Figure 4.3 and was previously described in Chapter 3. Fortunately, crossbar, links and buffers consume most of the leakage power in a router, and they can all be powered off using our fine-grained power gating. DSENT [120] reports that 99% of the leakage power consumption of the baseline mesh router synthesized at 22nm can be attributed to its 5 datapath segments. The remaining 1% is attributed to shared units such as route computation and allocators. Note that Panthre is unable to exclude the local datapath segment (the one connecting to the local core) in absence of alternate paths for them to connect to the NoC. Therefore, in the rest of this chapter, we exclude these local datapath segments from our computations, and assume that orthogonal power-saving schemes are being deployed for them.

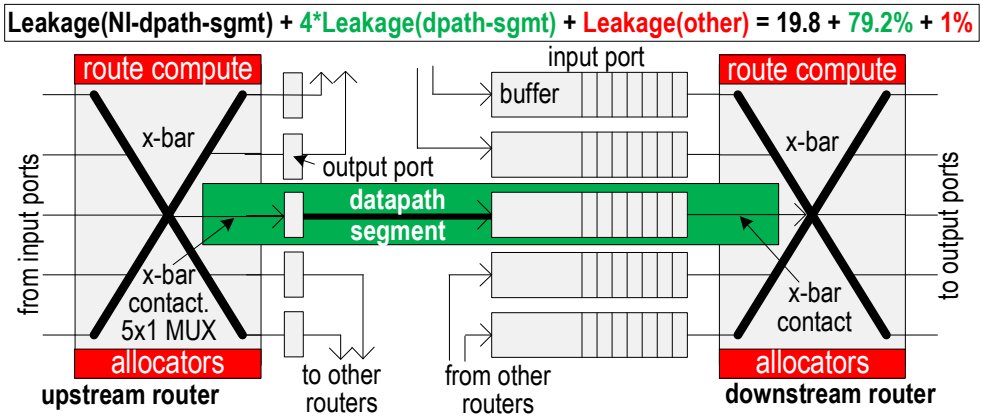


Figure 4.3 A datapath segment. 99% of the mesh router's leakage power is dissipated by its 5 datapath segments. Results are obtained from DSENT at 22nm node.

4.2.2 Execution Flow

Panthre's execution is divided into fixed time windows or *epochs*, with power-gating decisions made at the beginning of each epoch. Thus, Panthre ensures that power-gated components are off for at least an interval of one epoch. The distributed activity counter units (ACC, Figure 4.2) periodically collect datapath segment usage statistics by the means of simple counters: the data is then used to guide the decision process for the next epoch. Panthre's decision process is simple: all datapath-segments that experience activity below a threshold (A_{TH}) are put to sleep. Thereafter, a route update process updates routing tables

to operate the network in the new configuration. However, different communication loads require different A_{TH} for Panthre to be effective. A fixed value could lead to an excessive or insufficient number of power-gated segments. Therefore, we propose a threshold update algorithm that leverages feedback from the *anomaly detection unit* deployed at each router.

When the number of power-gated segments is excessive, two types of anomalies may arise: i) a large fraction of packets suffering long detours to their destinations, and ii) congestion due to the increased load on active links. Thus, we detect these anomalies to provide feedback to our threshold update algorithm. In addition, these anomalies are detected locally at each router and broadcasted globally using single-bit wires. Each ON/OFF decision engine is equipped with logic to update the threshold value based on this feedback information. Note that the global anomaly broadcast ensures that A_{TH} values are kept consistent throughout the NoC. This aspect, in turn, guarantees that power-gating decisions are fair, tackling the least used segments in the NoC.

Panthre provides the ability to systematically trade-off performance for power savings by adjusting the criterion for the detection of these anomalies. With Panthre, stable and power-efficient configurations are typically attained 10-15 epochs (1 epoch is 10K cycles in our design) after a program phase change, which is quick considering that application phases are up to 10s of millions of cycles.

4.2.3 Reconfiguration Algorithm

A hallmark of Panthre is that all power-gating decisions can be made independently at each router, while deadlock-freedom and connectivity among all nodes is still guaranteed throughout execution. This allows for frequent reconfigurations, in the order of one reconfiguration event per tens of thousands of cycles. In addition, Panthre eliminates the need of any additional hardware to recover from pathological scenarios such as deadlock. This is a great advantage in terms of silicon cost (and power), and it also limits the impact of reconfiguration on performance. Panthre's reconfiguration is based on the up*/down* routing algorithm, which breaks deadlocks by forbidding certain through-router connections between non-local links (*turns*). Up*/down* routing works by organizing all network nodes on a spanning tree starting from a root node of choice. Each node receives a unique order based on its distance from the root, and equidistant nodes are ordered arbitrarily. Thereafter, all routes first leading away from the root node (down-traversal) and then towards it (up-traversal) are marked invalid. This ensures deadlock-freedom as all dependency cycles involve at least one 'down→up turn'(down-traversal followed by up-traversal). A breadth-first construction of the spanning tree rooted at node 0 is shown in Figure 4.4a. As an

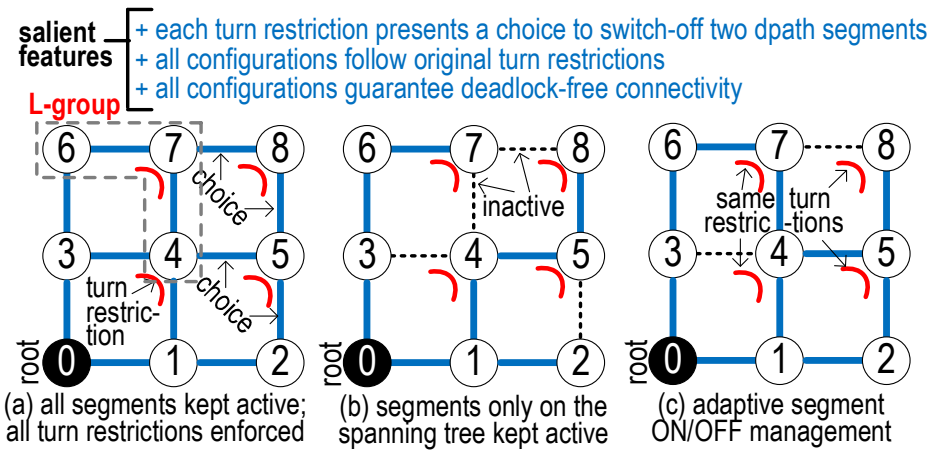


Figure 4.4 Panthre reconfiguration algorithm allows power-gating decisions to be made independently at each router without causing disconnection or deadlock. a) Breadth-first construction of the up*/down* spanning tree and corresponding turn restrictions. Each turn restriction node (L-group) presents a power-gating choice between datapath-segments. b) A minimally-connected network configuration degenerates into a spanning tree. c) A dynamically-adapted NoC where low-usage datapath-segments are power-gated.

example, 1→4 is a down-traversal, while 4→3 is an up-traversal. Therefore, turn 1→4→3 must be marked invalid.

Note that a spanning tree, by definition, connects all the nodes in the network and it is acyclic. In this context, Panthre's route-reconfiguration algorithm leverages the fact that turn restrictions are placed between two links if and only if one of them can be part of the spanning tree. As a result, it is possible to power-gate one of the two datapath segments connected to a disabled turn and still maintain full network connectivity. Figure 4.5 illustrates the property just outlined: the left portion of the figure shows a spanning tree construction, such that nodes 0,1 and 3 are already on the spanning tree rooted at node R, while node 2 is being added. It can be noted that either link 0-2 or link 3-2 are sufficient to connect to node 2. Since both are available, there will be a turn restriction 0-2-3. A similar situation is shown on the right side of the figure, where links 1-3 and 2-3 are sufficient to reach the to-be-added node 3, and the turn restriction is 1-2-3. The middle part of the figure shows a more general case, where both node 2 and 3 are being added to the spanning tree, using links 0-2 and 1-3, respectively. In this case, either the turn 0-2-3 or 1-3-2 must be disabled to break the cycle. Depending on the turn restriction placement, this situation degenerates into the one shown on the left or the right.

In order to organize Panthre's reconfiguration process, we call any two links connected by a disabled turn an *L-group*, as shown in Figure 4.4a. Therefore, a decision can be taken locally at each L-group, to power-gate one of the two bi-directional datapath links, while

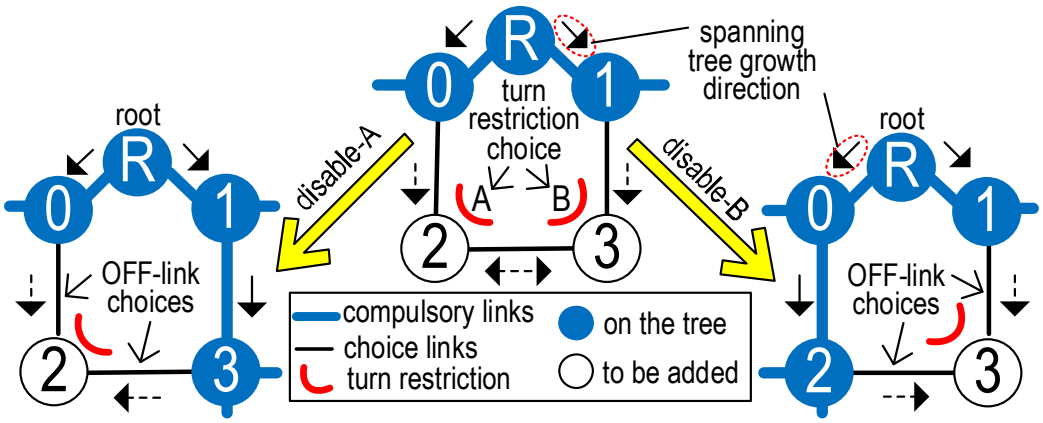


Figure 4.5 Each turn restriction provides an opportunity for power-gating one bidirectional link. Panthre leverages this property to put low-utilization links to sleep. Note how the property holds for any topology, and it is independent of how the up*/down* tree is grown.

the network would still be connected globally. Note that each bi-directional links comprises two opposite unidirectional datapath segments (see Section 4.2.1). In the extreme case when all L-groups decide to power-down two datapath segments each, the topology will degenerate into a spanning tree, as shown in Figure 4.4b. Panthre leverages a distributed and adaptive datapath-segment ON/OFF decision engine that determines the power-gating decisions locally at each L-group, according to application communication characteristics. An example network configuration produced using Panthre is shown in Figure 4.4c.

