TAL TECH

DOCTORAL THESIS

Techniques for Robust Routing, Communication and Computation in Multiprocessor Systems

Karl Janson

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY
TALLINN 2021

TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
3/2021

Techniques for Robust Routing, Communication and Computation in Multiprocessor Systems

KARL JANSON

TAL TECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

The dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and Systems Engineering on 14th of December 2021

Supervisor: Prof. Thomas Hollstein, PhD,
Department of Computer Systems, School of Information Technologies,
Tallinn University of Technology
Tallinn, Estonia

Co-supervisor: Prof. Jaan Raik, PhD,
Department of Computer Systems, School of Information Technologies,
Tallinn University of Technology
Tallinn, Estonia

Opponents: Prof. Dr. Fernanda Lima Kastensmidt,
The Federal University of Rio Grande do Sul,
Porto Alegre, Brazil

Prof. Dr-Ing. Thilo Pionteck,
Otto-von-Guericke University Magdeburg,
Magdeburg, Germany

Defence of the thesis: 14th of January 2021, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for any academic degree elsewhere.

Karl Janson _____
signature

European Union
European Regional
Development Fund

Investing
in your future

ISSN 2585–6898 (publication)
ISBN 978-9949-83-653-6 (publication)
ISSN 2585–6901 (PDF)
ISBN 978-9949-83-654-3 (PDF)
Printed by Koopia Niini & Rauam

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
3/2021

Robustse marsruutimise, side ja arvutuse tehnikad mitmeprotsessorilistes süsteemides

KARL JANSON

TAL TECH
KIRJASTUS

Contents

List of Publications 7

Author's Contributions to the Publications 8

Abbreviations 9

1 Introduction 10
1.1 Motivation 11
1.1.1 Trends and Barriers in Scalable CMOS System Design 11
1.1.2 Vulnerabilities of Nano-Scale CMOS Systems 12
1.1.3 Reliability Challenges in Distributed Embedded Systems 13
1.2 Problem Formulation 13
1.3 Contributions 15
1.4 Thesis Organization 18

2 Background 19
2.1 Dependability 19
2.1.1 Dependability Impairments 19
2.1.2 Mechanisms Behind Radiation-Induced Transient Faults 20
2.1.3 Fault Tolerance 22
2.2 Networks-on-Chip 23
2.2.1 Turn Model-Based Routing 24
2.2.2 Switching Strategies 26
2.2.3 Example Packet Structure 27
2.2.4 Flow Control Mechanism 28
2.2.5 Bonfire Router Architecture 29

3 Relaxed Transmission Architecture 33
3.1 Introduction 33
3.2 Literature Review 33
3.3 The Effects of Soft Errors in Network-on-Chip Packets 36
3.4 Architecture of the Relaxed Transmission Mechanism 36
3.5 Optimized Relaxed Transmission Mechanism 38
3.6 Experimental Results 39
3.7 Chapter Conclusions 41

4 Comprehensive Performance and Robustness Analysis of 2D Turn Models for Network-on-Chips 42
4.1 Introduction 42
4.2 Literature Review 44
4.3 Evaluation of Turn Models for Deadlock Freeness 44
4.3.1 Routing Graph 45
4.3.2 Proof of Deadlock Freeness 46
4.4 Identification of Usable Turn Models 47
4.5 Metrics for Adaptivity 48
4.6 Average Connectivity Evaluation 49
4.7 Latency Evaluation 51
4.8 Implications on Permanent Fault Tolerance in NoCs 52
4.8.1 Overview of the Routing Algorithm Reconfiguration Process 52

4.8.2 Choosing the Best Turn Model for a Fault Situation 53
4.9 Chapter Conclusions 54

5 Software TMR with Distributed Voting 55
5.1 Introduction 55
5.2 Literature Review 57
5.3 System Requirements and Fault Model 59
5.4 STROBES Fault Handling Algorithm 61
5.4.1 Communication Between the Processing Elements 61
5.4.2 Algorithm Description 62
5.4.3 Discussion on Real-Time Implementation of STROBES 69
5.5 Fault Handling and Synchronization in STROBES 70
5.6 Reliability Assessment 73
5.6.1 Definition of Failure Condition 74
5.6.2 Markov Model Definition 75
5.6.3 Reliability Calculation 80
5.7 Simulator 83
5.8 Experimental Results 86
5.8.1 Type-SF faults 87
5.8.2 Type-TF faults 87
5.8.3 Type-NF faults 89
5.8.4 Additional Observations 91
5.9 Chapter Conclusions 92

6 Conclusions 94

List of Figures 99

List of Tables 100

References 101

Acknowledgements 111

Abstract 112

Appendix 1 117

Appendix 2 125

Appendix 3 131

Appendix 4 137

Appendix 5 145

Curriculum Vitae 153

Elulookirjeldus 154

List of Publications

The present Ph.D. thesis is based on the following publications that are referred to in the text by Roman numbers.

I K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "AWAIT: An ultra-lightweight soft-error mitigation mechanism for Network-on-Chip links," in *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–6, July 2018

II K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "Handling of SETs on NoC links by exploitation of inherent redundancy in circular input buffers," in *2018 16th Biennial Baltic Electronics Conference (BEC)*, pp. 1–4, Oct 2018

III S. P. Azad, B. Niazmand, K. Janson, T. Kogge, J. Raik, G. Jervan, and T. Hollstein, "Comprehensive performance and robustness analysis of 2D turn models for Network-on-Chips," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May 2017

IV S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in Network-on-Chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 48–53, April 2017

V K. Janson, C. J. Treudler, T. Hollstein, J. Raik, M. Jenihhin, and G. Fey, "Software-level TMR approach for on-board data processing in space applications," in *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 147–152, April 2018

Other related publications

VI J. Malburg, K. Janson, J. Raik, and F. Dannemann, "Fault-aware performance assessment approach for embedded networks," in *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 1–4, April 2019

Author's Contributions to the Publications

I The author developed the concept of relaxed transmission and specifically the AWAIT approach, which was introduced in this publication. The author also developed and ran the experiments and was the main contributor and to writing the paper. The author also presented the work at the conference.

II This publication is the extension of Publication I. The author developed the concept of the ReUSE mechanism introduced in the paper. The author also designed and ran the experiments, was the main contributor to the paper and presented the work at the conference.

III The author was involved in designing the concept and experiments used in the publication. The author ran the experiments for the work and also co-wrote the paper and presented the work at the conference.

IV The author was involved in engineering work for the Bonfire network-on-chip platform presented in the publication and also co-wrote different parts of the paper.

V The author developed the STROBES algorithm introduced in the paper and implemented it in a custom-made simulation tool for running the experiments. The author also designed and ran the experiments, was the main contributor to the paper and presented the work at the conference.

Author's contributions to other related publications

VI The author helped in implementation if the wrapper for Bonfire network-on-chip platform for the network simulation tool presented in the paper, wrote a background section in the paper about the Bonfire NoC platform and presented the work at the conference.

Abbreviations

2D	Two-Dimensional
3D	Three-Dimensional
BER	Bit Error Rate
BW	Band Width
CDG	Cyclic Dependency Graph
CMOS	Complementary Metal–Oxide–Semiconductor
COTS	Commercial-off-the-Shelf
CP	Checkpoint
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DoA	Degree of Adaptivity
E2E	End-to-End
ECC	Error Correction Code
EDC	Error Detecting Code
F-RET	Flit Retransmission
FIFO	First-In-First-Out
FIR	Fault Injection Rate
FSM	Finite State Machine
HBH	Hop-by-Hop
HCI	Hot Carrier Injection
hi-rel	Highly Reliable
IP	Intellectual Property
LBDR	Logic Based Distributed Routing
MET	Multiple Event Transient
MPSoC	Multi-Processor System-on-Chip
MTTF	Mean Time to Failure
NACK	Negative Acknowledgment
NBTI	Negative-Bias Temperature Instability
NoC	Network-on-Chip
OS	Operating System
P-RET	Packet Retransmission
PE	Processing Element
PIR	Packet Injection Rate
RG	Routing Graph
RT	Relaxed Transmission
RTL	Register Transfer Level
RTOS	Realtime Operating System
SaF	Store-and-Forward
SEE	Single Even Effect
SET	Single Event Transient
SEU	Single Event Upset
SHMU	System Health Monitoring Unit
SoC	System-on-Chip
TM	Turn Model
TMR	Triple-Modular Redundancy

1 Introduction

Computing technology has developed at an incredible pace over the past decades. The main contributor to this process is the ongoing reduction of transistor's feature size. As smaller transistors are introduced, an ever-larger number of components can be integrated on a single chip without increasing the physical area of that chip.

Modern distributed embedded systems, such as the ones used in automotive applications and satellites, are complex systems. No matter whether they are implemented in the form of multi-processor systems-on-chips (MPSoCs) or built from separate interconnected processor boards, distributed embedded systems consist of multiple processing elements, which are connected using some form of interconnection infrastructure. The entire system is then precisely orchestrated by software to achieve a functionality that supersedes the sum of its parts. The components are often designed and built independently and then integrated by a third party.

However, systems built using smaller technology nodes are also more susceptive to faults [7, 8]. The faults can be caused by many factors, such as aging [9–12], radiation [13–15], or manufacturing defects. Radiation has the biggest influence on satellites [16–19], since they spend the entire mission duration without any maintenance or repairs in the high radiation environment of space [20, 21]. However, faults caused by cosmic radiation are not only a problem in space. Even though most cosmic radiation particles are deflected by Earth's magnetic field, errors attributed to cosmic radiation are still a serious problem for airplane electronics [22] and even for some terrestrial devices, such as servers [23, 24] in data centers. In order to guarantee the proper and safe functioning of these critical systems, the presence of fault tolerance is not only necessary, but it is essential.

Faults can occur in many parts of the system. The approaches presented in this thesis provide fault detection and correction capabilities for different fault types, and for both, the processing elements and for the interconnects, which connect the processing systems together. The first two content chapters (Chapter 3 and Chapter 4) of the thesis concentrate on protecting the interconnects in MPSoCs. On the other hand, the software-based method for protecting the software that is running on the processing elements, explained in Chapter 5, is applicable for a wider variety of distributed embedded systems. While the approaches presented in this thesis can be used separately, they will complement each other when used together.

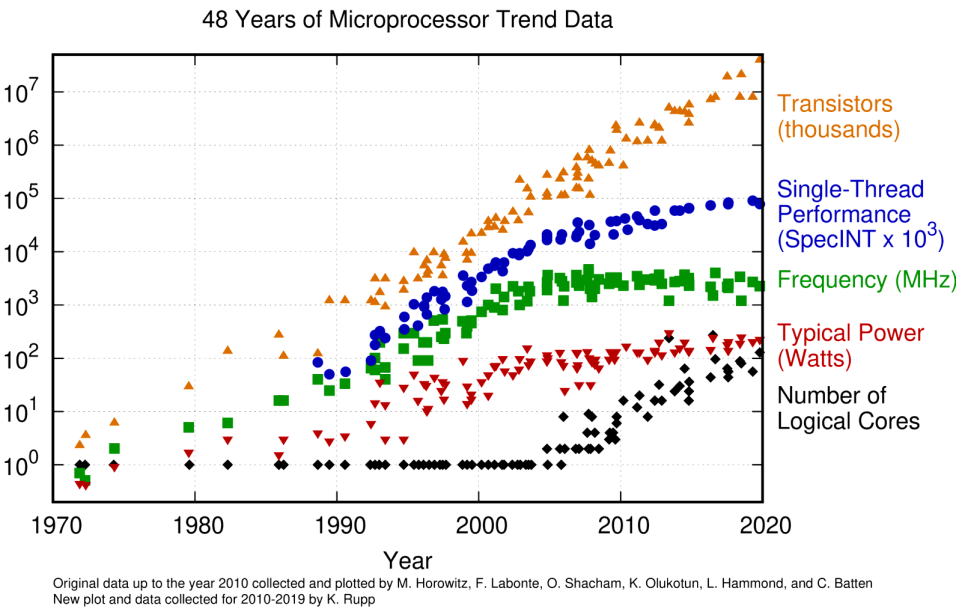


Figure 1: Changes of CPU parameters over 48 years [25, 26]

1.1 Motivation

1.1.1 Trends and Barriers in Scalable CMOS System Design

Moore's law predicts that the number of transistors in a chip doubles every two years, while the chip area remains constant. This observation has held true for many decades and it is still adequate today. Historically, as shown in Figure 1 [25,26], as the number of transistors grew, it resulted in an increase of the single core performance of the Central Processing Unit (CPU). However, since around the year 2005, it has been progressively more difficult to continue that trend as rising the core frequency has become increasingly impractical. These days, the main source of additional CPU performance is the increase in core count. As seen in Figure 1 [25, 26], starting from the year 2008, the single-thread performance, frequency, and power consumption have remained almost constant. Simultaneously, the number of CPU cores is following the slope of the graph that shows the transistor count.

Currently, the core count in processors is still increasing. While it may be not practical to continue this trend beyond certain point due to Amdahl's law [27], the additional chip space made available by the Moore's law also allows the chip designers to integrate specialized accelerators or components that have previously been located off-chip into the same system-on-chip (SoC) as the generic processor cores. Examples of such cores are graphics processors and memory controllers. The accelerators will increase the overall computation speed by decreasing the latency in the communication between the components. Additionally, higher levels of integration also help to reduce the physical size and power usage of devices.

Large-scale integration is pushed to extremes by mobile and embedded SoC manufacturers, who build entire computer systems on a single chip by integrating together components that they buy from third parties in the form of intellectual properties (IPs). Those systems include, in addition to CPU cores and graphics processors, also display controllers, wireless networking adapters, etc. However, SoCs with high core

count require a well-scalable, high-throughput interconnect that allows the cores to communicate to each other with minimal latency.

The common shared bus architecture that has been traditionally used for such purposes, does not scale very well and becomes a bottleneck for systems with high core counts. This scalability bottleneck, predicted by Figure 1 [25, 26], has been overcome by introduction of Networks-on-Chips (NoCs) [28]. Currently, NoCs are used in numerous commercial and academic many-core devices [29–34]. In contrast to bus-based interconnects, a NoC can provide simultaneous network access to all connected nodes. In addition, depending on the routing algorithm that is used within the NoC, it also enables multiple communication paths between each node pair. This helps to distribute traffic and to increase the reliably of the system.