Even though many reconfiguration algorithms [4, 104] replace deadlock-free routing paths with another set of deadlock-free routing paths after reconfiguration, packets in-transit following the old routing paths can cause deadlocks by interacting with packets following the new routing paths. This is because paths valid in the old routing function might be disabled in the new routing function, or vice-versa. To circumvent this issue, Panthre ensures that any newly developed routing configuration complies with the turn-restrictions that were determined for an all-powered-ON NoC configuration (e.g., Figure 4.4a), eliminating the possibility of reconfiguration-induced deadlocks. Since Panthre only disables a link if it is a part of a turn-restriction (L-group), the corresponding turn would not be exercised, whether the corresponding link is enabled or disabled. Intuitively, if all L-groups maximally power-gate, the network topology will degenerate into a spanning tree (Figure 4.4b), and none of the restricted turns would be exercised. Note that Panthre's reconfiguration, though transparent and deadlock-free, may lead to reordering of packets between a source-destination pair. For systems where point-to-point ordering is essential, like for certain cache coherence protocols, we suggest the use of tag matching and reordering of packets at network interfaces, as is done in the Tilera architecture [130].

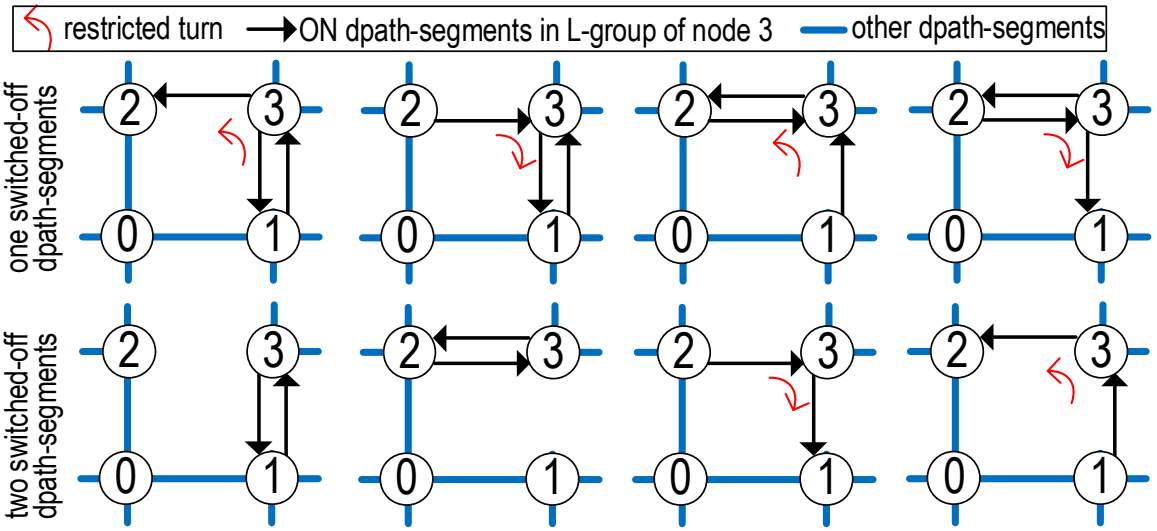


Figure 4.6 All possible power-gated configurations at each L-group. Configuration decisions can be made independently at each L-group without causing deadlock or disconnection. Topology degrades into a spanning tree when maximum possible (2) datapath segments are switched-off at each L-group.

Figure 4.6 shows the eight power-gated configurations that are possible at each L-group (around node 3 in the example). Any L-group in the network can have any of these eight configurations (or an all-on configuration), and the network would still be connected globally. In addition, Panthre guarantees that even though the L-groups transition locally from one configuration to the other, the network never enters a deadlock-situation. The nine configurations at each L-group give Panthre great flexibility in choosing the low-usage datapath segments to be power-gated.

Table 4.1 summarizes Panthre's leakage power saving potential when applied to different topologies. The table reports the total number of non-local datapath-segments in each topology and the number of datapath-segments required to construct the spanning tree (#span-seg). During periods of low activity, all the non-spanning datapath-segments could potentially be power-gated without sacrificing connectivity. We observe that Panthre has a great potential for reducing power consumption in popular topologies, such as mesh and tori, up to a 51% static power saving in a 8x8 torus.

topology	**#seg**	**#span-seg**	**%off**	**topology**	**#seg**	**#span-seg**	**%off**
mesh-4x4	48	30	38	**mesh-8x8**	224	126	44
torus-4x4	64	30	53	**torus-8x8**	256	126	51

Table 4.1 Panthre's leakage power saving potential for various topologies.

4.2.4 Implementation

In this section we discuss the detailed functionality and the hardware requirements of each of Panthre's four components as shown in Figure 4.2. We then overview the application-adaptive algorithm.

Activity counters and comparator (ACC). An ACC unit is associated with each datapath-segment that belongs to an L-group. The activity counters are 10-bit counters, incremented upon each flit traversing the corresponding datapath-segment. In addition, a 6-bit comparator (compares higher order bits only) is required to compare against the A_{TH} value provided by the ON/OFF decision engine to determine the power-gating status of the segment in the next epoch. An 8x8 mesh has 49 L-groups, with 4 datapath-segments each.

We monitor usage activity on an epoch-to-epoch basis. The smaller the epoch size, the more frequently Panthre initiates reconfiguration to quickly adapt to application characteristics. However, the lower-bound on epoch size is constrained by the latency of our: i) reconfiguration ($\sim$4K cycles), and ii) statistics collection (few thousand cycles for capturing patterns). We determined that an epoch size of 10K cycles provides a good trade-off between reconfiguration overhead and the amount of time Panthre takes to adapt to changing application characteristics. We further determined that power-gating a datapath segment that is used for more than 2^{10} cycles within an interval of 10K cycles, is always detrimental to performance. Therefore, our activity counters are only 10-bits wide, and all datapath-segments with more usage than that are considered vital and always kept active. In addition, the A_{TH} values are incremented or decremented in quanta of at least 2^4, and thus comparing the 6 higher order bits of the counters is sufficient.

The **ON/OFF decision engine** is deployed for each L-group and maintains and updates the A_{TH} value. It interfaces with the anomaly management unit to decide when the A_{TH} value should be incremented or decremented. To this end, a 6-bit adder/subtracter circuit is required at each L-group. In addition, the ON/OFF decision engine instructs the route computation unit to initiate a route-update once new power-gating decisions are made. For simplicity of implementation, decision engines associated with all L-groups operate in a synchronized manner. This is achieved by simply ensuring that the A_{TH} updates, the ON/OFF decisions and the route-updates are all applied only at epoch ends. After a datapath-segment has been power-gated, a few packets may still require to go through old routes to make forward progress. In this situation, the datapath-segment behaves as a conventional power-gating state machine: waking up on packet arrival and sleeping again upon its departure. Note that this scenario is extremely rare, and does not offset the benefits of Panthre: our experiments incorporate the delays and power costs due to these situations.

The **Panthre-enabled route computation unit** is shown in Figure 4.7. It consists of two sub-components: i) a logic-based distributed routing (LBDR) implementation [40] that provides routes corresponding to an all-segments-ON configuration, and ii) a routing table that stores the most up-to-date routes reflecting the power-gating status of the network. Having a backup LBDR implementation has three advantages: i) upon detection of an anomaly it allows the network to instantly switch to an all-segments-ON mode and limit performance impact, ii) it allows the routing table to be updated bit-by-bit in the background by providing a default path if no valid option yet exists in the table, and iii) it can be implemented cheaply. LBDR is a critical unit for Panthre as it allows uninterrupted operation even during reconfiguration. Note that all dynamic NoC route configurations follow the minimal set of turn restrictions, and hence packets are never stalled in router buffers due to unavailability of valid routing paths.

Panthre is based on up*/down* routing, a naïve implementation of which leads to congestion at a relatively low-load compared to the popular XY routing. Therefore, we choose less congested topology nodes as root and implement popular optimizations such as depth-first construction of the spanning tree and load-balanced path selection [105]. Therefore, we are able to extract at-par performance compared to XY routing, as is evident from the results in Section 4.4. Also note that Panthre disables all its functionalities at heavy loads because its leakage power savings are minimal at high loads. This keeps Panthre free of any additional congestion or power dissipation at high loads.

We leverage a distributed route-update algorithm inspired by Ariadne [4] to update the routing table on each reconfiguration event. Ariadne utilizes time-synchronized broadcasts from all destination routers in turn, communicated to all routers through simple forwarding operations. Each broadcast takes 64 cycles in an 8x8 mesh, and the entire reconfiguration process is completed in $\sim$4K cycles. Figure 4.7 summarizes the features of Ariadne's route-update algorithm. If Ariadne-style functionality is already available in fault-tolerant NoCs, it can be leveraged by Panthre with only minor modifications.

The **anomaly management unit** monitors two types of adverse behaviors that arise due to power-gated datapath segments: i) excessive detours, and ii) network congestion. For the purpose of detecting excessive detours, a special 'misroute'bit is reserved in the header flit of every packet. We set this bit if, at any router, a packet is routed through a port that takes it further away from the destination. For a mesh topology, this is as simple as calculating the relative X and Y coordinates of current and destination nodes: this information is already available in logic-based routing algorithms [40]. Each destination counts the fraction of packets that suffered a detour over those that did not. If the ratio is > 1, the destination node will broadcast a misrouting flag on a single-bit wired-OR ring (Figure 4.2). We deploy

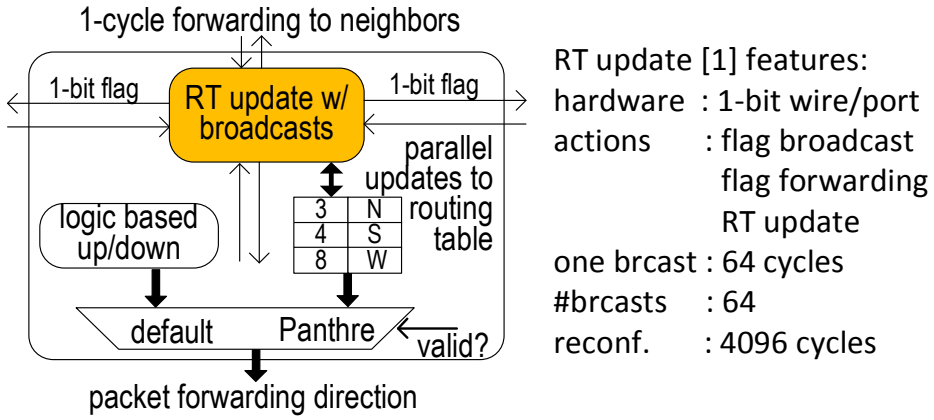


Figure 4.7 Panthre route computation unit consists of an up-to-date routing table reflecting the power-gating status of the NoC, and a default logic-based routing unit to provide backup routes if the routing table is unavailable due to an ongoing reconfiguration. It also incorporates a broadcast-based route-update scheme inspired by Ariadne.

one wired-OR ring for each of 4 8x2 regions in our 8x8 mesh. These 4 rings drive a separate wired-AND connection, and the root node is designated to monitor its anomaly status. If the wired-AND connection is set, the root node broadcasts an anomaly code on a 1-bit global wire that all routers can snoop. In other words, our detour detection scheme raises a flag if at least one node in each of the 4 regions observes more than 50% misroutes. We use a similar, but simpler scheme for congestion detection, inspired by the maximum buffer occupancy metric of [31]. If the total buffer occupancy, at any time and at any router in the NoC, is more than a certain threshold (29 in our implementation), the router noting the congestion broadcasts the anomaly code on the same 1-bit global wire used for reporting excessive detours.

Panthre's application-adaptive algorithm is shown in Figure 4.8. At the start of execution, A_{TH} is initialized to its maximum value, A_{THmax} (= 800 in our setup). All datapath segments with utilization above this value are never considered for power-gating. Among the others, the ones with activity below A_{TH} are switched-off. At the end of the execution epoch, the application-adaptive algorithm takes different actions based on whether or not an anomaly is flagged. If an anomaly is flagged (right part of the figure), suggesting that too many links are powered-off, then all components are instantly powered back on. This anomaly indicates that the current A_{TH} value is causing too aggressive power-gating. Therefore, if anomalies are detected in the last L consecutive epochs (L=3 in our design), A_{TH} is lowered. Power-gating decisions are then reassessed in light of the new A_{TH} value. Note that decreasing A_{TH} reduces the amount of switched-off segments and, in turn, decreases the likelihood of anomalies in the near future. In addition, the threshold values

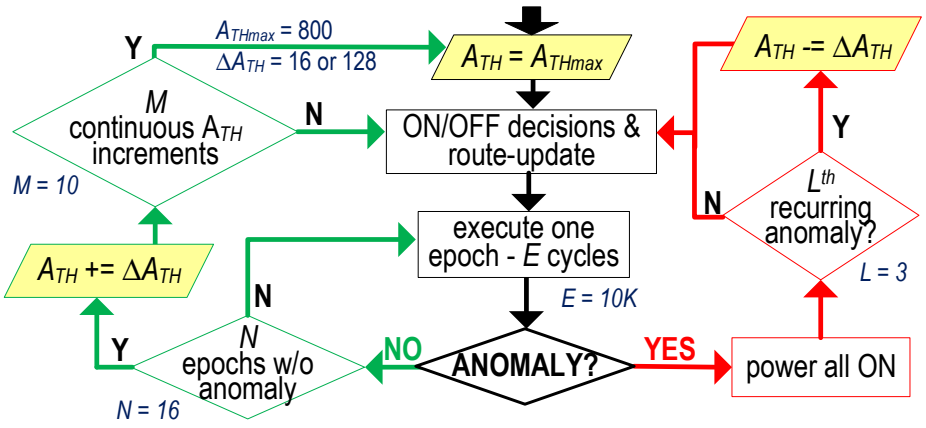


Figure 4.8 Flow-chart of Panthre's usage-threshold update algorithm.

are lowered in two phases, with first a coarse-grain (ΔA_{TH} = 128 in our setup), and then a fine-grained tuning (ΔA_{TH} = 16).

If an entire epoch is executed without any anomaly being flagged (left portion of the figure), indicating that our current configuration is performance-friendly, the next epoch is executed without updating power-gating decisions. However, if no violations are observed in the last N consecutive epochs (N=16 in our design), suggesting that our power-gating selection is too conservative and there is room for greater power savings, A_{TH} is increased and ON/OFF decisions are redone. After M (=10) successive A_{TH} increments, we determine that the application load has considerably increased and the current A_{TH} value is far from optimal, thus we update A_{TH} to A_{THmax}, and re-execute the algorithm from the start. The state machine for our algorithm is very simple and can be implemented in hardware at low cost. It is replicated in each ON/OFF decision engine at every L-group, requiring a 6-bit adder/subtracter for A_{TH} updates, and small counters and comparators for the other parameters (L,M,N).

Implementation overhead. The unidirectional ring wire for anomaly broadcasts, combined with wired-OR and wired-AND wires for detection of excess detours, leads to a wiring area overhead of only 0.39% [120] in comparison to the channels of our baseline router. We also assume that Ariadne-style route-updating functionality is already available for fault-tolerance. All other Panthre components are extremely lightweight, with small counters, comparators and adders added to each L-group. Therefore, compared to the deep buffers and many virtual channels of modern routers, Panthre-specific logic is trivial both in terms of power and area. Finally, the additional hardware is primarily for monitoring, and thus does not add to the delay of the critical timing paths.

4.3 Complete Router Shutdown

If some cores in a CMP system are idle, neither sending nor receiving traffic, the routers corresponding to these cores could be completely power-gated as long as doing so does not isolate any active node. A complete router shutdown is equivalent to shutting down 8 datapath segments (4 incoming and 4 outgoing), and hence it is a very lucrative power saving option. Panthre's deadlock-free and connectivity-preserving reconfiguration algorithm can be easily extended to shut down entire routers and still provide all its valuable properties.

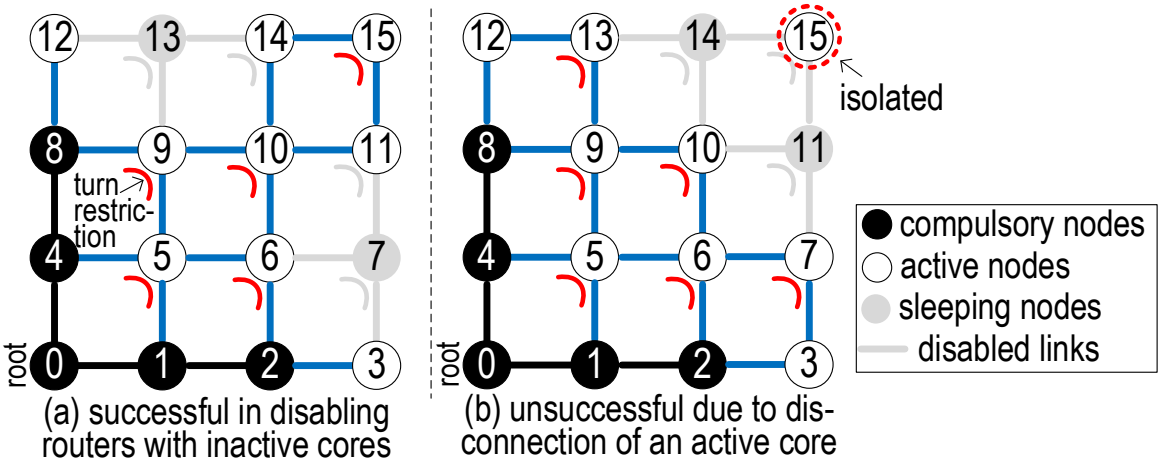


Figure 4.9 Complete router shutdown scheme within Panthre. Routers associated with communication-idle cores can be put to sleep while maintaining system connectivity, as long as they correspond to leaf nodes in the routing's spanning tree(s). Routers that can never be considered for shutdown are called 'compulsory'. a) Successful shutdown of routers 7 and 13. b) Routers 11 and 14 cannot be simultaneously shutdown because this would lead to isolating an active node, *i.e.*, node 15.

A router can be considered a candidate for shutdown only if it is a leaf node in at least one of Panthre's spanning tree constructions. The intuition behind this observation is that a leaf node is connected to the rest of the tree only via a single link. In addition, that link is only used for transferring packets originating or destined for that leaf router. Therefore, if the leaf node is communication-idle, the corresponding link can be switched-off without affecting the remaining topology. To provide an example, in Figure 4.9, the 'compulsory'routers that can never be completely shut down are shaded in black. In the setup of the Figure, the root is node 0 and the spanning tree was generated in a breadth-first fashion. Note that the compulsory routers are set once a root node and a turn-restriction configuration have been selected. Note also that in an 8x8 mesh, only 13 out of 64 routers are compulsory. We equipped Panthre to distinguish between compulsory nodes and those that can be shut down, and to apply shutdowns whenever possible for communication-idle

nodes. If all compulsory routers are kept active, and if connectivity is at all possible after shutting down all routers associated with sleeping cores, Panthre too is successfully able to provide connectivity and deadlock-freedom.

For applications running on a multicore with 25% idle cores, our entire router shutdown scheme can be applied successfully to more than 48% of the configurations. The integration of this scheme with Panthre is also simple: upon a group of cores notifying their idle state, the routers associated with them (if they are not compulsory for connectivity) are put to sleep. A route-update procedure is then executed and, if all active cores are still connected to the root core (this can be easily detected by analyzing the routing table), Panthre continues in router shutdown mode. If however, connectivity is lost, Panthre defaults back to its baseline routing, which power-gates at the granularity of a datapath segment.

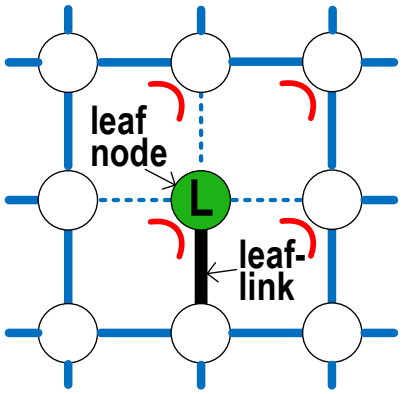


Figure 4.10 Leaf node and a leaf link. Figure shows a valid configuration with Panthre, where all-but-one links around the leaf-node are power-gated. In this scenario, the leaf-link only carries traffic originating or terminating at the leaf-node. Further, the leaf-link is not involved in any dependency cycle. Therefore, switching-off the leaf-link, and hence switching-off the entire leaf-router, will not affect connectivity or deadlock-freedom among the remaining nodes.