1.1.2 Vulnerabilities of Nano-Scale CMOS Systems

Modern technologies provide vast performance at a relatively low power. This is a result of a decades-long trend in miniaturization of transistor's feature size. However, reducing size of the technology node beyond the sub-micron domain has also made the devices much more susceptive to faults [7, 8]. This translates into a considerable decrease in system's reliability [35–37].

Faults in computer systems can be caused by increased wear on the transistors due to a number or degenerative physical phenomena, such as hot carrier injection (HCI), negative-bias temperature instability (NBTI) [9–12], etc. These effects are more pronounced in sub-micron technologies and over time often result in the development of permanent faults as the transistors that are affected by these phenomena become incapable of performing their function within the required parameters.

However, faults can be also caused by environmental factors, most notably cosmic radiation [13, 14]. Consequently, faults caused by radiation are a large problem in devices located in high radiation environments, such as satellites [17–19, 38–42] whose orbits are located outside of the protection of Earth's magnetic field. In fact, the orbits of many satellites pass through the radiation belts surrounding the Earth [20, 21].

The amount of cosmic radiation that reaches Earth and ground-based technology is much lower than in space and reduces with altitude [13, 14]. However, random bit flips that are caused by cosmic radiation have been detected in not only airplane electronics [22], but also in terrestrial devices, such as in CCD camera sensors [43] and DRAM memories [44]. Radiation-induced faults in computer memories are especially large problem for servers, such as the ones used in data centers [24] and supercomputer clusters [23].

A radiation particle can flip bits in the memory or temporarily change signal values in combinational logic. However, faults in the combinational logic become dangerous only if the faulty value gets read into a memory element. Most radiation-induced faults are transient and mostly disappear either by themselves, or after overwriting the faulty value in the memory. Radiation-induced faults do not usually cause any lasting physical damage to the system [13, 14].

However, any fault, whether it is permanent, or transient, can lead to failures. Therefore, for many critical applications, fault tolerance mechanisms have become a hard requirement to guarantee that the systems work reliably, even under faults.

1.1.3 Reliability Challenges in Distributed Embedded Systems

A fault can occur at different parts of the system. In case of MPSoCs, one of the most critical areas is the interconnection network, which connects the processing elements. For example, a broken link between the routers in a NoC can render a part of the network inaccessible. The link failure, in this context, does not mean only that the physical wire between the routers itself is broken. Any fault in NoC router that renders at least one of the ports inoperable would manifest itself on a link, as a link fault. Additionally, even a transient link fault could cause misrouting and lead to a deadlock in the NoC, which renders the entire interconnection network unusable until a full reset of the system is performed. However, if a fault occurs in the processing element, it can overwrite variables in memory or cause a calculation error. These errors can easily propagate to the software level.

Most of the faults caused by radiation are transient and their effects disappear after a reset or memory scrubbing. For this reason, satellites and many other systems have watchdog timers that will monitor system for crashes and automatically reboot it if a crash is detected. However, a watchdog timer will not protect against faults that keep the application responsive but causes it to perform a calculation error. A calculation error, however, can lead to as bad as, or even worse side effects than a full system crash. In satellites, the number of radiation-triggered faults is usually minimized by utilizing special, radiation-hardened, components. However, the radiation-hardened components are much more expensive and are usually based on older technologies, when compared to commercial off-the-shelf (COTS) alternatives. As such, the COTS components provide a higher performance. For this reason, COTS components without built-in fault tolerance are preferred for low-cost satellites, or when high performance is required [45]. In this case, the fault tolerance needs to be implemented on top of the COTS components, for example in software.

All faults, except software bugs (which are out the scope of this thesis), occur in hardware. However, they often manifest at higher layers (e.g., in software). This means that fault tolerance for distributed embedded systems must be handled at different parts of the system. For example, data correction for communication links needs to be fast and thus should be implemented in hardware. On the other hand, overcoming the problems with permanently broken links requires a specialized system health monitoring unit (SHMU), such as the one proposed in Publication IV, which can reconfigure the network and would be located somewhere in between of the hardware and the application layers. Reconfiguration needs to be performed transparently to the application that running on the processing elements. At the same time, the SHMU should have a more abstract view of the network when compared to the routers. Finally, if COTS processing units, without built-in fault tolerance, are used, the faults in processing elements' hardware often manifest themselves in the software that is being run on the processing elements. This marks the need for a software-based method for detecting and mitigating the effect of faults in the processing elements.

1.2 Problem Formulation

On an abstract level, a distributed embedded system consists of processing elements, interconnects, and software running on these processing elements. Hardware faults, caused by either aging or defects (usually permanent faults) or by environmental effects, such as radiation (usually transient and correctable) can occur either in the processing elements or in the interconnects.

Faults in the interconnects usually manifest themselves as communication errors on the interconnection links. For example, some faults can render an output port of a NoC router non-functional. This will cause the link connected to this port to appear broken, which can cause some network nodes, or even entire sections of the network to be isolated from the rest of the NoC. Other types of faults in interconnect routers will cause wrong data to appear at an otherwise functional link.

In case of a processing element that consists of a general-purpose processor and a memory, faults often manifest themselves at the software layer. Any software application can be viewed as a finite state machine (FSM). Its state is a set of variables in the application, which hold certain values. Actions performed on these variables change the state. Faults can occur either in the memory, directly corrupting the state, or in the combinational logic of the processor, causing the new state to be wrongly calculated and a wrong value to be written into one or more variables. In processing elements, many hardware faults that do not get masked, will eventually manifest themselves at the software level.

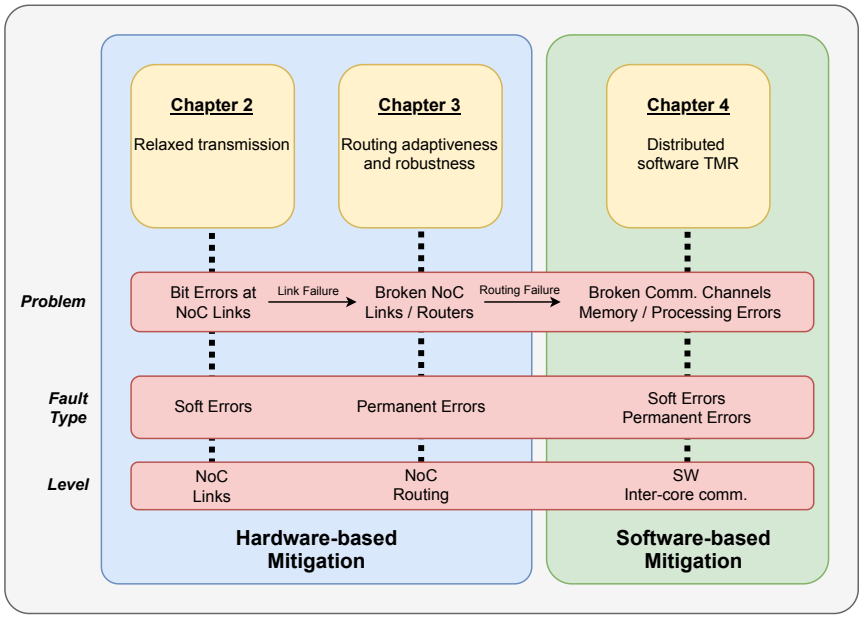


Figure 2: Fault tolerance approaches presented in this thesis

This dissertation presents three fault tolerance approaches for use in different parts of distributed embedded systems. While they are designed to work independently, the three fault tolerance approaches can be also used together for increased fault tolerance. As shown in Figure 2, the presented fault tolerance approaches are the following:

1. First, a relaxed transmission approach is presented to handle soft bit errors at NoC links that could otherwise lead to interconnect congestions or cause a packet re-transmission and thus additional latency. These errors are caused by transient faults in the combinational parts of the NoC routers. The proposed relaxed transmission mechanism is lightweight, very fast and can isolate and correct bit errors on NoC links.

2. When a permanent error occurs on the NoC link, changing a routing algorithm can often fix the problem by bypassing the broken link. To this end, an in-depth analysis of all two-dimensional (2D) turn model-based routing algorithms in a NoC is done and formal robustness and adaptivity metrics are provided. These metrics, together with the results of the in-depth analysis, allow the global fault manager in the NoC to reconfigure the routing algorithm to bypass NoC links which are broken due to permanent faults.

3. Finally, a distributed software-based fault tolerance algorithm, called STROBES, is proposed. STROBES can guarantee a fault free execution of mission critical software in the processing elements, which do not possess any hardware-based fault tolerance mechanisms. Furthermore, the proposed algorithm can correct non-permanent errors upon their detection. The approach presents formally provable constraints and due to its deterministic latencies, it is real-time capable.

1.3 Contributions

To address the problems described in the previous section, this thesis proposes three novel fault mitigation approaches at different layers of the system stack. The thesis goes beyond the existing state of the art by addressing the following aspects of dependability:

- **Relaxed Transmission Architecture** – A design of a relaxed transmission-based mechanism is provided for correcting soft errors on NoC links. Compared to similar approaches, the proposed mechanism does not require additional re-transmission registers and provides very fast error recovery time. Fault-free operation can be resumed at the next raising clock edge after the transient fault that caused the soft error on the link, has disappeared.

- **Comprehensive Performance and Robustness Analysis of 2D Turn Models for Network-on-Chips** – For a first time, an exhaustive analysis of all uniform 2D turn model-based routing algorithms for networks-on-chip is performed. While previous works only describe a small number of popular routing algorithms, the analysis performed in this thesis resulted in discovery of many new turn model-based routing algorithms that were not described before in the literature. Additionally, new metrics for qualitative comparison of turn model-based routing algorithms are proposed, which enable to utilize a novel approach to determine an optimal routing algorithm reconfiguration for achieving a maximized use of healthy NoC routing resources based on any specific permanent fault pattern.

- **Software TMR with Distributed Voting** – A novel high-level fault tolerance algorithm for protecting mission critical software is proposed. The proposed approach relies on a high-level fault model that allows to implement fault tolerance completely in software in a way that the performance of the fault tolerance algorithm does not depend on the behavior of the software it protects. Therefore, the proposed algorithm can be used to protect a wide range of applications against faults by connecting the application with the fault tolerance library through a simple interface. In comparison, most similar approaches are limited to only a very small range of applications with similar behavior or rely on modified or inherently fault tolerant hardware to achieve fault tolerance.

  The high-level fault model allows to fine-tune the algorithm's parameters to further optimize its fault tolerance and performance using a high-level, application independent simulation.

Additionally, the fault correction capabilities of the proposed algorithm are formally proven by Markov analysis and fault injection experiments using the aforementioned application independent high-level simulation tool.

The advantage of these approaches is that they go along with modern IP-based design techniques for scalable distributed embedded systems and MPSoCs. This thesis is based on the following specific contributions and publications.

The contributions on relaxed transmission are based on Publications **I**, **II** and **IV**:

- A novel low latency and low area fault tolerant transmission method for NoC links, called relaxed transmission, is presented for mitigating errors that are caused by transient faults at NoC links. These errors can be caused by, for example, single event transients (SETs) in the combinational part of the sending router's output port. In simplified terms, the relaxed retransmission approach pauses sending and receiving routers and repeats reading the same data until the transient fault has disappeared and error-free data is read from the link. The routers are un-paused and fault-free operation can be resumed at the next raising clock edge after the fault has disappeared.

- An optimized version of the proposed relaxed transmission mechanism is proposed, which takes advantage of the inherent redundancy found in the input buffers of NoC routers to avoid using of additional re-transmission buffers. Experimental results show the effectiveness of the proposed mechanism in very high fault rate environments.

However, a permanent fault in a NoC router can make some of the router's links unusable by breaking (a part of) a channel in a router. For this reason, the routing algorithm used in the NoC needs to be robust so that the system could continue working with broken NoC links. Additionally, a formal way of comparing the different routing algorithms in terms of their connectivity and adaptivity is needed. This leads to the contributions on connectivity and adaptivity in NoCs based on Publications **III** and **IV**:

- A connectivity metric for NoCs is proposed. The connectivity metric tells if a NoC under certain turn model (routing algorithm) can provide full connectivity between all node pairs in the network. This allows to filter out useless turn models, which cannot provide full connectivity. However, the connectivity metric can also be used by the system's fault manager for restoring connectivity by changing the turn model in a NoC after a link has suffered a permanent failure. The fault manager can simulate the effect of any turn model on a faulty NoC by applying it to the routing graph of the NoC with the broken link. It is formally proven that if a turn model achieves full connectivity on the graph representation of the broken NoC, it also guarantees full connectivity in the real NoC.

- An Extended Degree of Adaptivity (DoA_{ex}) metric for turn model evaluation is introduced. The proposed metric improves on the existing adaptivity (DoA) metric by considering non-minimal path routing. The existing DoA metric can only be used for calculating adaptivity for minimal path routing. The metric describes the number of alternative paths a packet can take under a turn model for non-minimal path routing. A turn model with higher Degree of Adaptivity is more fault tolerant.

- An algorithm for calculating average connectivity is provided. The average connectivity value calculated for k broken links is equal to connectivity values averaged over all fault configurations with k broken links. Average connectivity helps to compare turn models in terms of fault tolerance.
- An analysis of all uniform 2D turn model based routing algorithms for NoCs is performed and all deadlock free turn models that provide full connectivity (usable turn models), are extracted. Additionally, an in-depth comparison of all usable 2D uniform turn models in terms of connectivity, Degree of Adaptivity, Extended Degree of Adaptivity, average connectivity and latency, is performed. To the best of the author's knowledge, a full comparison of all uniform 2D turn model based routing algorithms has not been done before.
- Many new turn models that have not been described in the literature, but provide good fault tolerance with low latency, are discovered.
- A strong correlation between turn model's average connectivity, latency and its (extended) Degree of Adaptivity is shown.
- Finally, an algorithm is provided that can be used by the system's fault manager to choose the best turn model for any fault scenario.