4.3.1 Connectivity with Complete Router Shutdown

Complete router shutdown can be easily implemented with Panthre and provides excellent leakage power-saving opportunities. In this section, we provide insight into the properties of Panthre's reconfiguration algorithm that make complete router shutdown possible without causing deadlock or disconnection. Complete router shutdown can be used to isolate a router if that router is at the leaf node of at-least one spanning tree instance realizable with Panthre. Panthre guarantees connectivity and deadlock-freedom for all network configurations that are generated by obeying the principles outlined in Section 4.2.3. In a particular configuration, if a router is at the leaf of the spanning tree, then by definition, it is connected

to the rest of the network via just one bidirectional link. Let us call this bidirectional link, a leaf link. The concept of a leaf-node and a leaf-link is shown in Figure 4.10. In such a configuration, the leaf link is only used for communication originating or terminating at the leaf router. In other words, the leaf link is not used for providing connectivity between any node pair that does not contain the corresponding leaf node. Additionally, the leaf link is not part of any cyclic dependencies. Therefore, switching-off the leaf link would not affect the connectivity or deadlock status of the rest of the network. Since the configuration before the link was switched-off, was deadlock-free and connected, so the configuration after switching-off the leaf link will also be connected and deadlock-free. Notice that powering-down the leaf link is equivalent to powering-down the entire leaf router in such a scenario.

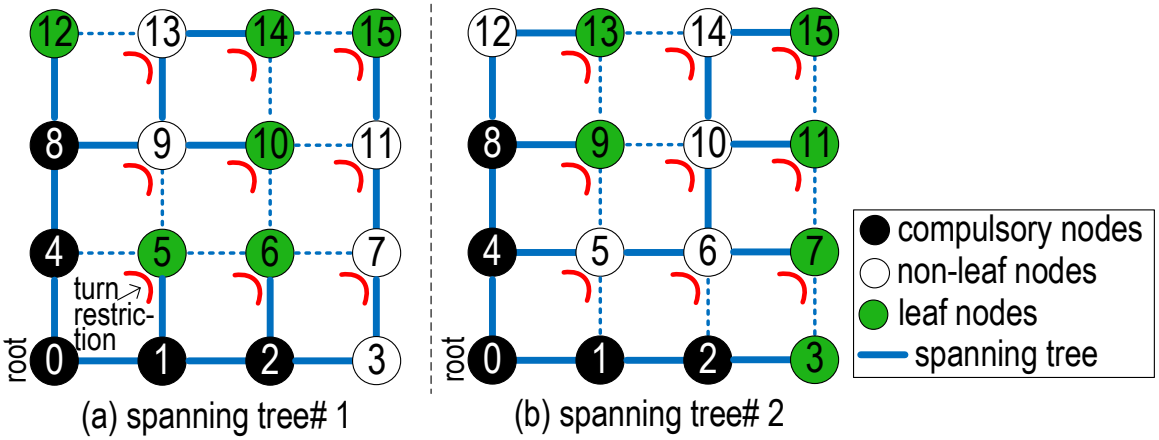


Figure 4.11 Complete router shutdown for a 4x4 mesh. Every router that is a leaf node in at least one of Panthre's spanning tree constructions, is candidate for shutdown. The candidates for shutdown are fixed for a particular root and turn-restriction configuration. Figure shows two spanning tree instances realizable with Panthre, showing all shutdown candidate routers at the leaf of the (at least one) spanning tree. The nodes that are not at the leaf of any spanning tree are marked as 'compulsory'.

Figure 4.11 shows two spanning tree instances, which together cover all the possible router candidates for shutdown in the 4x4 mesh example. In other words, the two spanning tree instances combined show all the possible leaf routers. Routers at node 0, node 1, node 2, node 4 and node 8, cannot be at the leaf node of any spanning tree that is rooted at node 0 and that follows the turn-restrictions shown in the figure. We denote such nodes as 'compulsory', as it is necessary to keep them active if deadlock-freedom and connectivity is to be maintained. In addition, the set of compulsory nodes is fixed for a particular choice of root node and turn-restrictions. The location of the compulsory nodes can be hardwired to complete router shutdown controllers at design-time.

4.4 Experimental Results

We evaluated Panthre on a cycle-level trace-driven multi-core simulator [30], modeling a 64-core CMP system as described in Table 4.2. We used a front-end functional simulator based on Pin [95] to collect instruction traces from applications, which are then transfered to the trace-driven cycle-level simulator. We also integrated a detailed on-chip network model, simulating a state-of-the-art two-stage router, described in Table 4.2b. In addition to the baseline design with no power-gating and the Panthre design, we also implemented router-level conventional power-gating with lookahead wakeup (*PG_conv*) [80] and a fine-grained port-level power-gating scheme (*PG_fg*) [81] for comparison. For both PG_conv and PG_fg, we used an idle-detection time of 4 cycles of inactivity [80]. The Panthre design parameters described in Section 4.2.4 (e.g., A_{THmax}) are calibrated after detailed design space exploration to provide a suitable trade-off between performance and power: the values we used in our evaluation are reported in Figure 4.8.

(a) Processor @2GHz

Cores	2-wide fetch/commit 64-entry ROB
coherence	4-hop MESI, 64B block
L1 cache	Private: 32KB/node ways:4 latency:2
L2 cache	Shared: 256KB/node ways:16 latency:6
Memory	Distributed: 1GB/bank banks:4 latency:160

(b) Network @2GHz

Topology	8x8 mesh, 128 bit links
Pipeline	2-stage VC flow ctrl
VCs	4 VCs/port, 8 flits/VC
Routing	XY for baseline up*/down* for Panthre
Workload	synthetic: uniform multi-prog: SPEC CPU06
Simulation (cycles)	synthetic: 5M multi-prog: 10M

Table 4.2 Experimental CMP: configuration of processor and network.

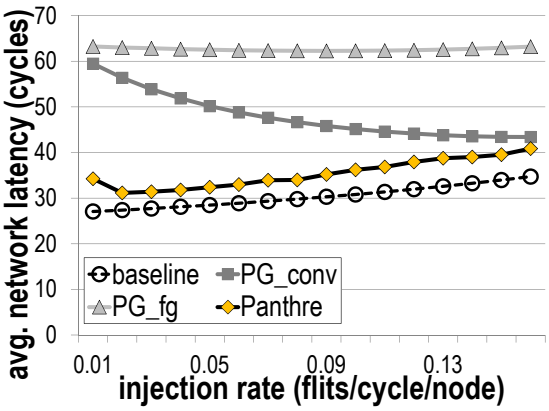
We analyzed our framework with two types of workloads: synthetic uniform random traffic, as well as 35 applications from the SPEC CPU2006 and commercial ($sap, tpcw, sjbb, sjas$) benchmark suites. We experimented across 40 randomly generated multi-programmed workload mixes, with each mix containing 10 copies each of 6 applications randomly picked from our suite of 35 applications. For brevity of results, we categorize the 40 workload mixes into four categories of 10 workloads each based on the amount of cache misses per kilo instructions (MPKI). Most network transactions originate because of misses in the caches, and hence MPKI correlates well with application communication load. The *light* category has benchmarks within MPKI of 200, while *light-med* spans the MPKI range 200-500. The *med* group includes relatively network heavy benchmarks, with MPKI between 500-1500, while the *heavy* category covers the MPKI range 1500-2500.

A considerable amount of energy is spent in putting components to sleep and bringing them back up. The amount of sleeping time required to compensate for this energy loss is called the 'breakeven time'. In our evaluation, we assume a breakeven time of 10 cycles and a wakeup delay of 4ns in agreement with previous research [24, 80]. The effective sleeping time after accounting for breakeven energy is called compensated sleep cycles (CSC) [80]. The CSC over total execution is a direct measure of leakage power savings. In our results, we report CSC values, in addition to latency increase and application slowdown. Finally, we use DSENT [120] to estimate total network power, accounting for both dynamic and static power at the 22nm technology node. For dynamic energy, DSENT is used to report energy spent per event (for e.g., buffer write), which is then tracked accurately in our cycle-level simulator.

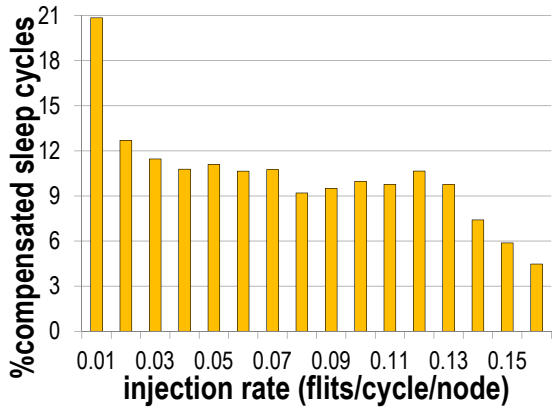
4.4.1 Synthetic Traffic

We first compare Panthre with other power-gating schemes using synthetic random traffic. Random traffic is the worst case scenario for Panthre as it distributes traffic uniformly across the network, reducing the number of low-usage links. Panthre's primary goal is to extract maximum power-savings by only turning-off low value datapath-segments, while keeping latency degradation in check. Figure 4.12a plots the average packet latency for each of the solutions evaluated. It is clear from the figure that PG_conv and PG_fg both lead to a high latency increase (>2x at low load). This is due to the fact that, at low load, network components observe packet traversals infrequently, and spend most of their time sleeping. Therefore, packets accumulate wakeup latency at each hop. At this injection rate, both the conventional power-gating schemes spend more than 75% of the time asleep. However, this level of latency degradation leads to unacceptable (>10%) application slowdown, as we will note in Section 4.4.2. Note that the latency and CSC for PG_conv, both decrease with increasing network load. However, we observe that at the injection rate where the latency degradation becomes acceptable for PG_conv (0.16 flits/cycle/node), the CSC value drops down to only 2.7%.

In contrast, Panthre's latency degradation is only 16.5% on average for the range of injection rates shown in the graph. At an injection rate beyond 0.16 flits/cycle/node, Panthre leads to negligible latency degradation as it keeps all links active at such high load. Naturally, the power-savings are also little at that point as almost all components are considered vital and not switched off. At low traffic loads, however, Panthre can lead to more than 20% leakage power savings (CSC) as can be noted from Figure 4.12b. With increasing load, CSC values decrease, but Panthre still saves 10.3% leakage power on average for the



(a) Average network latency. Conventional power-gating schemes suffer up to 2x increase in latency, and hence they are detrimental to performance. In contrast, Panthre keeps latency degradation under check.



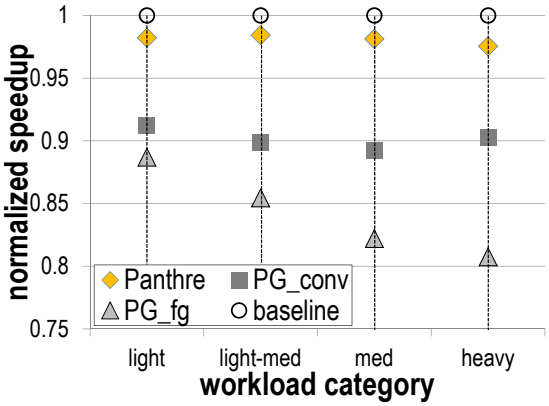
(b) Compensated sleep cycles (CSC) obtained with Panthre. Panthre's leakage power savings decrease with increasing load, the average ranges between 9.8% and 20.8% for injection rates varying from 0.01 to 0.16.

Figure 4.12 Panthre's network latency and CSC under uniform random traffic.

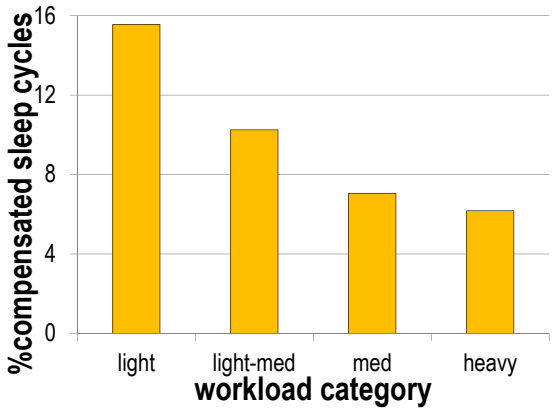
range shown in the graph.

4.4.2 Multiprogrammed Workloads

Panthre is designed to save leakage power only when it is possible without degrading performance. Figure 4.13a shows the speedup values for all power-gating schemes, normalized to the baseline CMP with no power-gating capability. The average slowdown with Panthre is only 1.9% across all four benchmark categories. In contrast, PG_conv and PG_fg lead to 9.8% and 15.7% slowdown averaged over all benchmark categories, respectively. Slowing down the application by such an amount is detrimental to system energy consumption. Therefore, these conventional power-gating schemes cannot be deployed in modern CMPs. In Figure 4.13b, we show that Panthre saves 15.6% leakage power for communication-light workloads on average, while 9.8% leakage power is saved on average across all benchmark categories. The total network power is reduced by 14.5%, 9.3%, 6.1% and 5.1% for light, light-med, med and heavy workloads, respectively. With substantial power savings at very little performance degradation, Panthre provides a good design trade-off for power-aware systems. In addition, if designers are willing to sacrifice more performance, the Panthre algorithm can be easily tuned for more aggressive power-gating.



(a) Normalized application speedup. Panthre limits the performance degradation to within 2% in almost all cases, while conventional power-gating typically causes more than 10% application slowdown on average.

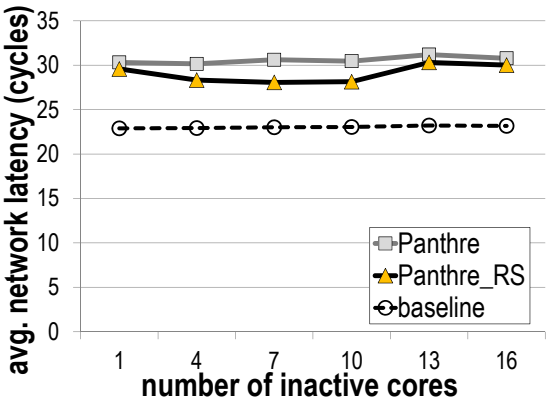


(b) Compensated sleep cycles (CSC) saved by Panthre. For light workloads, the average leakage power savings are 15.6%, while Panthre saves less for heavy workloads (6.7%).

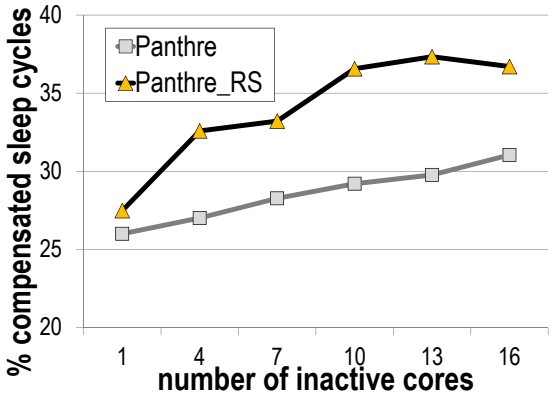
Figure 4.13 Panthre's network latency and CSC with multi-programmed workloads.

4.4.3 Complete Router Shutdown

As discussed in Section 4.3, Panthre can completely shutdown a significant number of routers associated with idle cores, and still ensure connectivity and deadlock-freedom among the active cores. This property is useful in scenarios where certain cores are communication-idle for substantial periods of time. We experimentally assessed Panthre's potential at successfully connecting all active cores after shutting down all routers associated with idle cores. To this end, we power-gated all routers associated with idle cores, varying the number of idle cores and selecting the pool randomly (among the non- compulsory ones). For each number of idle cores, we generated 1,000 distinct configurations and reconfigured using Panthre; we finally tested the NoC to determine if any active node was disconnected. Table 4.3 presents the results of this study for 6 different amounts of idle cores. It can be seen that with its simple scheme, Panthre can isolate up to 25% (16 routers) of the routers in most cases without causing any disconnection. The gains are reflected in the power-savings achieved by Panthre with complete-router-shutdown (*Panthre_RS*) under low injection rate (0.01 flits/cycle/node) and uniform traffic, as shown in Figure 4.14b. It can be noted that 36.9% of leakage power can be saved on average for 10-16 idle cores. Note that, if Panthre_RS is unsuccessful at maintaining connectivity for active cores after complete router shutdown, it reverts to its baseline approach of isolating only datapath segments. As shown in Figure 4.14a, Panthre_RS keeps latency increase under 30% on average, providing both better power saving and latency profile than our baseline Panthre solution.



(a) The **average network latency** increase in both Panthre and Panthre_RS is kept under check. Panthre_RS delivers a lower latency compared to basic Panthre.



(b) Percentage compensated sleep cycles (CSC) for Panthre_RS is 37% for 10-16 idle cores. Panthre_RS improves power savings further over Panthre.

Figure 4.14 Panthre's latency and CSC when complete router shutdown is enabled at a low injection rate and for uniform random traffic.

#idle-nodes	1	3	7	10	13	16
%connected after router shutdown	100	99	96	86	70	48

Table 4.3 Percentage of complete router shutdown configurations that provide connectivity among active nodes. More than 48% configurations are connected with 16 or less shutdown idle cores.

4.4.4 Adapting to Application Phase

In this section we present a case study to show Panthre's adaptivity to changing applications' communication characteristics. In this experiment, the network is operated at low load (injection @ 0.01 flits/cycle/node) for 1,000 epochs, and then the network load is suddenly increased to 0.10 flits/cycle/node. After 1,000 epochs at high load, the network traffic is again decreased to low load. This case-study is synonymous to typical application behavior, where major phase changes are rare and in spans of millions of cycles (1,000 epochs = 10M cycles). In addition, this case study tests Panthre's adaptivity to both increasing and decreasing network load. Figure 4.15 plots the updates to the A_{TH} values over the timeline of execution. At the start of execution, A_{TH} is initialized to A_{THmax}. Since the network is observing little traffic, activity of almost all datapath segments is below the A_{THmax} value, and Panthre's ON/OFF decision engines switch-off datapath-segments maximally at each L-group. Consequently, during the first epoch execution, Panthre's anomaly management units broadcast a flag indicating excess detours due to the many powered-down links. On reception of this flag at any L-group, all its datapath-segments will be instantly powered-up. At the same time, the A_{TH} is decreased by ΔA_{TH}. Note that, Panthre first goes through coarse-grain adaptation (ΔA_{TH} = 128) and quickly settles around the stable value at around

15 epochs. Thereafter, the A_{TH} value stays around the stable value, with periodic and fine-grained adjustments using ΔA_{TH} equal to 16.

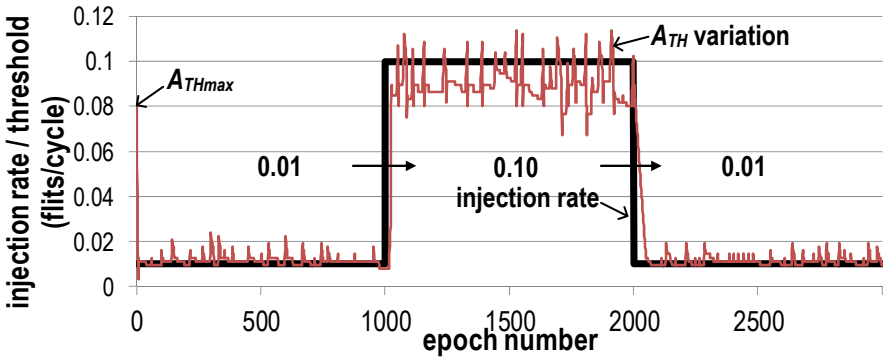


Figure 4.15 Panthre quickly adapts to low-to-high and high-to-low load transitions in applications' traffic.

Just before the transition from low-to-high load, the A_{TH} value is stable at a low value, and hence none of the links are powered-down when traffic suddenly increases. Thus, Panthre operates with very few powered-down datapath segments for the next few epochs, without observing an anomaly. The A_{TH} value starts incrementing in small steps after N (=16, Figure 4.8) such epochs, in absence of an anomaly. Eventually, the A_{TH} value is increased (after M = 10 such epochs of small A_{TH} increments, Figure 4.8) to A_{THmax}. Thereafter, the A_{TH} value stays around this stable mark for the duration between 1,000-2,000 epochs. Finally, during the high-to-low traffic load transition at 2,000th epoch, the A_{TH} value again reduces to a stable value, in a pattern similar to the one observed at the start of execution. This case study shows that Panthre can effectively handle both low-to-high and high-to-low load transitions in an application phase. Similarly, Panthre was able to adapt to both low-to-medium (0.01-0.05 flits/cycle/node) and medium-to-low (0.05-0.01 flits/cycle/node) load transitions in application phase. As can be seen from Figure 4.16, the value of A_{TH} stays around the optimal mark during medium traffic load, with only a few exceptions when A_{TH} increases to A_{THmax}. However, Panthre's anomaly detection units quickly detect excess power-gating in such cases and reduce the value of A_{TH}.