On the other hand, transient faults can also occur in the processing elements themselves. However, in case this happens, it is essential that critical software continues to run reliably. This leads to the final contribution of the thesis, a software-based TMR approach for distributed embedded systems, called STROBES. This contribution is based on Publication **V**:

- In this thesis, a software-based fault tolerance algorithm, called STROBES, is proposed. The proposed algorithm is directly derived from a high-level fault model, which provides good coverage of faults that can occur in software, which is run on a multiprocessor system, while still being agnostic to the behavior of that software. STROBES' performance does not depend on the behavior of the protected application, but only on the constraints of the system it is run on and the size of the application's memory that is protected by the algorithm.
- Since the STROBES algorithm is application agnostic, the application can be updated without any changes to STROBES. Furthermore, existing applications can be easily adapted to work with STROBES. In order to make the protected application compatible with STROBES, it needs to be linked with the STROBES library, which can be accessed using a simple interface. This is in contrast with most other similar approaches, where the fault tolerance algorithm is tightly coupled to the protected application and cannot be separated.
- STROBES enables to run mission critical tasks on unmodified COTS processing elements, which do not have built-in fault tolerance. This is possible because STROBES is a fully software-based solution that does not require modification of the hardware it is running on.

- STROBES' performance can be fine-tuned using a set of internal parameters. These parameters can be estimated formally and then fine-tuned in a high-level simulation. Since STROBES is application agnostic, this simulation can be performed independently of the protected application. Only information about the basic properties of the application, such as the size of its state, are needed. Additionally, in this thesis, the design of a custom simulation tool is provided for this purpose.
- The STROBES algorithm runs the protected application in parallel on three processing elements. Errors in the application's state are detected using distributed majority voting. Non-permanent faults in the application's state can be corrected. The STROBES algorithm can additionally also handle network and timing errors in the distributed system.
- Reliability improvements of STROBES over a non-protected system are proven formally using Markov chains and experimentally through fault injection experiments by using a custom high-level application-independent simulation environment that is also presented in this thesis.

1.4 Thesis Organization

The rest of the thesis is organized as follows. First, Chapter 2 will give an introduction into the topic of dependability and familiarize the reader with the basics of Networks-on-Chips. The background chapter is followed by Chapter 3 which introduces the concept of relaxed transmission and its application on protecting combinational logic in NoC routers against transient faults. Next, in Chapter 4 an in-depth analysis of all uniform 2D turn model-based routing algorithms is performed. The analysis results in the discovery of many new turn models with good performance that have not been described previously in the literature. Chapter 5 introduces a novel application agnostic fault tolerance algorithm for handling faults in critical software that is running on processing elements without built-in hardware-based fault tolerance. The proposed solution does not require any modifications to the hardware it is running on. Finally, the thesis is concluded by Chapter 6.

2 Background

2.1 Dependability

Dependability is defined as the ability of a system to deliver its intended level of service to its users [46]. The term "dependability" is often mistakenly used interchangeably with "reliability". However, dependability is a much wider term, with reliability being one of its components. In addition to being reliable (behaving according to its specification), a dependable system must also be available and safe. Additionally, a dependable system not any malicious entities tamper with, in other words, it needs to be secure. However, in this thesis, only fault tolerance against naturally occurring faults is investigated. Errors and failures introduced deliberately by a malicious entity are out of the scope of thesis.

To this end, the three parameters of dependability mentioned above that are investigated in this thesis – reliability, availability and safety – are defined in the literature as follows [46]:

Reliability $R(t)$ is the probability that the system will work during the interval $[0,t]$ without any failures. In other words, system's reliability is a probabilistic measure of its continuous fault-free functioning. The more reliable a system is, the less likely it is for it to fail at any given time. A faulty system can still be reliable, if the faults do not manifest themselves as failures, e.g., the faults are masked or otherwise mitigated by some form of fault tolerance mechanism (see Section 2.1.3) and the system continues to function withing normal parameters.

Availability $A(t)$ of a system is the probability that the system is functioning properly at time t. Unlike reliability, the availability parameter also considers any possible downtime of the system due to repairs. A higher the reliability of the system, and faster repair times, also result in higher availability. Availability of a system, which cannot be repaired after a failure has occurred, is equal to its the system's reliability.

Safety $S(t)$ is defined as a probability that the system is in a safe state at time t. A safe state is defined as a state in which the system does not pose any danger to people or the environment surrounding it. It can be that a system is not reliable, or even available, but still safe, if it fails in a way where it does not pose danger to its surroundings.

2.1.1 Dependability Impairments

Faults, Errors, and Failures

Based on their severity, problems in the systems can be divided into faults, errors or failures. A **fault** is a defect in a system. A fault can be either the result of manufacturing problems (a manufacturing defect), or it could appear during the device's lifetime. Faults which occur during the lifetime of a device can be the result of normal wear on the system (aging) or a direct influence from the environment (vibration, radiation, etc.).

A fault becomes an **error** when it causes a difference in the system's behavior. However, not all faults result in errors. For example, a fault might get either neutralized using by a fault tolerance mechanism or just masked by digital logic. For example, a faulty value at an input of an AND gate gets masked by the gate and will not be visible at its output if at the non-faulty input of the AND gate is the value '0'.

A **failure** is an error that causes the device to behave in a way that contradicts the specification. If an error produces an output, which is different, compared to a non-faulty system, but still valid according to the specification, it is not considered a failure. An example if this is an error in the routing logic of a NoC router that causes a

packet to be routed into a wrong output. If the wrong output is still valid according to the routing algorithm, it is not a failure.

A failure can result in an expensive repair, or even complete loss of system if the system it cannot be repaired or if the repair is too expensive (e.g., in case of satellites). Additionally, a failure in a poorly designed system can also cause a serious safety hazard and become dangerous to people or its environment. For these reasons, it is necessary to minimize the probability of faults leading to failures in devices.

Types of Faults

Based on their behavior and the underlying cause, faults can be roughly categorized into three group – permanent, intermittent and transient faults [46].

A **permanent fault** is a defect in a device that cannot be corrected without replacing the broken component. The cause of permanent faults can be either a manufacturing defect, or it can develop over time during the system's lifetime, for example, due to aging of the transistors.

An **intermittent fault** is a temporary (non-permanent) fault that occurs randomly at the same position. The cause of an intermittent fault is always a defect in the device itself. An example of an intermittent fault can be a cold solder joint in an electronic device, which connects and disconnects randomly due to vibration. In time, intermittent faults often turn into permanent faults.

A **transient fault** is a randomly occurring temporary fault. Unlike intermittent faults, transient faults are usually caused by environmental phenomena. For example, a radiation particle can trigger a transient fault in the system by hitting a transistor. This can cause a short voltage spike at the transistor's output, which could be interpreted as a change in the logic value by the rest of the circuit. This type of transient fault is called a single event effect (SEE). Transient faults can be differentiated from intermittent faults by the fact that intermittent faults occur repeatedly at the same physical location, while transient faults are random in their nature. Transient faults do not cause permanent damage to the system and their effects will dissipate over a short time. In case the faulty value is latched into a memory element, overwriting the value with a known good value will correct the fault. However, if the faulty value is not detected in time and corrective actions are not taken, the transient fault can still result in a failure.

2.1.2 Mechanisms Behind Radiation-Induced Transient Faults

The radiation particles that are responsible for causing SEEs in modern electronics come from different sources. In the literature, there are reports of cases where materials used in the manufacturing of integrated circuits (ICs) were contaminated with radioactive elements [47, 48]. This resulted in the IC packaging or solder material itself radiating alpha particles and causing SEE-induced faults in the system.

However, most particles that are responsible for SEEs in electronics can be traced back to cosmic radiation. Cosmic radiation is, obviously, a big problem for satellites and other computer systems that are positioned in space [16–19]. These systems receive a large amount of radiation from the sun, but also in form of cosmic rays which originate outside the solar system. Additionally, the Earth is surrounded by the Van Allen radiation belts. The radiation belts consist of a lot of charged particles that have been caught and trapped by the Earth's magnetic field. Satellites whose orbits pass through the Van Allen belts also receive a large amount of radiation [20, 21] from the particles in the belts.

However, radiation-induced errors are not limited to devices in located space. They are also a notable problem for terrestrial applications. Soft errors – non-permanent errors that are largely attributed to cosmic radiation, have been detected not only in supercomputers [23] and servers [24] in data centers, but also in camera sensors [43] and even in pacemakers [47]. At ground level, the main source of high energy particles is still the cosmic radiation [13, 14]. However, unlike in satellites, the most damage to terrestrial systems comes from cosmic neutrons, not charged particles, such as ions or electrons. This is the case because most of the charged particles will be deflected by Earth's magnetic field and usually do not make it to the ground level. Neutrons lack charge and are not influenced by the magnetic field [14]. However, neutrons cannot cause SEEs on their own since they lack an electric charge. For this reason, the SEEs caused on ground level are mostly due to highly energetic secondary ions [14]. These ions are produced as the result of a collision between cosmic neutrons and other atoms in the atmosphere or with the silicon atoms in ICs [49]. Additionally, studies have suggested that even low energy cosmic neutrons can indirectly cause SEEs by inducing a fission process in boron-10 atoms. As a side effect of such event, an alpha particle is generated, which can cause SEEs. Unfortunately, boron-10 is often used in the insulation materials in integrated circuits (ICs) [50].

A radiation-induced fault (an SEE) occurs when a high-energy, charged, particle hits a transistor. This starts a chain of events that can lead to an SEE-induced voltage pulse to be generated at the output of the transistor. The most sensitive parts of the transistor regarding to charged particles are its reverse-biased junctions. The closer to the junction the particle hits, the stronger will be the effect [13].

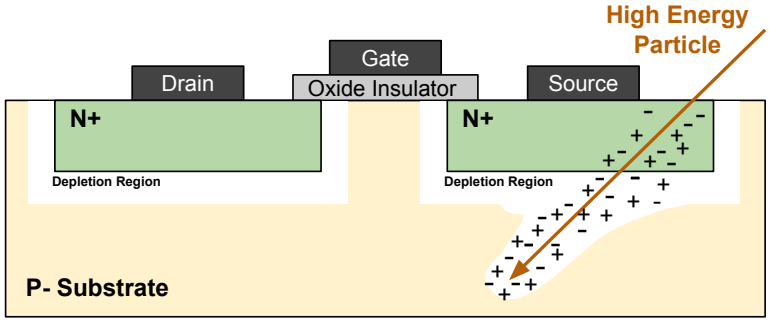


Figure 3: Effects of a particle strike on a transistor

The sequence of events that lead to an SEE upon a charged particle hitting a transistor have been described in detail in [13]. The process explained in [13] is illustrated by Figure 3 and it can be summarized as follows: When a charged particle hits the transistor, it causes the formation of a track of electron-hole pairs with very high carrier concentration along the trajectory of the energetic particle. Once this track reaches the depletion region, the generated electric field starts to collect charge carriers from the region. If the track extends further and reaches the substrate, a funnel-shaped electric field forms that extends the depletion region deep into the substrate and further increases its ability to collect charge carriers. This phase usually lasts tens of picoseconds [14], after which, the funnel collapses. The next stage is the diffusion phase, when electrons are diffused back into the depletion region, and the situation normalizes. During the diffusion stage, charge is collected. When the amount of collected charge is exceeds a critical level ($Q_{coll} > Q_{crit}$), a current pulse is generated. If this occurs, a

voltage spike, is created at the affected junction. The length of the current pulse is typically around 200 picoseconds, but the actual length depends on many parameters, such as the energy and trajectory of the particle, but also on parameters of the transistor itself [14].

If the voltage spice occurs in combinatorial circuit, it results in a transient fault, called a single event transient (SET) [46]. An SET by itself is not dangerous since it does not permanently damage the system and its effects disappear by themselves. However, a sufficiently strong SET can propagate through the circuit. If the SET is not masked by the logic gates that it passes through, the faulty signal may be latched into a memory element, causing a single even upset (SEU) [51, 52]. The same can occur when the particle directly hits a memory element, causing a bit flip. An SEU can be easily corrected by overwriting the faulty bit in the memory with a correct value [46]. However, if the SEU is not detected and corrected early enough, it can lead to serious errors or even to a failure.

The critical charge (Q_{crit}) depends on many parameters of the involved node. Most notably, the operation voltage of the device [14]. Since newer devices run at lower voltage, it means that newer technologies are more susceptive to SEEs. Additionally, in older technologies, the problem of an SET being latched into a memory element was not a large problem because of the relatively large sized of the gates and slow clock speeds. However, due to lower propagation delays in modern systems, SETs can traverse through many gates before dissipating. At the same time, fast clock rates increase the probability of an SET becoming getting read into a memory element, and thus causing an error, considerably [13].

2.1.3 Fault Tolerance

To guarantee fault-free functioning of critical systems, such as satellites [16–19], servers [23, 24] and even pace makers [47], it is necessary to implement fault tolerance mechanisms. The goal of fault tolerance is to guarantee that any faults that occur in the system will be corrected before they have a chance to cause an error or failure. All fault tolerance approaches implement some form of redundancy. On an abstract level, fault tolerance mechanisms can be divided mechanisms based on fault masking and mechanisms, which implement some form of fault detector, coupled with mechanisms for localizing, containing and recovering from the fault.

Fault Masking

The simplest probably most widespread form of fault-masking is TMR [53]. Essentially, in a system that utilizes TMR, the critical components of that system are triplicated. The outputs of the three components are then compared by using a majority voter. When a fault occurs in one of the triplicated modules, the output of the faulty component will differ from the correct components. However, since the other two functional components still produce identical results, the faulty value is masked. This results in majority voting still producing a correct result. The main benefit of TMR is its simplicity. For this reason, TMR can be implemented for almost any system. This can be done both in hardware and in software.

However, TMR has also disadvantages. Mainly, the cost of triplication of components is high. Not only financially, but also in terms of power usage, physical size, and, in case of software-based solutions, performance. Additionally, the majority voter is essentially a single point of failure in the system. While this issue can be corrected by also triplicating the checking mechanism [54], it will increase the cost even further.

Fault Detection, Localization, Containment and Recovery

The limitations of TMR motivate the development of alternative fault tolerance solutions. Usually those approaches work by replacing some of the redundancy introduced by TMR with time or data redundancy. However, it is important to realize that such solutions can only be developed for specific use cases and are not generic, like TMR. In general, those approaches work in four steps – fault detection, localization, containment and recovery from the fault.

The easiest way of detecting faults is by duplicating of the critical component and majority voting on the outputs of these components (DMR – Double-Modular Redundancy). Unlike TMR, duplication cannot mask the faults, but it can still detect them. Duplication, compared to TMR, reduces the cost, but can still be expensive, especially since it provides no fault correction. However, in many cases, DMR can be further optimized by implementing a system which compares the work of the critical component to a golden model on a more abstract level. Examples for this are functional online checkers [55], which are, essentially, hardware assertions that verify the components' work against the specification in real time.