Figure 4.17 shows the adaptation of A_{TH} when the network is stimulated with uniform random traffic at a stable injection rate of 0.01 flits/cycle/node. The figure also shows the number of datapath segments that are put to sleep in each epoch. As can be seen from the figure, both A_{TH} and number of power-gated datapath segments start at high values. Therefore, Panthre detects excess detours and reduces the A_{TH} value. A stable A_{TH} value is reached within 10-15 epochs, leading to 47 power-gated datapath segments on average. In addition, Panthre quickly adjusts the number of power-gated datapath segments, if

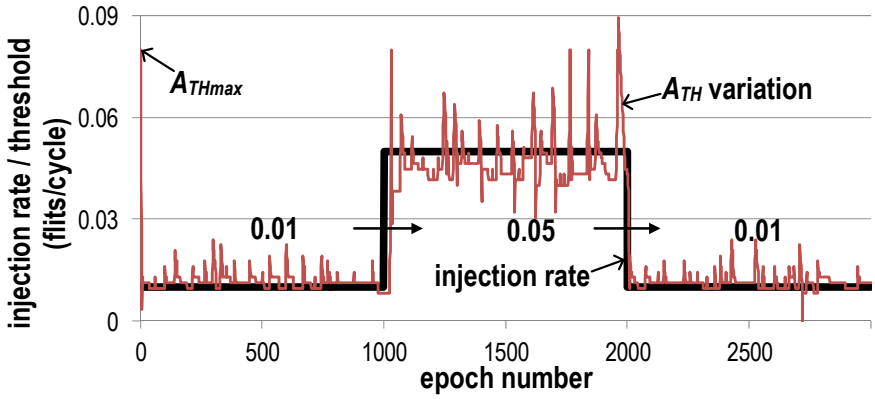


Figure 4.16 Panthre quickly adapts to low-to-medium and medium-to-low load transitions in applications' traffic.

power-gating decisions become too aggressive or too conservative.

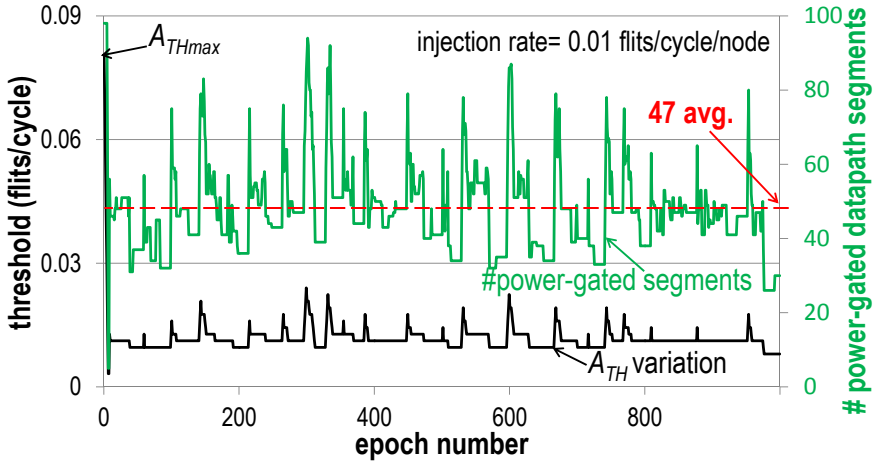


Figure 4.17 Variation in A_{TH} and the number of power-gated datapath segments across 1000 epochs of low-load uniform random traffic. Panthre updates A_{TH} to maintain a suitable level of power-gating.

4.5 Related Work

A number of power-gating schemes have been been proposed for NoC components, operating at different levels of granularity: routers [80], ports [81], VCs [82], buffers [66]. However, all such schemes suffer from accumulated wakeup times and excess energy spent in each power-gating event. NoRD [24] elongates periods of router sleep by steering light, sleep-interrupting traffic to a low-power ring network that is always kept active. However, NoRD requires additional VC resources both in the ring and primary network

to break deadlocks that may arise due to route-reconfiguration. In addition, NoRD requires substantial dedicated hardware design and verification effort. Router Parking [104] proposes to completely switch off routers associated with idle cores by leveraging route-reconfiguration at a central node. Note that this schemes suffers from all the drawbacks of centralized software-based reconfiguration algorithms. Additionally, Router Parking provides no power benefits when all cores are active, while Panthre saves up to 15% network power even in a fully-active CMP. Catnap [31] proposes the use of multiple lightweight networks in CMPs, while applying power-gating at the network granularity to keep the performance overhead low. However, Catnap is only applicable to CMPs with high-bandwidth requirements, and unlike Panthre, it cannot be utilized for fine-grained power-gating. Certain other route-reconfiguration solutions target performance improvement by dedicating more resources to contentious paths. For example, the Abacus turn model [45] leverages local and deadlock-free reconfigurations to divert additional routing paths towards heavy traffic. However, it cannot isolate components and save leakage power by power-gating.

Finally, route-reconfiguration has been well studied in the literature in the context of fault-tolerant routing. However, faults are rare occurrences; therefore, the majority of the algorithms designed for fault tolerance are centralized and take significant time and hardware resources. A notable exception is Ariadne [4], which describes a very quick (4K cycles for a 64 node network) and lightweight route reconfiguration scheme ($<$2% area overhead). Ariadne, however, also requires suspending the NoC operation during reconfiguration and can lead to reconfiguration-induced deadlocks. In our context, since we frequently reconfigure the NoC routing, network suspension would be detrimental. In contrast, our route reconfiguration algorithm, although inspired by Ariadne, provides deadlock-freedom throughout the reconfiguration, with no interruption of the mainstream NoC activity.

4.6 Summary

In this chapter, we targeted NoCs' runtime health from the perspective of runtime power dissipation. NoCs consume a significant portion of the on-chip power budget: during periods of heavy activity NoC can consume up to 30% of the on-chip power. The leakage phenomenon, that causes power dissipation irrespective of operational activity, contributes a majority share to the NoC power at 22nm, and the ratio is worsening further with shrinking transistor dimensions. We expect leakage power to be the limiting phenomenon for the coming silicon generations. We identified that power-gating, though effective in drastically

reducing leakage power when in effect, cannot be naïvely applied to shared resources, such as the NoC. Packets communicated over the network seldom allow NoC components to have long periods of sleep. In addition, packets that encounter sleeping components on their paths, accumulate latencies required to wake-up the sleeping components.

To this end, we propose Panthre, that maximizes leakage power savings by guaranteeing long periods of uninterrupted power-gating for NoC components. It leverages topology and routing reconfiguration to steer traffic away from sleeping components so to minimize the latency impact. Panthre's application-adaptive reconfiguration algorithm is implemented in a lightweight and distributed manner, and it guarantees a globally-connected and deadlock-free network at all times. In addition, Panthre monitors events indicating network performance degradation, and updates its power-gating decisions to provide a more suitable power-performance trade-off. Our experiments with light multi-programmed workloads show that Panthre reduces total network power by 14.5% on average, with only a 1.8% degradation in performance. Panthre's network power savings increases to as much as 36.9% on average, if 10-16 nodes are idle in a 64-node CMP.

Chapter 5
Conclusion

This dissertation addresses the most pressing challenges in ensuring the runtime health of networks-on-chip, which are becoming more and more the interconnect of choice in multi-core chips and SoCs. Communication infrastructure's health is critical to the operation of concurrent digital systems, where multiple computational units collaborate to complete a same task. In these systems, a malfunctioning interconnect may constitute a single point of failure if it disconnects one or more compute units from the rest of the system. Unfortunately, the complexity and size of these NoCs render them prone to design failures. In addition, its constituent components are increasingly susceptible to transistor failures as silicon dimensions shrink. Finally, contemporary NoCs bind together multitude of heterogeneous components, often forcing the common-case NoC designs to deliver sub-optimal power-efficiency at runtime, which, in turn, might cause excessive power dissipation.

The solutions developed in this thesis mitigate threats to NoCs' health by addressing these problems at runtime, *i.e.*, in a reactive manner. Consequently, they can address a wide range of unforeseen problems. Further, we take an integrated approach to ensure NoCs' runtime health, proposing a unified error detection and recovery framework. To this end, we divide the execution of the NoC into small time windows (or epochs) so as to provide quick error detection and recovery. During each epoch, our solutions monitor the execution activity of NoCs in a localized and distributed manner using lightweight checkers. Upon detection of critical events, we update the configuration of the routing scheme and the network topology to better adapt to the runtime environment. The end result of our work is a reliable and adaptable NoC infrastructure that is resilient to failures both in design and silicon, and that adapts its power dissipation at runtime to the needs of the heterogeneous designs and workloads.

5.1 Summary of the Contributions

This thesis develops design methodologies to enhance the runtime health of NoCs of the future. It particularly focuses on: i) guaranteeing correctness under unmanageable design complexity, ii) providing dependable communication with an unreliable silicon substrate, and iii) designing a power-aware and adaptable interconnect that tackles excessive power dissipation. To this end, we developed novel monitoring and reconfiguration techniques that exploit the unique characteristics of each failure class to provide low-cost, yet effective, solutions.

Runtime protection against design errors. Our first contribution is runtime protection against design errors that escape the pre-runtime verification effort. We presented SafeNoC, a runtime end-to-end error detection and recovery technique to guarantee the functional correctness of CMP interconnects. SafeNoC can detect and recover from a broad range of functional design errors, while incurring a low performance impact only on bug manifestation. This dissertation then described ForEVeR that improves error coverage and reduces area overhead compared to SafeNoC. ForEVeR complements the use of formal methods and runtime verification to ensure complete functional correctness in NoCs.

Runtime protection against transistor failures. The second contribution of this thesis is the protection of NoCs against both soft- and permanent- faults. Protection against soft-errors requires detection capabilities, followed by a recovery procedure. To this end, we exploited ForEVeR's infrastructure as soft-errors share many manifestation characteristics with design errors. For permanent faults, however, diagnosis of the fault site is essential, in addition to fault detection. To counter permanent faults, we first proposed a fine-grained detection and diagnosis scheme that is highly accurate and runs in the background, incurring no performance overhead in absence of errors. Once the faulty component is determined, the reconfiguration procedure either replaces the malfunctioning component by a healthy counterpart, or it disables the faulty component permanently, while leveraging the redundancy in the system. We then propose a reconfiguration solution, uDIREC, which utilizes fine-grained diagnosis information to disable components frugally, while using the redundancy built into the NoC for graceful performance degradation with increasing faults. We finally proposed a high-availability fault-tolerant solution, BLINC, based on a novel quick-reconfiguration scheme that limits the effect of a fault manifestation to few NoC components. Consequently, BLINC is able to provide uninterrupted availability for mission-critical applications, even during fault manifestations.

Runtime solution to avoid excessive power dissipation. The final contribution of this

thesis consists of avoiding excessive power dissipation in NoCs by monitoring execution and reconfiguring accordingly at runtime. NoCs consume a significant portion of the on-chip power budget: during periods of heavy activity NoC can consume up to 30% of the on-chip power [103, 130]. Our solution focuses on leakage power savings by power-gating idle components. However, the typical network traffic seldom allows NoC components to have long periods of sleep. In addition, packets that encounter sleeping components on their paths, accumulate latencies required to wake them up. To this end, we proposed Panthre, that maximizes leakage power savings by guaranteeing long periods of uninterrupted power-gating for NoC components. It leverages topology and routing reconfiguration to steer traffic away from sleeping components so to minimize the latency impact. In addition, Panthre is application-adaptive, *i.e.*, power savings are more aggressive for communication-light workloads, while savings are less for communication-heavy workloads.

5.2 Future Directions

The distributed checking and reconfiguration framework developed in this thesis can be extended to protect against other threats to NoCs' runtime health. Among the threats that we have not addressed in this dissertation, we believe that security breaches are the most devastating; therefore, addressing them will be the most compelling future research direction. Our framework can also be leveraged for runtime optimization of more conventional computer architecture metrics, such as performance or energy.

Secure communication. NoC is a centralized resource that is shared by all programs running on the CMP or the SoC. Therefore, applications actively sharing the NoC can cause interference in each other's execution. A tainted application running on an unprotected NoC can thus extract information from critical applications sharing the same NoC. Conventional wisdom to design secure NoCs has been to over provision hardware resources to provide isolation at runtime [129]. Using the techniques developed in this thesis, we propose a different approach to avert security breaches on NoCs. Rather than prevention, we rely upon continuous traffic monitoring to detect anomalous behavior, and triggering a recovery solution to avoid/thwart security breaches. Such a solution will result in a low-cost solution that suffers from performance degradation only when a security attack is underway.

Runtime optimization for performance. With the growing integration of application-specific components in CMPs and SoCs, optimizing the on-chip interconnect for the various execution scenarios is becoming increasingly difficult. This trend, combined with short-

time-to-market targets, leads to an over-provisioned NoC that is designed for the worst case. The schemes developed in this thesis can accurately monitor and predict applications' communication characteristics, and optimize the NoC at runtime for better performance or energy-efficiency. We can also employ our quick and transparent routing and topology reconfiguration techniques to tailor the NoC hardware to application traffic at runtime.

Bibliography

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

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

[3] Advanced Micro Devices, Inc. *AMD Athlon(TM) 64 and AMD Opteron(TM) Processors Thermal Design Guide*, September 2003.

[4] K. Aisopos, A. DeOrio, Li-Shiuan Peh, and V. Bertacco. ARIADNE: Agnostic reconfiguration in a disconnected network environment. In *Proc. PACT*, 2011.

[5] Konstantinos Aisopos and Li-Shiuan Peh. A systematic methodology to develop resilient cache coherence protocols. In *Proc. MICRO*, 2011.

[6] M.A. Al Faruque, T. Ebi, and J. Henkel. Configurable links for runtime adaptive on-chip communication. In *Proc. DATE*, 2009.

[7] Armin Alaghi, Naghmeh Karimi, Mahshid Sedghi, and Zainalabedin Navabi. Online NoC switch fault detection and diagnosis using a high level fault model. *Proc. DFT*, 2007.

[8] M.A. Alam. A critical examination of the mechanics of dynamic NBTI for PMOSFETs. In *Proc. IEDM*, 2003.

[9] K. V. Anjan and Timothy Mark Pinkston. An efficient, fully adaptive deadlock recovery scheme: DISHA. In *Proc. ISCA*, 1995.

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

[11] James Balfour and William J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *Proc. ICS*, 2006.

[12] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proc. ICCAD*, 2005.

[13] S. Bell, et al. TILE64 - processor: A 64-core SoC with mesh interconnect. In *Proc. ISSCC*, 2008.

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. PACT*, October 2008.

[15] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *Micro, IEEE*, 25(6), 2005.

[16] 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.

[17] M. Boule, J.-S. Chenard, and Z. Zilic. Assertion checkers in verification, silicon debug and in-field diagnosis. In *Proc. ISQED*, 2007.

[18] Robert Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Proc. CAV*, 2010.

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

[20] Rafael Casado, Aurelio Bermudez, Francisco Quiles, Jose Sanchez, and Jose Duato. Performance evaluation of dynamic reconfiguration in high-speed local area networks. In *Proc. HPCA*, 2000.

[21] F. Chaix, D. Avresky, N-E Zergainoh, and M. Nicolaidis. A fault-tolerant deadlock-free adaptive routing for on chip interconnects. In *Proc. DATE*, 2011.

[22] S. Chalasani and R.V. Boppana. Communication in multicomputers with nonconvex faults. *IEEE Trans. Computers*, 46(5), 1997.

[23] S. Chatterjee, M. Kishinevsky, and U.Y. Ogras. xMAS: Quick formal modeling of communication fabrics to enable verification. *IEEE Design & Test*, 29(3), june 2012.

[24] Lizhong Chen and Timothy M. Pinkston. Nord: node-router decoupling for effective power-gating of on-chip routers. In *Proc. MICRO*, 2012.

[25] Ge-M. Chiu. The odd-even turn model for adaptive routing. *IEEE Trans. Parallel and Distributed Systems*, 11(7), 2000.

[26] K. Constantinides et. al. BulletProof: A defect-tolerant CMP switch architecture. In *Proc. HPCA*, 2006.

[27] E. Cota, F.L. Kastensmidt, M. Cassel, M. Herve, P. Almeida, P. Meirelles, A. Amory, and M. Lubaszewski. A high-fault-coverage approach for the test of data, control and handshake interconnects in mesh networks-on-chip. *IEEE Trans. Computers*, 57(9), 2008.

[28] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[29] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Computers*, C-36(5), 1987.

[30] R. Das, O. Mutlu, T. Moscibroda, and C.R. Das. Application-aware prioritization mechanisms for on-chip networks. In *Proc. MICRO*, 2009.

[31] Reetuparna Das, Satish Narayanasamy, Sudhir Satpathy, and Ronald G. Dreslinski. Catnap: energy proportional multiple network-on-chip. In *Proc. ISCA*, 2013.

[32] A. DeOrio, K. Aisopos, V. Bertacco, and Li-S. Peh. DRAIN: Distributed recovery architecture for inaccessible nodes in multi-core chips. In *Proc. DAC*, 2011.

[33] A. DeOrio, D. Fick, V. Bertacco, D. Sylvester, D. Blaauw, J. Hu, and G. Chen. A reliable routing architecture and algorithm for nocs. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 2012.

[34] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Proc. IRPS*, 2011.

[35] A. Dutta and N. A. Touba. Reliable network-on-chip using a low cost unequal error protection code. In *Proc. DFT*, 2007.

[36] Masoumeh Ebrahimi, Masoud Daneshtalab, Juha Plosila, and Farhad Mehdipour. MD: minimal path-based fault-tolerant routing in on-chip networks. In *Proc. ASP-DAC*, 2013.

[37] H. Esmaeilzadeh, E. Blem, R.S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. ISCA*, 2011.

[38] F. Fazzino, M. Palesi, and D. Patti. Noxim: Network-on-chip simulator. http://noxim.sourceforge.net.

[39] David Fick, Andrew DeOrio, Jin Hu, Valeria Bertacco, David Blaauw, and Dennis Sylvester. Vicis: A reliable network for unreliable silicon. In *Proc. DAC*, 2009.

[40] J. Flich and J. Duato. Logic-based distributed routing for NoCs. *Computer Architecture Letters*, 7(1), 2008.

[41] J. Flich, A. Mejia, P. Lopez, and J. Duato. Region-based routing: An efficient routing mechanism to tackle unreliable hardware in network on chips. In *Proc. NoCs*, 2007.

[42] J. Flich, T. Skeie, R. Juarez-Ramirez, O. Lysne, P. Lopez, A. Robles, J. Duato, M. Koibuchi, T. Rokicki, and J. Sancho. A survey and evaluation of topology-agnostic deterministic routing algorithms. *IEEE Trans. Parallel and Distributed Systems*, 23(3), 2012.

[43] Harry Foster, Lawrence Loh, Bahman Rabii, and Vigyan Singhal. Guidelines for creating a formal verification testplan. In *Proc. DVCon*, 2006.

[44] B. Fu, Y. Han, J. Ma, H. Li, and X. Li. An abacus turn model for time/space-efficient reconfigurable routing. In *Proc. ISCA*, 2011.

[45] Binzhang Fu, Yinhe Han, Jun Ma, Huawei Li, and Xiaowei Li. An abacus turn model for time/space-efficient reconfigurable routing. In *Proc. ISCA*, 2011.

[46] Y. Fukushima, M. Fukushi, and S. Horiguchi. Fault-tolerant routing algorithm for network on chip without virtual channels. In *Proc. DFT*, 2009.

[47] P. B. Ghate. Electromigration-induced failures in VLSI interconnects. In *Proc. Reliability Physics Symposium*, 1982.

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

[49] Christopher J. Glass and Lionel M. Ni. Fault-tolerant wormhole routing in meshes without virtual channels. *IEEE Trans. Parallel and Distributed Systems*, 7, 1996.

[50] C.J. Glass and L.M. Ni. The turn model for adaptive routing. In *Proc. ISCA*, 1992.

[51] M.E. Gomez, J. Duato, J. Flich, P. Lopez, A. Robles, N.A. Nordbotten, O. Lysne, and T. Skeie. An efficient fault-tolerant routing methodology for meshes and tori. *Comp. Arch. Letters*, 3(1), 2004.

[52] Cristian Grecu, Andre Ivanov, Resve Saleh, and Partha Pande. Testing network-on-chip communication fabrics. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 26(12), 2007.

[53] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The StageNet fabric for constructing resilient multicore systems. In *Proc. MICRO*, 2008.

[54] O. Hammami, Xinyu Li, and J.-M. Brault. NOCEVE: Network on chip emulation and verification environment. In *Proc. DATE*, 2012.

[55] Daniel Holcomb, Bryan Brady, and Sanjit A. Seshia. Abstraction-based performance analysis of NoCs. In *Proc. DAC*, 2011.

[56] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Proc. ISSCC*, 2010.

[57] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *Proc. ISLPED*, 2004.

[58] R. Hum. How to boost verification productivity. EE Times, 2005. `http://www.eetimes.com`.

[59] Reliability Wearout Mechanisms in Advanced CMOS Technologies. *Strong, A. W. and Wu, E. Y. and Vollertsen, R.-P. and Sune, J. and La Rosa, G. and Rauch, S.E. and Sullivan, T.D.* Wiley & Sons Inc., 2009.

[60] Intel. *Intel Core2 Duo and Intel Core2 Solo Processor for Intel Centrino Duo Processor Technology Specification Update*, September 2007.