Another way for detecting faults is to use information or time redundancy. Information redundancy is normally used in data transfer and storage, where a parity bit, checksum or error correction code is added to the data. Time redundancy usually involves re-execution of code or re-transmitting the to resolve transmission errors.

2.2 Networks-on-Chip

The ongoing down-scaling of transistor's feature size has made possible the emergence of multi and many-core SoCs with ever-higher numbers of cores. As the number of Processing Elements (PEs) in the SoC has increased, traditional bus-based communication approaches became a bottleneck for system's speed and reliability. Networks-on-Chip (NoCs) [28] have emerged as an alternative communication paradigm to overcome these limitations.

Presently, NoCs have become the primary inter-core communication mechanism for used for many-core chips with high core count. NoCs are not only used in academic research [29, 30], but also found their way into commercially produced devices. For example, NoC are used in Intel's Skylake-based Xeon server processors [31] and their Stratix 10 FPGAs [56]. Additionally, AMD's new chips [32] use their proprietary NoC-based Infinity Fabric interconnect [57], which utilizes an improved version of NoC-based interconnect from older Opteron [33] server processors. Additionally, Infinity Fabric supports 3D NoCs by vertically integrating multiple chiplets [57] on an interposer that joins the NoCs on the chiplets into a single network [34].

In a NoC-based system, the traditional bus-based interconnect is replaced with inter-connected routers and data between the PEs is transmitted in form of packets over an interconnection network, the NoC. While NoCs can be built using different network topologies, this dissertation considers only the full mesh topology because it offers many alterative paths between all nodes in the network. The existence of multiple alternative paths is necessary for fault tolerance. Figure 4 depicts an abstract, high-level structure of a NoC-based SoC utilizing 3×3 full mesh topology. Each network node in such a NoC consists of a NoC router (colored blue in the figure) and a processing element (green). In a full mesh topology, each router is connected using bi-directional links to a local PE and to all of its neighboring routers.

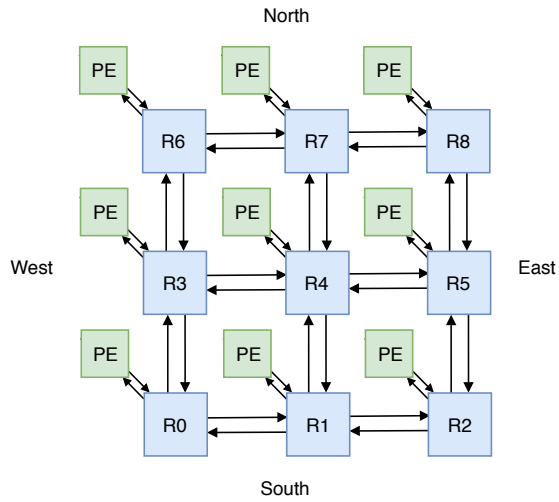


Figure 4: High-level structure diagram of a 3×3 NoC-based SoC utilizing the full mesh topology

The rest of this section of the thesis explains main technologies and mechanisms utilized in NoCs, using the open source Bonfire NoC router, proposed in Publication IV and [58] as an example. The Bonfire router is used through this dissertation as the main research and experimentation platform.

2.2.1 Turn Model-Based Routing

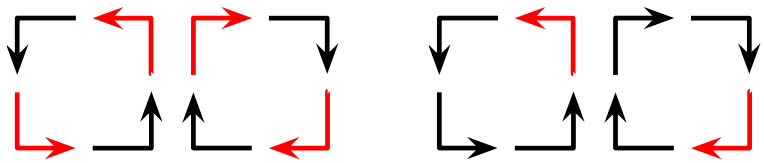
The full mesh topology has a very regular structure, which does not change during the system's lifetime. Even when a network node or a link between the nodes fails, the rest of the structure and connections will remain the same. By knowing this, it is possible to implement simple, yet scalable, routing algorithms that are based on the concept of *turns*. These routing algorithms are called *turn models* [28, 59, 60].

In order to better understand the concept of turn models, some background information is needed on the coordinate systems used in NoCs. Every router in a two-dimensional (2D) full mesh NoC has four possible directions to route packets to, in addition to the local connection with its PE. The four directions are named after the cardinal directions as North (N), East (E), South (S) and West (W) (see Figure 4). Additionally, two axes can be defined in the network based on the cartesian coordinate system. The horizontal *West* $\leftrightarrow$ *East* axis is called the *X-axis* and the vertical *North* $\leftrightarrow$ *South* axis is called the *Y-axis*.

The main concept of a turn model-based routing algorithm is that in every router in the NoC certain turns are prohibited. A turn, in this context, is defined as a situation where a packet changes the axis it is travelling on while passing through the router. An example of a turn is a packet that enters the router from the South input and is routed to the East output. In this case, the packet switches from the Y-axis to the X-axis. Straight paths through the router, where the packet does not change its axis, are not considered turns and are always allowed in all turn models. In addition, turn models also do not regulate connections to and from the local port. The local connections are always allowed. If all routers in the NoC are using the same set of disabled and enabled turns, it is said that the turn model is *uniform*.

One of the main advantages is that the routing logic can be implemented in a distributed fashion across all routers, using Logic Based Distributed Routing (LBDR) [61]. In a NoC that is utilizing LBDR, the routing decision is made locally in each router, based on certain criteria. LBDR has a very small overhead compared to other methods, since the packets do not have to contain the entire routing path. Additionally, LBDR can be implemented using only combinational logic, with only a couple of logic gates. It takes much less area when compared to routing tables [61].

In 2D full mesh NoCs, there are 8 possible turns in each router, E2S (East-to-South), E2N, N2E, N2W, W2N, W2S, S2E, S2E. Under this notation, for example, E2S means that a packet enters the East input port of the router and it is forwarded to the South output port. Under a turn model, each one of the turns can be either allowed or disallowed, which leaves 256 possible uniform turn models, combinations of enabled and disabled turns. However, not all turn models are *usable*. If a turn model is not constrained enough, deadlocks can occur. A deadlock in a NoC is a situation, where packets are waiting for resources in a cyclic manner. As such, a deadlock can clog the network and often results in a system failure. On the other hand, if the turn model is too constrained, it might be impossible to form connections between all the node pairs in the network. A usable turn model must satisfy both conditions – it should not allow deadlocks to occur but still provide connectivity between all node pairs. This problem is investigated in more detail in Chapter 4.



(a) XY turn model *(b) West-First turn model*

Figure 5: Example turn models. Red arrows denote a disabled turn, black arrows represent an enabled turn

Figure 5 depicts two well-known turn models. In the figure, a red arrow denotes a turn which is disallowed under the turn model and a black arrow denotes an allowed turn. Subfigure 5a shows the XY turn model [62]. Under the XY turn model, a packet that is already travelling on the Y axis is not allowed to make a turn that would take it to the X axis. This means that packets must first travel to their destination column on the X axis and only then change to the Y axis to reach their destination. The XY routing model is one of two *deterministic* [62] turn models, the other being YX. A deterministic turn model means that there is always exactly one path between each two nodes. While this approach provides predictable data arrival times, deterministic turn models are not very robust, since under deterministic turn models, it is not possible to bypass broken communication links.

Subfigure 5b shows the West-First [63] turn model. The West-Fist turn model does not allow turning to West once a packet is moving on the Y axis. This means that if the destination is to the West of the current router, the packet needs to be routed to the West direction, because a turn to the West cannot be taken once the packet is moving in any other direction. This needs to happen even if the destination is not on the same row, as the router which is making the routing decision. As this turn model is not as constrained as the XY model and provides multiple paths between each node pair, it called an *adaptive* [62] turn model. The conditions that are used for choosing

the paths the packet takes, depend on the implementation of the routing logic. For example, the packet can take the path with the least traffic. Additionally, a high Degree of Adaptivity makes the turn model more *robust*. More robust turn models have higher probability of finding an alternative path between two nodes in case of link failures (see Chapter 4).

The concept of turn models, including conditions for finding usable turn models and a detailed analysis of all the usable turn models in terms of average connectivity, adaptivity and latency is discussed in more detail in Chapter 4.

2.2.2 Switching Strategies

In essence, switching strategies in NoC are different techniques that are used by the NoC routers for forwarding data from the router's input port the correct output port. There are many different switching strategies, which are described in detail in the literature [62]. Therefore, in this section, an overview of only two well-known strategies, Store-and-Forward (SaF) and Wormhole switching, are explained, since they are needed for understanding the approaches presented in this thesis.

SaF, also called packet switching, is the simplest and the most straightforward switching strategy. In case of SaF, the entire packet is received and stored in the router's input buffer or internal memory before forwarding it to the output port. While the SaF strategy is very simple and, arguably, the easiest to implement, it also comes with many drawbacks. Mainly SaF requires very large input buffers or additional memory for storing the entire packets. Large buffers, however, mean high chip area usage and consume a lot of power. Additionally, the process of waiting for an entire packet to be received before forwarding it to the next router results in substantial latency overhead.

The shortcomings of the SaF switching strategy have been solved by Wormhole switching. In Wormhole switching, packets are divided into smaller transfer units, called flits. Every packet includes three types of flits: header flits, body flits and tail flits. The header flits contain the information that is required for routing the incoming packet, such as the address of the packet's destination node. When a header flit is received, the following sequence of events occurs:

1. The router reads the information required for routing the packet from the header flit and uses that information to make a routing decision.

2. Resources that are required for forwarding the packet are allocated. This will form a channel or a "wormhole" through the router, from the input port to an output port.

3. Afterwards, all arriving flits are immediately forwarded through the "wormhole" to the correct output, without buffering the entire packet.

4. When the tail flit is received, the resources used for the "wormhole" are freed up.

In contrast to SaF switching, the Wormhole switching strategy does not require input buffers or additional memory for storing the packet, since all flits are immediately forwarded to the output, provided that the required resources are available. However, usually small input buffers are used for load balancing purposes. For example, the Bonfire NoC router, which uses Wormhole switching, also has input buffers that can store up to three flits at any time.

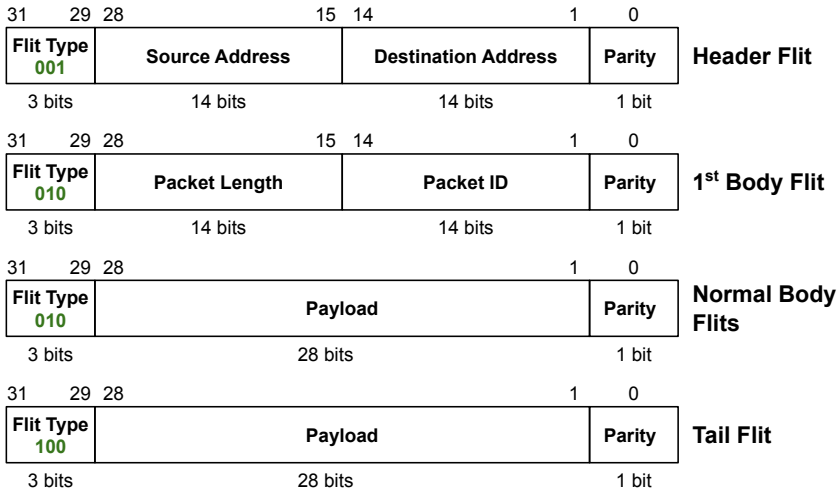


Figure 6: The packet format used by the Bonfire NoC

2.2.3 Example Packet Structure

Most of the work presented in this dissertation utilizes the open-source Bonfire Network-on-Chip router (see Publication IV) as the platform of choice for obtaining the experimental results. Every router in Bonfire NoC has a bi-directional (full duplex) connection with up to four other routers and a local connection to the locally connected PE. In each direction, 32 bits of data can be transferred at once over a parallel interface. As such, all packets are divided into 32-bit wide flits. Each flit can carry up to 28 bits of data. The rest of the space in the flits is taken up by a 3 bits long flit type indicator and a parity bit. The packet format that is used in Bonfire NoC is shown in Figure 6. In general, there are four types of flits:

- **Header flit.** A newly received header flit marks the arrival of a new packet. It causes the router to allocate resources in preparation for routing an incoming packet. Additionally, the header flit contains the source and destination addresses, which are needed for routing the packet towards the correct destination.
- **First body flit.** The header flit is followed by body flits. Usually, the first body flit is used to store additional information packet information, such as the packet's length and identification code. The information stored in the first body flit is not needed for routing decisions, but it can be useful for the application layer.
- **Normal body flits.** The rest of the body flits are used for actual data transfer. The number of body flits can differ from packet to packet, depending on the amount of data that needs to be transferred. The number of body flits can range from zero to $(2^{14}) - 3$.

- **Tail flit.** Every packet ends with a tail flit. The tail flit is very similar to body flits, except it carries a different flit identification code than body flits. When a tail flit is received, it indicates to the router that it has received the last flit of the packet. When the body flit is forwarded, all resources that were allocated for the packet are released. Additionally, the body flits in Bonfire NoC can carry 28 bits of data.

2.2.4 Flow Control Mechanism

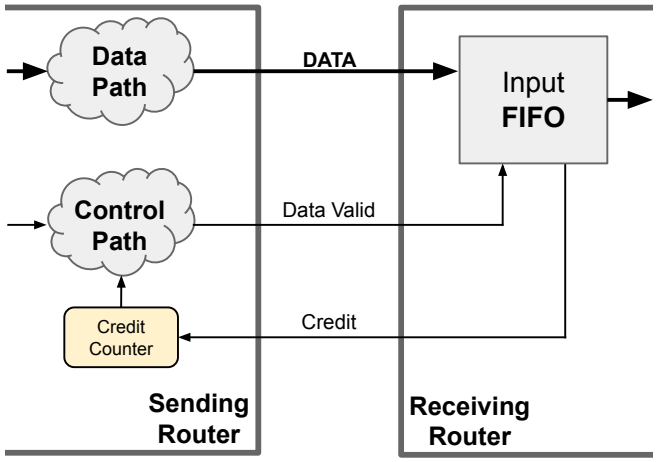


Figure 7: Credit-based flow control mechanism

The flow control mechanism is needed in NoC to provide communication between the sender and receiver nodes [62]. The receiver uses flow control mechanism for communicating to the sender when it is ready to receive new data. The sender uses the flow control mechanism to inform the receiver of its intent to send data. Different flow control protocols exist for providing such communication in NoCs [62]. This chapter provides an overview of the credit-based flow control mechanism that is used in the Bonfire NoC.

Figure 7 shows a single direction of the full duplex inter-router link. The opposite direction of the link is identical to the one depicted in the figure. In Figure 7 for simplicity only the data and control signals and components, which are required for explaining the credit-based flow control mechanism, are shown.

Using credit-based flow control, it is possible to send data at maximum rate of one flit per clock cycle. To achieve data transmission, the *Sending Router* puts that data on the *DATA* line and generates a pulse with length of one clock cycle on the *valid* line. This pulse informs the *Receiving Router* of incoming data. Upon detecting the pulse on the *Valid* line, the *Receiving Router* reads the data into its input buffer (*Input FIFO*).

However, data transmission can only work if there is enough free space in the input buffer of the *Receiving Router*. This problem is solved in credit-based flow control by the addition of a credit counter into the *Sending Router*. At reset, the *Credit Counter* is set to the maximum number of flits that can be stored in the *Receiving Router's Input FIFO*.

Every time a flit is transmitted, the value inside the *Credit Counter* is decreased by one, since the available storage in the *Receiving Router's Input FIFO* has decreased. This way, while the value in the *Credit Counter* remains larger than zero, the *Sending Router* can transmit data with the maximum speed of one flit per clock cycle. When the value in *Credit Counter* reaches zero, it means that the input buffer of the *Receiving Router* is full and no more data can be sent.

When space in the *Receiving Router's* input buffer frees up, a one clock cycle long pulse is generated at the *Credit* line. This pulse causes the value in the *Credit Counter* to be increased by one. As such, the value in the *Credit Counter* represents the amount of free space in the *Input FIFO* of the *Receiving Router*.

2.2.5 Bonfire Router Architecture

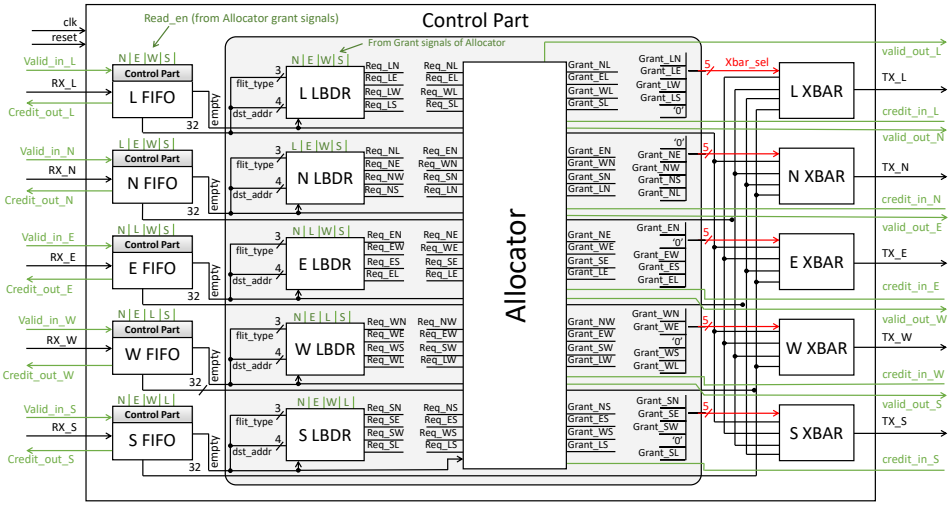


Figure 8: Block schematic of the open-source Bonfire router [58]

This section gives an overview of the NoC router's architecture, using the Bonfire router as an example. The architecture of the Bonfire router can be seen in Figure 8. The router has five identical full duplex communication ports: four inter-router connections and one local connection for communicating to the local PE. The router can be divided into four stages (from left to right in Figure 8):

1. **Links.** The physical connections between two routers or the router and the PE that is connected to the router.

2. **Input Buffers.** The input FIFO buffers are used to store the incoming flits. The depth of the buffers depends on the switching scheme that is used, average predicted network load and other design constraints. In case of Wormhole switching, the router does not need very deep input buffers because it is not necessary to store an entire packet. For routers with Wormhole switching, the buffer depth is a trade-off between the ability to cope with high network loads and chip area and power usage.

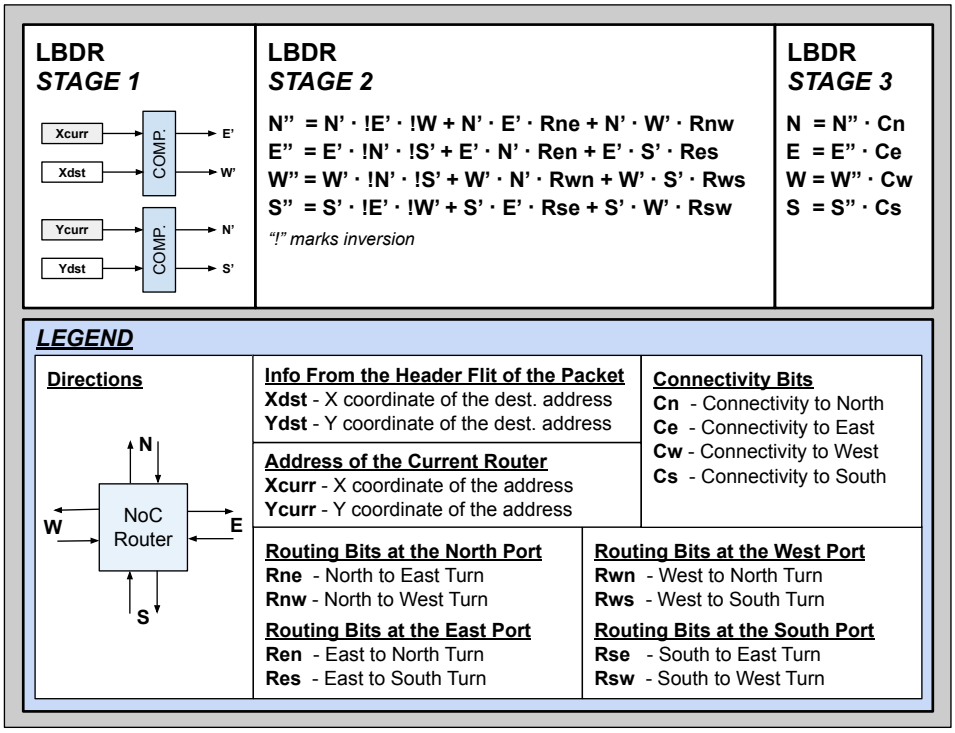


Figure 9: Description of the logic-based distributed routing (LBDR) method [61]

3. **Routing logic** – The Bonfire router uses LBDR [61] for making routing decisions. The LBDR method is shown in Figure 9. LBDR works in a distributed manner. Each router makes the routing decision by themselves, based on its own address in the network and the address of the destination node that is stored in the packet header. LBDR relies on the fact that the direction towards every node in a NoC, relative to the current router, can always be easily calculated. This allows implementing the routing mechanism as combinational logic, using only a few gates.

   In LBDR-based NoCs, each router has two control registers, which are used to configure the routing logic. The R_{xy} register contains eight bits that represent the turn model; each bit corresponds to a specific turn, '1' means the turn is enabled and '0' means that the turn is disabled. The connectivity register C_i contains four bits of data, out of which each corresponds to a cardinal direction. '1' in the C_i register means that there is a connection in that direction and '0' means there is no connection. This can be used to disable routing "over the edge" of the network for routers on the network border. Alternatively, the connectivity bits can be used by the fault management system to disable routing over faulty links.

   LBDR makes the routing decisions in three stages:

   Stage 1. First, the direction of the destination, in relation to the current router, is determined. This is achieved by using two comparators that compare the X and the Y component of the destination address to the respective values in the current router's address, as seen in Figure 9. If the X component of the destination

address X_{dst} is larger than the X component of the current router's address X_{curr}, the destination must be East from the current router. As such, the E' output is activated. In case X_{dst} is smaller than X_{curr}, the destination must be West from the current router, causing the W' output to be activated. When $X_{\text{dst}} == X_{\text{curr}}$, the destination is in the same row as the current router and both comparator outputs are kept at '0'. The direction on the Y axis is calculated in a similar manner. When both the X and Y components of the destination are equal to current router's address, the destination is the current router and the incoming packet forwarded to the local port.

Stage 2. Next, the requests for the arbitration unit are calculated. Request calculations for all routing bits are shown in Figure 9. More information on this can be found in [61]. To explain the logic behind the equations, let's consider the calculation of the North request. The markings N', E', W', S' represent the respective outputs of the comparator stage:

$$N'' = N' \cdot \overline{E'} \cdot \overline{W'} + N' \cdot E' \cdot R_{ne} + N' \cdot W' \cdot R_{nw}$$

The North request is activated when one or more of the following conditions is satisfied:

- $N' \cdot \overline{E'} \cdot \overline{W'}$ – The destination is exactly North from the current router (both the East and West outputs of the X comparator are '0')
- $N' \cdot E' \cdot R_{ne}$ – The destination is located North-East from the current router and the routing bit for North to East is "1". The routing bit represents the North to East turn in the turn model. If the routing bit is "1", the turn is allowed. Therefore, if the packet is routed to North in the current router, it will be sent to East in the next router.
- $N' \cdot W' \cdot R_{nw}$ – The destination is located North-West from the current router and the routing bit for the turn from North to West is enabled.

As seen in Figure 9, the rest of the requests are constructed similarly. Additionally, the routing logic can output more than one request. For example, under adaptive turn models, it might be possible to reach a destination that is located North-East from the current router, via both the North and the East output. In this case, the final decision on which output to use depends on the implementation of the arbitration logic. This behavior allows to use routing algorithms, which can produce multiple paths between the nodes.

Stage 3. Finally, before the requests are sent to the arbiter, they are further filtered to only allow requests for valid directions:

$$N = N'' \cdot Cn$$

For example, routers that are in the first row of the network do not have any other routers located in North of them, therefore the North connectivity bit (Cn) will always be '0'. As such, in these routers, the North request would never propagate to the arbiter. Additionally, the connectivity bit might be set to '0' on purpose by the global fault manager to disable routing over a broken link.

4. **Arbitration Logic** – The arbiter (also called allocator) is responsible for resource allocation in the router. In the Bonfire NoC router, each output port can be used by only a single packet at a time (there is no interleaving of flits). Whenever a request for an output is generated by the LBDR, the allocator will allocate the output resource for the incoming packet.

   Resource allocation is done separately for each output and in the Bonfire router, the allocator uses the round-robin algorithm for processing the requests. Thus, creating a fair and equal chance for all inputs to access the output port. As mentioned before, under an adaptive turn model, LBDR can generate multiple requests to different outputs. In this case, it is also the allocator's task to make the final routing decision, based on resource availability. The allocator in the Bonfire router will just choose the first requested output that is available. However, the decision process could be performed in a more complex manner by considering the network load, fault information, etc.

   Once the final routing decision is made, a grant signal is generated to for the crossbar to commit the data transaction.

   Additionally, in the Bonfire routers, the arbitration logic contains the credit counter and is therefore responsible for handling flow control.

5. **Crossbar** – The crossbar (also known as Xbar) is responsible for creating a physical connection between the input and the output ports to commit the actual data transfer. The crossbar is essentially just a 32-bit wide multiplexer that can connect all input buffers directly to the output port. There is one crossbar for each output port. The crossbar's input selection is made based on the grant signal from the allocator.

3 Relaxed Transmission Architecture

3.1 Introduction

This chapter proposes a novel Relaxed Transmission (RT)-based fault tolerant data transmission method, which can mitigate soft errors in data that is transferred over NoC links. Unlike other approaches, the proposed mechanism has very low fault correction latency and does not require additional retransmission buffers because it takes advantage of the already existing redundancy found in the input buffers of NoC routers.

The soft errors in NoC links are caused by transient faults, like SETs, which occur in the combinational logic at the output port of the router that is transmitting the data. In the proposed approach, transmission errors are detected at the receiver's side using a parity checker. Fault detection causes system to pause its operation and re-transmit the same data until the transient fault has disappeared. Fault-free operation can resume at the next raising clock edge after the fault has disappeared.

The rest of this chapter is organized as follows. Section 3.2 will provide an overview of related works. Next, Section 3.3 investigates the effect of soft errors in the data path of NoC routers. The proposed Relaxed Transmission mechanism's architecture is presented in Section 3.4. Next, an optimized version of the proposed RT mechanism with smaller area and power overhead, is introduced in Section 3.5. This is followed by experimental results in Section 3.6. Finally, the chapter is concluded in Section 3.7.

This chapter is based on the following publications:

- **I**
  K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "AWAIT: An ultra-lightweight soft-error mitigation mechanism for Network-on-Chip links," in *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–6, July 2018

- **II**
  K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "Handling of SETs on NoC links by exploitation of inherent redundancy in circular input buffers," in *2018 16th Biennial Baltic Electronics Conference (BEC)*, pp. 1–4, Oct 2018

- **IV**
  S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in Network-on-Chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 48–53, April 2017

3.2 Literature Review

Many works have investigated the problem of SET mitigation in NoC links (see Table 1). However, most of the mechanisms proposed in the literature are very costly in terms of latency or area or have a large data overhead. In general, all fault tolerance mechanisms used for guaranteeing data integrity at NoC links can be categorized, based on the fault detection and mitigation granularity of the approach, as *End-to-End (E2E)* or *Hop-by-Hop (HBH)* mechanisms.

Table 1: Comparison of approaches for protecting NoC links against the effects of soft errors

Approach	Used Method	HBH/E2E	No Extra Buffers	Targeted Fault Model	Correction Latency
[4]	P-RET	E2E	✓	SETs + 50% of METs	≫ 1 clk
[64]	P-RET	E2E	✓	SETs	≫ 1 clk
[65]	P-RET	E2E	✓	SETs + METs	≫ 1 clk
[66]	P-RET	E2E	✗	SETs + METs	≫ 1 clk
[67]	P-RET	E2E	✗	SETs	≫ 1 clk
[68]	ECC	E2E	✓	SETs	N.A.
[69]	ECC	HBH	✗	SETs	N.A.
[70]	ECC	HBH	✗	SETs	N.A.
[71]	F-RET	HBH	✓	SETs	N.A.
[72]	F-RET	HBH	✗	SETs + METs	3 clks
[73]	RT + ECC + EDC	HBH	✗	SETs + METs (Prediction)	2 clks
[74]	P-RET	HBH	✗	SETs + Some DET	N.A.
[75]	ECC	HBH	✗	SETs + METs	N.A.
[76]	ECC	HBH	✗	SETs	≫ 1 clk
Proposed	**RT**	**HBH**	✓	**SETs + 50% of METs**	**0-1 clk**

In HBH mechanisms, the packet or flit is checked for errors every time data is transferred from one router to another (during every hop). In E2E mechanisms, on the other hand, errors in data can only be detected at the destination node. This is a serious shortcoming, since in case of errors in the data fields, which are used for routing (such as the flit type, destination address, packet length, etc.) E2E error checking can result in routing errors or even deadlocks. Therefore, even though, compared to HBH methods, E2E methods have less area and power overhead, they are only useful as a second layer of fault tolerance on a NoC-based system, which already includes a HBH fault tolerance mechanism.