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

[62] Intel Corporation. *Intel(R) Pentium(R) 4 Processor on 90 nm Process Thermal and Mechanical Design Guidelines*, February 2004.

[63] A.B. Kahng, Bin Li, Li-Shiuan Peh, and K. Samadi. Orion 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proc. DATE*, 2009.

[64] K. Kailas, V. Paruthi, and B. Monwai. Formal verification of correctness and performance of random priority-based arbiters. In *Proc. FMCAD*, 2009.

[65] J. Keane, S. Venkatraman, P. Butzen, and C.H. Kim. An array-based test circuit for fully automated gate dielectric breakdown characterization. In *Proc. CICC*, 2008.

[66] Gwangsun Kim, J. Kim, and Sungjoo Yoo. Flexibuffer: reducing leakage power in on-chip network routers. In *Proc. DAC*, 2011.

[67] John Kim and Hanjoon Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proc. NoCArc*, 2009.

[68] Jongman Kim, C. Nicopoulos, Dongkook Park, V. Narayanan, M.S. Yousif, and C.R. Das. A gracefully degrading and energy-efficient modular router architecture for on-chip networks. In *Proc. ISCA*, 2006.

[69] A. Kohler and M. Radetzki. Fault-tolerant architecture and deflection routing for degradable noc switches. In *Proc. NoCs*, 2009.

[70] Michihiro Koibuchi, Hiroki Matsutani, Hideharu Amano, and Timothy Mark Pinkston. A lightweight fault-tolerant mechanism for network-on-chip. In *Proc. NoCs*, 2008.

[71] D. Lee, R. Parikh, and V. Bertacco. Blinc: a brisk and limited-impact NoC reconfiguration algorithm. In *Proc. DATE*, 2014.

[72] Teijo Lehtonen, David Wolpert, Pasi Liljeberg, Juha Plosila, and Paul Ampadu. Self-adaptive system for addressing permanent errors in on-chip interconnects. In *IEEE Trans. (VLSI) Systems*, 2010.

[73] Manlap Li et al. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *Proc. ASPLOS*, 2008.

[74] Xiaola Lin, P.K. McKinley, and L.M. Ni. Deadlock-free multicast wormhole routing in 2-d mesh multicomputers. *IEEE Trans. Parallel and Distributed Systems*, 5(8), 1994.

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

[76] Olav Lysne, José Miguel Montañana, Jose Flich, José Duato, Timothy Mark Pinkston, and Tor Skeie. An efficient and deadlock-free network reconfiguration protocol. *IEEE Trans. Computers*, 57(6), 2008.

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

[78] Milo Martin, Daniel Sorin, Bradford Beckmann, Michael Marty, Min Xu, Alaa Alameldeen, Kevin Moore, Mark Hill, and David Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4), 2005.

[79] Juan Miguel Martínez, et al. Software-based deadlock recovery technique for true fully adaptive routing in wormhole networks. In *Proc. ICPP*, 1997.

[80] H. Matsutani, M. Koibuchi, H. Amano, and D. Wang. Run-time power gating of on-chip routers using look-ahead routing. In *Proc. ASPDAC*, 2008.

[81] H. Matsutani, M. Koibuchi, D. Ikebuchi, K. Usami, H. Nakamura, and H. Amano. Ultra fine-grained run-time power gating of on-chip routers for cmps. In *Proc. NoCs*, 2010.

[82] H. Matsutani, M. Koibuchi, D. Wang, and H. Amano. Adding slow-silent virtual channels for low-power on-chip networks. In *Proc. NoCs*, 2008.

[83] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proc. MICRO*, 2007.

[84] A. Mejia et. al. Segment-based routing: An efficient fault-tolerant routing algorithm for meshes and tori. In *Proc. IPDPS*, 2006.

[85] S. Murali, T. Theocharides, N. Vijaykrishnan, M.J. Irwin, L. Benini, and G. De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design & Test*, 22(5), 2005.

[86] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proc. EUROSYS*, 2011.

[87] George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu. Next generation on-chip networks: what kind of congestion control do we need? In *Proc. Hotnets*, 2010.

[88] M. Palesi, S. Kumar, and V. Catania. Leveraging partially faulty links usage for enhancing yield and performance in networks-on-chip. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 29(3), 2010.

[89] Bernd Panzer-Steindel. Data integrity. Technical Report CERN/IT. Draft 1.3, CERN, Apr 2007.

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

[91] R. Parikh and V. Bertacco. uDIREC: unified diagnosis and reconfiguration for frugal bypass of NoC faults. In *Proc. MICRO*, 2013.

[92] R. Parikh and V. Bertacco. ForEVeR: A complementary formal and runtime verification approach to correct NoC functionality. *ACM Trans. Embedded Computing Systems*, 13(3s), 2014.

[93] R. Parikh, R. Das, and V. Bertacco. Power-aware NoCs through routing and topology reconfiguration. In *Proc. DAC*, 2014.

[94] Dongkook Park, Chrysostomos Nicopoulos, Jongman Kim, N. Vijaykrishnan, and Chita R. Das. Exploring fault-tolerant network-on-chip architectures. In *Proc. DSN*, 2006.

[95] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proc. MICRO*, 2004.

[96] Andrea Pellegrini, Joseph L. Greathouse, and Valeria Bertacco. Viper: virtual pipelines for enhanced reliability. In *Proc. ISCA*, 2012.

[97] Timothy Mark Pinkston, Ruoming Pang, and José Duato. Deadlock-free dynamic reconfiguration schemes for increased network dependability. *IEEE Trans. Parallel and Distributed Systems*, 14(8), 2003.

[98] 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.

[99] M. Prvulovic, Zheng Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. ISCA*, 2002.

[100] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide. Immunet: A cheap and robust fault-tolerant packet routing mechanism. In *Proc. ISCA*, 2004.

[101] J. Raik, R. Ubar, and V. Govind. Test configurations for diagnosing faulty links in noc switches. In *Proc. ETS*, 2007.

[102] S. Rodrigo et al. Addressing manufacturing challenges with cost-efficient fault tolerant routing. In *Proc. NoCs*, 2010.

[103] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, and N. Borkar. A 2 tb/s 6x4 mesh network for a single-chip cloud computer with dvfs in 45 nm cmos. *IEEE Journal of Solid-State Circuits*, 46(4), 2011.

[104] A. Samih, Ren Wang, A. Krishna, C. Maciocco, C. Tai, and Y. Solihin. Energy-efficient interconnect via router parking. In *Proc. HPCA*, 2013.

[105] J.C. Sancho, A. Robles, and J. Duato. An effective methodology to improve the performance of the up*/down* routing algorithm. *IEEE Trans. Parallel and Distributed Systems*, 15(8), 2004.

[106] José Carlos Sancho, Antonio Robles, and José Duato. A flexible routing scheme for networks of workstations. In *Proc. ISHPC*, 2000.

[107] S.K. Sastry Hari, S.V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Application resiliency analyzer for transient faults. *Micro, IEEE*, 33(3), 2013.

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

[109] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the International Joint Conference on Measurement and Modeling of Computer Systems*, 2009.

[110] M.D. Schroeder et. al. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Trans. Selected Areas in Communication*, 9(8), 1991.

[111] Frank Sem-Jacobsen and Olav Lysne. Topology agnostic dynamic quick reconfiguration for large-scale interconnection networks. In *Proc. CCGrid*, 2012.

[112] S. Shamshiri, A. Ghofrani, and K.-T. Cheng. End-to-end error correction and online diagnosis for on-chip networks. In *Proc. ITC*, 2011.

[113] A. L. Shimpi. The source of intel's cougar point sata bug. AnandTech, 2011. `http://www.anandtech.com`.

[114] Jared Smolens, Brian Gold, James Hoe, Babak Falsafi, and Ken Mai. Detecting emerging wearout faults. In *Proc. SELSE*, 2007.

[115] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The impact of technology scaling on lifetime reliability. In *Proc. DSN*, 2004.

[116] David Starobinski, et al. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Trans. Networks*, 11(3), 2003.

[117] J. H. Stathis, B. P. Linder, R. Rodrguez, and S. Lombardo. Reliability of ultra-thin oxides in CMOS circuits. *Microelectronics Reliability*, 43(9-11), 2003.

[118] Alessandro Strano et al. OSR-Lite: fast and deadlock-free NoC reconfiguration framework. In *Proc. SAMOS*, 2012.

[119] Alessandro Strano et al. OSR-Lite: fast and deadlock-free NoC reconfiguration framework. In *Proc. SAMOS*, 2012.

[120] Chen Sun, C.-H.O. Chen, G. Kurian, Lan Wei, J. Miller, A. Agarwal, Li-Shiuan Peh, and V. Stojanovic. Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proc. NoCs*, 2012.

[121] Synopsys. Synopsys Design Compiler, 2006.

[122] Synopsys. Synopsys Magellan, 2009. http://www.synopsys.com.

[123] Francisco Trivino, Davide Bertozzi, and Jose Flich. A fast algorithm for runtime reconfiguration to maximize the lifetime of nanoscale NoCs. In *Proc. INA-OCMC*, 2013.

[124] G. Tsiligiannis and L. Pierre. A mixed verification strategy tailored for networks on chip. In *Proc. NoCs*, 2012.

[125] Freek Verbeek and Julien Schmaltz. Hunting deadlocks efficiently in microarchitectural models of communication fabrics. In *Proc. FMCAD*, 2011.

[126] Freek Verbeek and Julien Schmaltz. Easy formal specification and validation of unbounded networks-on-chips architectures. *ACM Trans. Design Automation of Electronic Systems*, 17(1), 2012.

[127] Eduardo Wachter, Augusto Erichsen, Alexandre Amory, and Fernando Moraes. Topology-agnostic fault-tolerant NoC routing method. In *Proc. DATE*, 2013.

[128] Ilya Wagner and Valeria Bertacco. Engineering trust with semantic guardians. In *Proc. DATE*, 2007.

[129] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proc. ISCA*, 2013.

[130] D. Wentzlaff, P. Griffin, H. Hoffmann, Liewei Bao, B. Edwards, C. Ramey, M. Mattina, Chyi-Chang Miao, J.F. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5), 2007.

[131] Young Jin Yoon, Nicola Concer, Michele Petracca, and Luca Carloni. Virtual channels vs. multiple physical networks: A comparative analysis. In *Proc. DAC*, 2010.

[132] Bardia Zandian et al. WearMon: reliability monitoring using adaptive critical path testing. In *Proc. DSN*, 2010.

[133] Zhen Zhang, Alain Greiner, and Sami Taktak. A reconfigurable routing algorithm for a fault-tolerant 2D-mesh network-on-chip. In *Proc. DAC*, 2008.