An additional benefit of HBH mechanisms is that they have lower error repairing latency than E2E mechanisms, in which the entire packet needs to be retransmitted from the source to destination. In HBH mechanisms data needs to be retransmitted only from the previous router.

Four main types of approaches exist in the literature for handling soft errors in data on NoC links – data retransmission, error correction codes and relaxed transmission.

- **Data Retransmission:** Data retransmission approaches can be further divided into packet retransmission and flit retransmission.
  - **Packet Retransmission (P-RET):** Works such as [64–67] and Publication IV have proposed methods based on E2E packet retransmission. The main drawback of this approach is the high latency caused by retransmission. Additionally, as explained before, using E2E fault tolerance approaches, the correctness of the packet can only be evaluated at the destination node. This means that once a faulty packet is detected at the destination, often by using an Error Detecting Code (EDC), a negative acknowledgment (NACK) packet needs to be transmitted back to the sender, requesting a retransmission.

    During this process, the receiving node might also need to discard the non-faulty packets in order to preserve the packet order. Additionally, it is important to note that the E2E packet retransmission mechanisms take a long time to correct the errors and require distributed packet dropping mechanisms to be implemented in the NoC in order to prevent a network-wide failure caused by partially missing packets or corrupt routing information in the header flits. A distributed packet dropping approach, however, requires a fault detection unit implemented inside each router, which negates the advantages (lower area and power usage) of the E2E approaches.

On the other hand, [74] has proposed a HBH packet retransmission method which uses store-and-forward switching to overcome the problem with packet retransmission. However, the store-and-forward switching is a very inefficient since every packet needs to be stored in its entirety in every router before forwarding it. As a result, this approach requires very large input buffers in routers and limits the maximum packet size while also considerably reducing the throughput and increasing power usage.

  - **Flit retransmission (F-RET):** In contrast to packet retransmission, works such as [72] and [71] have proposed HBH flit retransmission. In these works, faults are detected every during every hop at a flit level. Therefore, when a faulty flit is received, only a single flit needs to be retransmitted. additional, the approaches introduced in these works provide much lower latency compared to E2E approaches but still need additional retransmission buffers in the router.

- **Error Correction Code (ECC):** Another class of fault tolerance mechanisms for NoC links are ECCs. Works such as [69, 70] and [76] have provided solutions for correcting single transient upsets, while [75] has used more sophisticated error correction techniques to cover multiple event transients. Like retransmission mechanisms, ECCs are also implemented either as HBH [69, 70, 75, 76] or as E2E [68]. However, the main problem with using error correction codes in NoCs is the large area and data overhead of such mechanisms.

- **Relaxed Transmission (RT):** In contrast to above-mentioned approaches, Relaxed Transmission uses the transient nature of the SETs for fault mitigation. The main concept of the RT approach is to pause the transmission until the fault disappears. The basic concept of such an approach has been proposed in [73]. However, they do not provide a design for the RT-based mechanism. Moreover, in the article, RT is proposed to be used in combination with ECC-based solutions to mitigate timing error and soft errors based on error prediction using machine learning and decision trees. Because SETs are random by nature, they cannot be predicted. However, if used in combination with a checker for detecting SET faults, the RT mechanism is good for correcting soft errors that are caused by SETs in the combinational part of the sending router with relatively low overhead and latency.

This chapter goes beyond the state of the art by presenting a design of an RT-based mechanism. The proposed mechanism allows to correct transmission errors that are caused by transient faults, such as SETs, in the output ports of NoC routers. The proposed mechanism has a very low error correction latency, the fault-free operation can be resumed at the next raising clock cycle after the transient fault has disappeared. Furthermore, the proposed mechanism can use the inherent redundancy in the input FIFOs of the NoC routers, hence removing the need for additional retransmission buffers that are needed by HBH retransmission mechanisms. This helps to reduce the usage of chip area and power.

3.3 The Effects of Soft Errors in Network-on-Chip Packets

On-chip networks are very sensitive to soft errors in the data that is transmitted over the inter-router links. A single erroneous value may lead to a network-wide failure. In this context, a network-wide failure is defined as the situation where the network is either completely or partially congested and the system cannot recover from it due to inability to route packets.

Errors can obviously occur in many parts of the packet. Different errors have different effect on the NoC:

- **An error in the flit type field.** A faulty flit type can lead to a network-wide congestion. An example of this case is when packet without a valid header flit that contains routing information is received.
- **An error in the destination address.** A wrong or even a non-valid destination address might lead to a network-wide congestion, since data cannot be successfully transmitted to the destination.
- **An error in other header information.** In case of the Bonfire network, errors in source address, packet length, etc. may cause problem at the application layer but have no impact on the network's behavior. Therefore, the effects of these faults can be ignored for the network layer. However, for other NoCs, which work differently (e.g., use routing tables instead of LBDR), an error in this information may also result in a failure.
- **An error in the payload data.** Faulty payload data does not impact routing. However, it might still cause a problem at the application layer. The effect of these faults can usually be ignored in the network layer.

In order to avoid network congestion and problems in application layer, a method for mitigating the faults is needed. Granted, most of the errors can be corrected using an error detection mechanism, such as the parity checker and a packet dropping mechanism. However, dropping and resending packets is expensive and should be avoided, if possible. Therefore, the RT mechanism was designed.

3.4 Architecture of the Relaxed Transmission Mechanism

The inner workings of the proposed RT mechanism can be investigated by analyzing a simplified concept of that mechanism, which can be seen in Figure 10. In the figure, only one of the output ports of the router that is transmitting data (*Sending Router*) and one of the input ports of the router that receives the data (*Receiving Router*) are shown. The modifications made to the NoC router, colored blue in Figure 10, are implemented in all the ports. In addition, for increased clarity, router components that are not necessary for explaining the concept, such as the flow control mechanism, are not included in the figure.

The RT mechanism is designed to detect errors that are caused by transient faults, such as SETs, in the combinational logic at the output of the *Sending Router*. The proposed mechanism can correct these faults with minimal time overhead. The SET detection is performed at the input of the *Receiving Router* using a parity checker.

While a parity checker requires adding only a single bit to every transferred flit, it does not offer any error correction, only detection. Therefore, the proposed RT mechanism includes additional techniques for correcting the parity errors. The simplified version of the proposed RT mechanism, seen in Figure 10, utilizes two additional registers. In

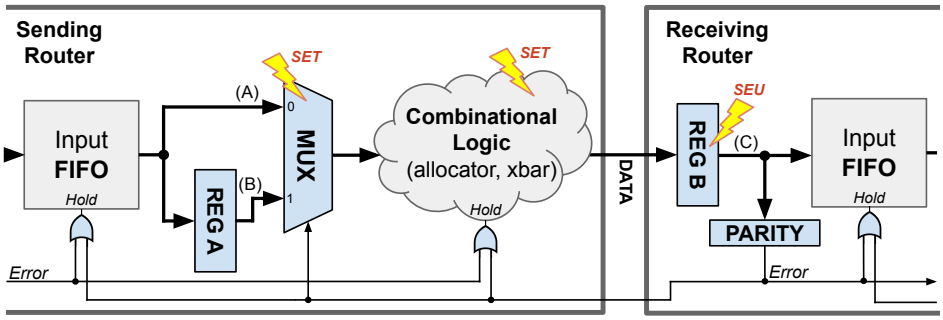


Figure 10: Simplified version of the proposed RT mechanism. The yellow lightning symbols represent the faults that can be corrected using the proposed mechanism.

the figure, data is transmitted from the *Sending Router* to the *Receiving Router* at maximum speed of one flit per clock cycle. The mechanisms behind flow control are explained in more detail in Section 2.2.3.

Every clock cycle, a flit is read from the *Sending Router's* input FIFO, then passed through the combinational logic that is responsible for routing the flit to the correct output port, before arriving at the *Receiving Router's* input FIFO. The main goal of the RT mechanism is to protect the combinational logic at the output of the *Sending Router* against SETs. As mentioned before, the SETs in the combinational logic are detected by the *Receiving Router* after the value is read into its input FIFO. However, since the *Receiving Router* detects the error during the next clock cycle, the data is not available anymore to the *Sending Router*. Therefore "re-sending" it after the fault caused by the SET has disappeared is not possible.

For this reason, as shown in Figure 10, in the *Sending Router*, every flit is transmitted to the *MUX's* first input over line *(A)* and simultaneously stored into register *REG A*. This results in the output of *REG A* (and, consequently, at the second input of the *MUX*) always having the value that was read from the FIFO at the previous rising clock edge. During normal operation, the data at *MUX's* input *(A)* is transferred the *Receiving Router*. However, when a parity error is detected, the *MUX* is switched to the *(B)* input, which is connected to *REG A)*. Combined with pausing the routers, this results in the previously read value being kept at the input of the combinational logic.

In the *Receiving Router*, as seen in Figure 10, every time a flit is received over the *DATA* link, it is stored into a register *REG B*. The value in the register is checked using a parity checker. If a parity error is detected, it will cause the transfer to be paused by triggering the *Hold* signal. The Hold signal going high will result in the following:

- The read and write pointers of the input FIFOs of both the *Sending Router* and the *Receiving Router* are frozen, effectively disabling the FIFOs
- Credit counter in the *Sending Router* is frozen.
- Allocator will stop giving grants.
- Last, but not least, *MUX* will be switched to input *(B)*. This results in holding the current flit on the *DATA* line.

When the system is paused, the *DATA* line is sampled to *REG B* every clock cycle and the data on that line is re-checked by the parity checker. Once the effects of the SET have disappeared and the parity checker does not detect a parity error anymore, normal operation will be resumed by releasing the *Hold* signal. This will cause the routers to un-pause and to switch the *MUX* back to input *(A)*. Because the data in register *REG B* is updated and checked each clock cycle, the mechanism is guaranteed to return to normal operation on the next rising edge of the clock signal after the fault has disappeared.

As visualized in Figure 10, the proposed mechanism is designed to handle SET faults in the combinational logic at the output of the *Sending Router*. However, it is also capable of detecting and handling SETs in the *MUX* and SEUs occurring in *REG B*, which are added by the RT mechanism.

3.5 Optimized Relaxed Transmission Mechanism

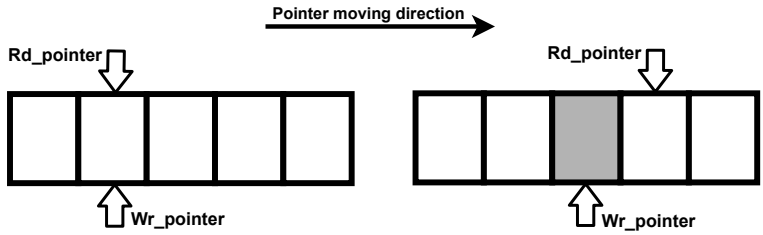


Figure 11: Circular buffer implementation of a FIFO

Since the register *REG B* in Figure 10 can be thought of as an extension to the *Receiving Router's* FIFO, the design can be optimized by using data written to the FIFO one clock cycle earlier as input to the parity checker, thus removing the need for *REG B* as a separate register. This can be easily implemented, assuming a circular buffer implementation of the FIFO, as shown in Figure 11.

In a circular buffer implementation of a FIFO, the buffer is accessed by using two pointers: one for reading ($Rd\_pointer$) the FIFO and the other one for writing ($Wr\_pointer$) into the buffer. If the buffer is empty, the read and write pointers both point at the same memory slot, as seen in Figure 11 (a). After each writing operation, the write pointer is shifted by one slot to the left. It can be seen that *REG B* can be implemented in such a way that parity checker is connected to the FIFO slot referred to by $Wr\_pointer - 1$.

The data is read from the memory slot referred by the read pointer. However, as it can be seen in Figure 11 (b), the condition for the FIFO being full is when the read buffer is ahead of the write buffer by one memory slot ($Wr\_pointer = Rd\_pointer - 1$). Writing would cause the write pointer to move to the same memory slot where the read pointer is, creating the "empty" state. Therefore, in such an implementation, the memory slot $Rd\_pointer - 1$ can be never overwritten. This definition helps to further optimize the RT mechanism, since *REG A*, as mentioned before, stores the flit that was previously read from the *Sending Router's* input FIFO during the previous rising clock edge. Therefore, the flit in *REG A* is the same as the one stored in FIFO's memory slot $Rd\_pointer - 1$.

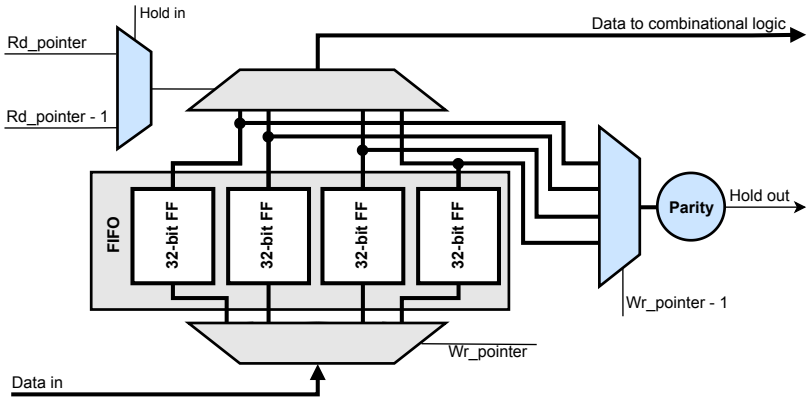


Figure 12: FIFO implementation for the optimized version of the proposed RT mechanism that utilizes the built-in redundancy in int circular buffer implementation of the input FIFO

The proposed RT mechanism takes advantage of the optimizations in order to provide SET fault tolerance for NoC links by protecting the combinational logic at the *Sending Router's* output. The implementation of the RT-protected FIFO in can be seen in Figure 12, where the components added by the proposed mechanism are shown in blue. In the figure, data is normally read from the FIFO's memory slot referred by $Rd\_pointer$ and written into the slot referred by $Wr\_pointer$. However, *REG B* has been implemented by connecting the parity checker to an additional multiplexer, which uses the previous write pointer as its select line. If an error is detected by the parity checker, the *error* output is activated. As discussed earlier, this will cause the sending router's operation to be paused until the fault has disappeared. The register designated as *REG A* in Figure 10 is implemented as reading of the memory slot referred by $Rd\_pointer - 1$, thus also removing the need for *REG A*. The selection between reading at position pointed to by $Rd\_pointer$ and $Rd\_pointer - 1$ is performed using a multiplexer with the parity checker's output used as the select line.

3.6 Experimental Results

The experiments presented in this section demonstrate SET mitigation capabilities of the proposed RT mechanism and provide information regarding the overheads of the proposed mechanism in terms chip area, power and critical path delay. All results, except for latency, were acquired by synthesizing the design using TSMC 40 nm CMOS technology standard cell library in Synopsys Design Compiler. The experiments were performed at 400 MHz clock frequency. The latency results were obtained using simulation.

The RT mechanism was implemented for the Bonfire (see Section 2.2.5) NoC routers in a 4×4 2D-mesh NoC. All experimental results are compared to a baseline Bonfire router in the same configuration. The baseline router does not include any fault tolerance mechanisms.

The feasibility of the RT mechanism was tested by measuring its capability correcting the errors caused by the SETs in the *Sending Router's* combinational part. The results were acquired by running a set of random uniform SET fault injection experiments by forcing the signals in Modelsim simulation at register transfer level (RTL) to random

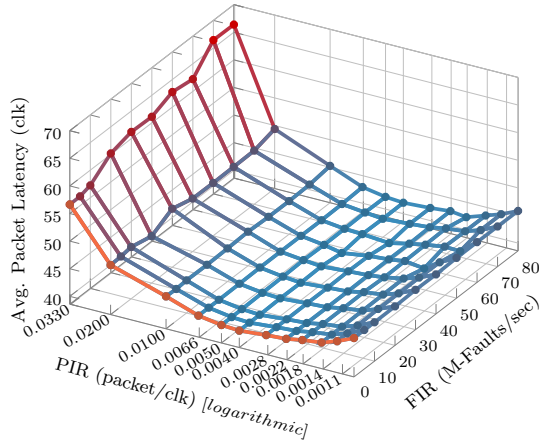


Figure 13: Average packet latency of a 4×4 2D-mesh network under different packet (PIR) and fault injection rates (FIR) on the links with fault duration of 10% of the clock cycle. Orange line represents fault-free behavior of the RT mechanism

values. The experiment was set up in the way that all injected faults resulted in a parity error in the *Receiving Router*. This approach to fault injection is sufficient because the goal of the experiment is to verify that the proposed mechanism can correct all injected faults and to measure the latency overhead, which is introduced to the system by fault correction. The probability of individual SETs propagating to the output is out of the scope of this work.

The proposed mechanism was, compared to fault-free experiments, able to correct 100% of the injected faults with almost no additional latency. The length of each SETs was set to 250 ps (10% of the clock period). The experiments were run with different network loads (PIR) and for fault rates (FIR).

As it can be seen in Figure 13, all injected faults were corrected successfully with almost no increase in latency, even at unrealistically high fault rates of up to 80 million faults per second. A slight increase in the latency is only visible in case of a very high PIR when the traffic in the NoC is already saturated.

Table 2: Area overhead of the proposed mechanism

	Area (μm^2)	**Area Overhead**
Baseline router	8276.45	–
Baseline with RT	9777.26	18%

Table 3: Critical path delay overhead of the proposed mechanism

	Critical Path Delay (ns)	**Overhead**
Baseline router	2.28	–
Baseline with RT	2.46	7.8%

Table 2 shows that the proposed RT mechanism imposes 18% area overhead to the Baseline router. As seen in Table 3, the critical path delay is increased by 7.8%.

The results were acquired on TSMC 40 nm complementary metal–oxide–semiconductor (CMOS) technology standard cell library in Synopsys Design Compiler. The synthesized RT router had the proposed mechanism implemented for all five input and output ports.

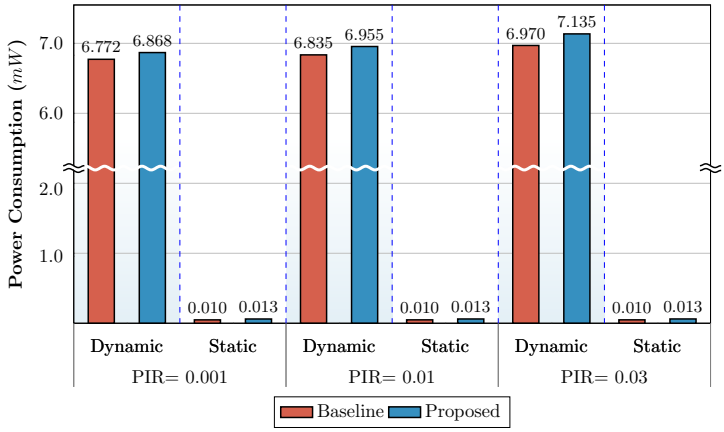


Figure 14: Comparison of dynamic and static power consumption (in mW) of the proposed and baseline 4×4 network under different packet injection rates

Finally, Figure 14 illustrates dynamic and static power consumption of the proposed mechanism. Results are compared to the baseline router under different packet injection rates. As it can be seen in the figure, the increase in power usage implied by the RT mechanism is minimal.

3.7 Chapter Conclusions

In this chapter, a novel RT-based mechanism was proposed for mitigating soft errors at NoC links. These errors are caused by transient faults, such as SETs, in the combinational circuits at the output ports of the NoC routers. The proposed mechanism uses parity checker for error detection. When a parity error is detected, the routers involved in the data transfer are paused and the same data will be re-transmitted until the transient fault disappears. Unlike re-transmission-based approaches, which usually have higher time overhead that is caused by the re-transmission protocol, the proposed mechanism can continue fault free operation at the next rising clock edge after the fault disappears.

Additionally, an optimized version of the proposed algorithm was presented that uses the inherent redundancy that is found in the input buffers of NoC routers. Therefore, the proposed mechanism does not need additional re-transmission registers that are usually needed by HBH re-transmission approaches.

The optimized mechanism has 18% area and 7.8% critical path delay overhead when compared to baseline router that does not include fault tolerance mechanisms. Experimental results show the effectiveness of the mechanism even in unrealistically extreme fault rates of up to 80 million faults per second with minimal additional latency compared to fault-free runs under the same traffic scenarios.

4 Comprehensive Performance and Robustness Analysis of 2D Turn Models for Network-on-Chips

In this chapter, for the first time, all 256 uniform TMs for 2D full-mesh NoCs have been enumerated and thoroughly analyzed. As the result of the analysis, 50 deadlock free TMs that provide a full connectivity (usable TMs) were found. In contrast, as explained in Section 4.2, only a limited number of usable TMs have been described previously in the literature. The usable TMs were evaluated even further to extract their figures of merit for adaptivity, extended adaptivity, average connectivity and latency. The analysis resulted in identification of many previously unknown TMs with good figures of merit.

This chapter is based on the following publications:

- **III**
  S. P. Azad, B. Niazmand, K. Janson, T. Kogge, J. Raik, G. Jervan, and T. Hollstein, "Comprehensive performance and robustness analysis of 2D turn models for Network-on-Chips," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May 2017

- **IV**
  S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in Network-on-Chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 48–53, April 2017

4.1 Introduction

In the previous chapter, a solution for mitigating transient faults on NoC links was provided. However, transient faults are not the only problem that NoCs routers face. Permanent faults can occur due to manufacturing defects or during the device's lifetime as a result of physical damage or aging. Permanent faults cannot be repaired without either physically replacing the failed components or without using redundant spare components.

Fortunately, in NoCs that utilize the full mesh network topology, there exists built-in redundancy that can be taken advantage of for handling permanent faults. Specifically, in a full mesh network, there are multiple paths between all nodes. As such, it is often possible to change the routing algorithm to bypass a broken link in the network and, therefore, to restore connectivity between the nodes. In this context, a faulty link refers to any fault in the router, which renders an output port inoperable or corrupts the data passing through that port. Since the precise location of the fault in the router is not important in the context of this chapter, link faults are a good abstraction for different communication-related problems in that can occur in NoC routers.

In case of full-mesh topology, routing in NoCs is often implemented using turn model-based routing. The concepts of TM-based routing were explained in more detail in Section 2.2.1. However, not all TMs are equally good for reliable communication. The goal of this chapter is to identify the TMs that can improve system's reliability while having minimal overhead.

In this chapter, all 256 uniform 2D TMs are thoroughly analyzed and all usable TMs that provide full connectivity and are deadlock free, are extracted. The results of this analysis help to make an informed decision while choosing the best TM for a certain fault scenario. For this reason, this chapter introduces two new reliability metrics for the assessment of TMs.

First, an equation for calculating the connectivity that is provided by the TM is proposed. Connectivity is defined as the TM's capability to ensure that each node can communicate with every other node in the NoC. If this is true, it is said that the TM can provide full connectivity in the NoC. This is important for identifying usable turn models (TMs that provide full connectivity while avoiding deadlock). Additionally, the connectivity metric can be used for checking if a TM can bypass link errors. If a TM still provides full connectivity in a NoC with a failed link, it means that switching to this TM would restore full functionality of the NoC.

Secondly, the Extended Degree of Adaptivity (DoA_{ex}) metric is introduced. The DoA_{ex} metric is an extension to the Degree of Adaptivity metric (DoA) that has been previously described in the literature [63]. The new DoA_{ex} metric describes the number of alternative paths a packet can take under TM in case of non-minimal path routing. In contrast, the previously published DoA metric only considered minimal path routing. Under both metrics, a TM that can provide a higher Degree of Adaptivity, usually also results in better fault tolerance.

In addition, the latency and average connectivity of TMs is investigated. Average connectivity is a metric that measures the connectivity for certain number of faults in the NoC, averaging the connectivity values over different fault configurations with the same number of link failures. Average connectivity is a metric, which is directly connected to the reliability of the TM. The data shows that there is a strong correlation between TMs average connectivity, the latency caused by the TM, and its adaptivity.

Finally, insights into the process of the TM reconfiguration, including a simple algorithm for identifying the most suitable TM for any fault configuration.

The rest of this chapter is structured as follows. In Section 4.2, related works are introduced. This is followed by Section 4.3, where an overview of the method used for guaranteeing deadlock-freeness of TMs is provided. Then, usable TMs are extracted in Section 4.4. Further analysis of the usable TMs in terms of adaptivity is performed in Section 4.5. Next, in Section 4.6, the algorithm for calculation of TMs' average connectivity is provided. Subsequently, in Section 19, the latency of all TMs is evaluated. This is followed by Section 4.8, where an algorithm for finding a best TM for any fault scenario is proposed. Finally, Section 4.9 concludes this chapter.

4.2 Literature Review

Many approaches exist for handling permanent faults on NoC links. Some works, such as [77–81], implement fault tolerant routing by adding (multiple) virtual channels. However, the experiments on our in-house Bonfire NoC router (described in Section 2.2) show that the input buffers account to $\approx 72\%$ of the chip area used by the router. Such approaches are very costly, since implementation of virtual channels causes the size of the input buffers to be doubled for each added virtual channel.

Other works present routing algorithms, which are designed to handle broken links. For example, [82] uses non-adaptive XY routing algorithm [62]. However, for bypassing faulty links, it utilizes a custom algorithm and routing matrix that is stored inside the routers. On the other hand, works like [83] and [84] combine XY and YX [85] using custom algorithms. Additionally, works such as [86] and [87] rely on reconfiguration of NoC routers to guarantee fault-free functioning.

The main problem with the approaches listed above is their overhead. In practice, the NoC should be as small as possible and use minimal amount of power. For this reason, the TM-based routing algorithms [62] (see Section 2.2) are an elegant solution and are often used in NoCs. This approach allows to leave more available chip area for useful functionality that is implemented by the processing elements and also reduce the idle power consumption of the system.

TM-based routing algorithms in NoCs can be classified as either deterministic or adaptive [88]. Deterministic routing algorithms use a single path for each source-destination pair, whereas adaptive routing provides more path diversity. In case of adaptive routing, alternative paths in the NoC could be chosen based on a criteria such as traffic on the links, systems' fault status, etc. Therefore, deterministic algorithms provide guaranteed arrival times for data and may be preferred for real time applications, while adaptive algorithms can be more reliable, if implemented correctly.

In literature, in addition to the two deterministic TMs (XY [62] and YX [85]), seven adaptive turn models are introduced. More specifically, three deadlock free TMs are introduced in [63] – West-First, North-Last and Negative-First. Additionally, [89] introduces the North-First, South-First and Restricted North First TMs. Finally, the East-First TM is addressed in [90].

However, to the best of the author's knowledge, prior to this work, a formal method for evaluation of all uniform TMs in terms of reliability has been missing in the literature. Moreover, while there exists a finite number of 2D uniform TMs, previous works, such as [91], have only covered performance comparison of a few well-known TMs.

4.3 Evaluation of Turn Models for Deadlock Freeness

A very important factor to be considered when choosing a routing algorithm is its deadlock freeness. Deadlock occurs when a cyclic dependency is created between the packets in a NoC. In this case, packets might end up waiting for available resources held by other packets in cyclic manner [88].

Two main approaches exist for addressing deadlocks: deadlock avoidance and deadlock recovery. In approaches utilizing deadlock recovery, deadlock is allowed to occur, but it is handled using a deadlock recovery mechanism. However, the focus of this chapter is on deadlock avoidance since it usually guarantees better performance. This is because in deadlock avoidance approaches deadlocks cannot occur at all due to inherent properties of the system. Therefore, deadlock avoidance mechanisms do not need additional time for recovering from deadlocks.

A similar phenomenon to deadlocks in NoCs is the livelock. In case of livelock, the packets are, unlike in deadlock, moving. However, they are constantly routed in a cyclic manner. The approach used in this chapter concentrates on deadlock avoidance by making cyclic routing in the system impossible. However, by making cycling routing of packets completely impossible, the approach guarantees the avoidance of both the deadlock and livelock.

In this chapter, TM-based routing is used for deadlock avoidance in 2D NoCs with full-mesh topology. It was first introduced in [63]. A TM is made of eight *turns*, that can be either enabled or disabled. A turn in a TM is defined as a change of direction in a packet's path. Directions are named based on cardinal directions: North (N), East (E), West (W) and South (S). In a 2D Mesh network, maximum of eight turn exist: N2E (North-to-East), N2W, E2N, E2S, W2N, W2S, S2E and S2W. For instance, S2E (↱) indicates a turn that, if it is allowed, enables a packet coming from the South input port of the router be forwarded to the East output port. Under a TM, each of the eight turn can be either allowed or disallowed. A total number of $2^8 = 256$ uniform 2D TMs can be derived from eight possible turns. In a uniform TM, that this chapter concentrates on, all routers in the NoC have the same turns disallowed, and the same turns allowed. A TM does not concern the connections going straight to the router, where the transmission axis is not changed (such as N2S or E2W) or connections from and to the local PE connected to the local port of the router. Those connection are always enabled under every TM. The concept of TMs is explained in more detail in Section 2.2

As mentioned above, guaranteeing deadlock freeness in NoCs means guaranteeing that cyclic routing is impossible in the NoC. This problem can be solved using graph theory. In [92], the concept of Cyclic Dependency Graph (CDG) is used for deadlock detection. In a CDG, the nodes represent the network channels and edges denote the channels' dependencies. It is proven that the CDG must be acyclic to guarantee deadlock-freeness. In this work a similar approach, the Routing Graph (RG), is used instead of the CDG. The concept of routing graphs was introduced in [93]. The rest of this section gives an overview on the construction of routing graphs and describes the process of evaluating deadlock freeness of TMs using the routing graphs.

4.3.1 Routing Graph

A Routing Graph, RG(V, E), is a directed graph, in which the set of vertices (V) denotes the set of all input and output ports in the network (two nodes per port) and the set of edges (E) represents the set of (v_i, v_j), where v_i is a vertex (input or output port) that depends on the v_j port. For the sake of simplicity, a vertex v in RG is denoted as $node_{i,p,dir}$, which describes direction $dir \in \{in, out\}$ of port $p \in \{N, E, W, S, L\}$ of node i in the network.

There exist two types of links, represented using edges in the routing graph:

- **Inter-router edges**, representing connections (links) between routers, connecting an output port of a router to an input port of an adjacent router.
- **Intra-router edges**, representing allowed connections inside the router, connecting an input port of a router to an output port in the same router. An intra-router link can be:
  - *A connection from or to local port.* These connections represent dependencies between the router's North, East, West and South ports and the local port connected to the PE.

– *A straight connection.* The straight connections describe dependencies between ports involved in maintaining straight connections inside the router (e.g., from West input to East output port of a router).
– *A turn.* A turn is a dependency of the ports in perpendicular direction, where the packets move from the X axis of the mesh to the Y axis, or vice-versa (e.g., from East input to South output port). See Section 2.2.1 for more info on turns.

Figure 15 shows an example of an RG for XY routing in a 3×3 2D full-mesh network, corresponding to Figure 16. For a routing algorithm to be deadlock free, its corresponding RG must be acyclic.

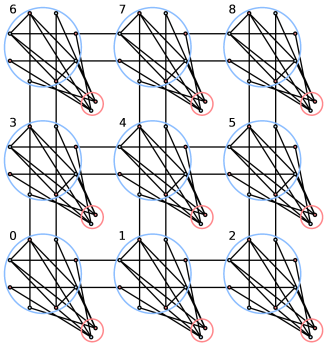


Figure 15: Routing graph for the XY routing algorithm

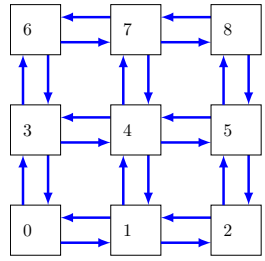


Figure 16: 3×3 full-mesh NoC

4.3.2 Proof of Deadlock Freeness

Theorem 1. *A deadlock in a turn model results in a cycle in the RG derived from the turn model.*

Proof. Let us assume that a deadlock in the turn model results in an RG without any cycles. An RG represents the sequence of all dependencies between the inputs and outputs of the routers, under the applied turn model. If there are no cycles in the RG, there also cannot be cyclic dependencies between the inputs and outputs of routers in the network. Hence, no deadlock can be formed. This is in contradiction with our initial assumption of having a deadlock for the turn model. It proves that a deadlock in a turn model results in a cycle in the RG derived from the turn model. □

Using this method, it is possible to discard 35 turn models with deadlocks out of the total 256 TMs, which leaves 221 deadlock-free turn models.

4.4 Identification of Usable Turn Models

A *usable* TM needs to not only to be deadlock-free, but also provide connectivity between all node pairs in the network. However, out of 221 deadlock-free TMs, some provide only partial connectivity. The obvious examples of this case are the one TM with zero enabled turns and 8 TMs that only have one enabled turn. To evaluate the connectivity of TMs, the following equation can be used:

$$Connectivity_{RG} = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C_{i,j,RG} \tag{1}$$

In Equation 1, N is the total number of nodes in the network. RG represents the routing graph of a full-mesh network using a TM-based routing algorithm. $C_{i,j,RG}$ indicates if there is a connection between nodes i and j under the RG:

$$C_{i,j,RG} = \begin{cases} 1 & \text{if a path exists from } node_{i,L,out} \text{ to } node_{j,L,in} \\ & \text{in RG where } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

A routing algorithm can use either minimal path routing (only shortest path in the RG), or non-minimal path routing (all simple paths in the routing graph). The maximum value for the $Connectivity_{RG}$ for a certain size of network can be found by counting all network pairs (setting $C_{i,j,RG}$ to constant one in Equation 1). Afterwards, the connectivity of TMs can be assessed by using the unmodified Equation 1, and replacing RG with the RG representing the TM that is being tested. For example, for a 3×3 mesh, the maximum $Connectivity_{RG}$ is 72. Any TM that results in a lower $Connectivity_{RG}$ does not provide full connectivity, and as such, is not usable.

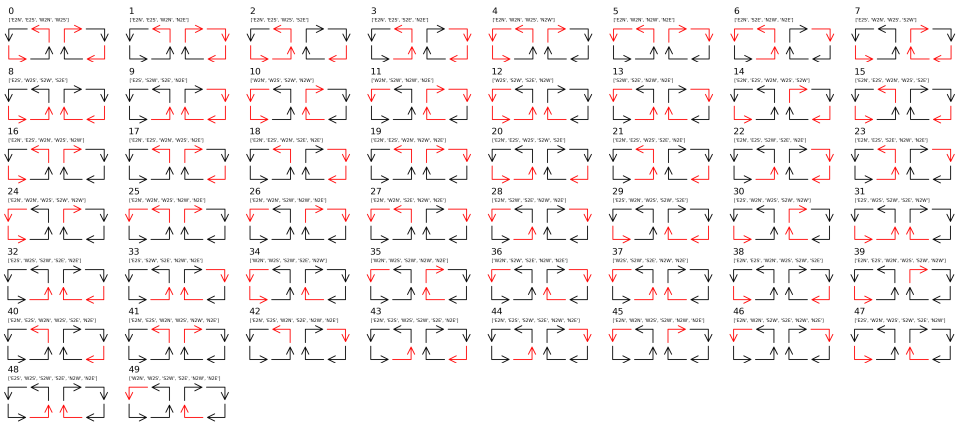


Figure 17: Visualization of all 50 usable uniform 2D TMs, deadlock free TMs that provide full connectivity. The forbidden turns are drawn in red.

Out of the 221 deadlock-free TMs, only 50 TMs are usable, they are deadlock-free and provide full connectivity. These TMs are visualized in Figure 17. There are 14 four-turn usable TMs, 24 five-turn and 12 six-turn usable TMs. For comparison, Table 4 lists all the TMs which are, to the author's best knowledge, previously named and addressed in the literature.

Table 4: List of TMs that have previously been described in the literature

#	Allowed turns	Conventional Name
0	E2N, E2S, W2N, W2S	XY [62]
13	S2W, S2E, N2W, N2E	YX [85]
33	E2S, S2W, S2E, N2W, N2E	Restricted North First [89]
39	E2N, E2S, W2N, W2S, S2W, N2W	East-First [90]
40	E2N, E2S, W2N, W2S, S2E, N2E	West-First [63]
41	E2N, E2S, W2N, W2S, N2W, N2E	North-Last [63]
42	E2N, E2S, W2N, S2E, N2W, N2E	Negative-First [63]
46	E2N, W2N, S2W, S2E, N2W, N2E	South-First [89]
48	E2S, W2S, S2W, S2E, N2W, N2E	North-First [89]

4.5 Metrics for Adaptivity

In order to classify usable TMs, the Degree of Adaptivity (DoA) metric introduced in [63], can be used. The DoA metric, which considers the shortest paths from the source node to the destination node in the RG. A general form of DoA metric can be formulated as shown in Equation 2:

$$DoA = \frac{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} NoSP_{i,j,rg}}{\text{number of node pairs in NoC}} \tag{2}$$

Where N is the number of nodes in the network. $NoSP_{i,j}$ is equal the number of shortest paths in the RG from node i to node j if $i \neq j$ and equal to 0 when $i = j$:

$$NoSP_{i,j,RG} = \begin{cases} \text{num. of shortest paths in } RG \text{ from } node_{i,L,out} \text{ to } node_{j,L,in} & i \neq j \\ 0 & \text{otherwise} \end{cases}$$

Table 5: DoA and DoA_{Ex} for all usable 2D routing algorithms. Previously known TMs are underlined

	4 turns			5 turns		6 turns	
TM Num	<u>0</u>, <u>13</u>	3, 5, 8, 10	1, 2, 4, 6, 7, 9, 11, 12	14, 15, 16, 17, 28, <u>33</u>, 36, 37	18-27, 29-32, 34, 35	<u>42</u>, 43, 45, 47	38, <u>39</u>, <u>40</u>, <u>41</u>, 44, <u>46</u>, <u>48</u>, 49
DoA	1			1.23		1.43	
DoA_{Ex}	1	1.41	1.63	2.11	2.41	3.83	4.33

However, the DoA metric only considers the shortest paths in the network. As a result, it does not allow the full adaptivity potential of non-minimal path routing. Non-minimal path routing provides more paths than minimal path routing, by allowing all simple paths (paths that do not have repeating nodes in them, and thus, no cycles) to be taken in the routing graph. For this reason, in this work, an extension to the DoA metric is proposed that includes all the simple paths in the network. The new DoA_{Ex} metric is described in Equation 3.

$$DoA_{Ex} = \frac{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} NoSP'_{i,j,RG}}{\text{number of node pairs in NoC}} \tag{3}$$

Where N is the number of nodes in the network. $NoSP'_{i,j}$ is equal the number of simple paths (paths without cycles) in the RG from node i to node j if $i \neq j$ and equal to 0 when $i = j$:

$$NoSP'_{i,j,RG} = \begin{cases} \text{num. of simple paths in } RG \text{ from } node_{i,L,out} \text{ to } node_{j,L,in} & i \neq j \\ 0 & \text{otherwise} \end{cases}$$

The resulting DoA and DoA_{Ex} values for all usable TMs are presented in Table 5. In the table, the previously known TMs are underlined. It is no surprise that TMs with higher number of turns also result in higher adaptivity, since they provide more freedom of routing. The data in Table 5 shows that in each class of TMs (TMs with four, five, and six enabled turns), there are sub-classes that have different characteristics. For example, TM no. 3 shares two turns with XY and two turns with YX routing, which allows it to have non-minimal de-routes. Similarly, under non-minimal path routing, TM no. 1 and no. 2 have even further advantage in providing path diversity.

Additionally, many new TMs with very high adaptivity were found. For example, TMs no. 38, no. 44 and no. 49 belong all into the group with the highest adaptivity. Yet, to the best knowledge of the author, they have not been described in the literature prior to this work.

4.6 Average Connectivity Evaluation

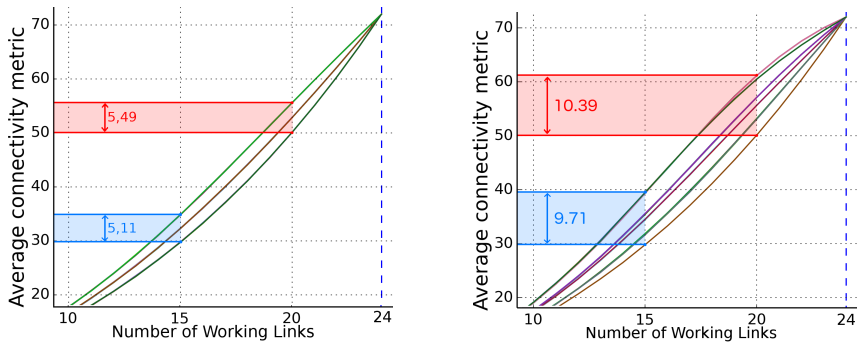
Algorithm 1 Algorithm for calculating the average connectivity of a TM

```
avgConnectivity = list()

for LnksBrokenCnt in range(0, LnksTotalCnt) do
    brokenConfs = [list of all 3×3 2D meshes with LnksBrokenCnt broken links]
    sumConns = 0

    // Calculate average connectivity over all configurations
    // with len(brokenConfs) broken links
    for all brokenConf in brokenConfs do
        RG = generateRG(turnModel, brokenConf)
        sumConns += connectivity(RG)

    avgConnectivity[LinksBrokenCnt] = sumConns / len(brokenConfs)
return avgConnectivity
```

In this section, the concept of TM's average connectivity is explained. Average connectivity of a TM for k faulty links is defined as the average of the connectivity values, calculated using Equation 1, taken over all possible fault configurations with k failed links. Average connectivity is the direct representation of the TM's resilience to link faults.

As such, experiments were conducted to evaluate the average connectivity of each TM. By using the connectivity metric introduced in Section 4.4, average connectivity of the network with `LnksBrokenCnt` permanently broken links (out of total number of `LnksTotalCnt` inter-router links), can be calculated using Algorithm 1. It is important to note that this algorithm is only used for offline TM evaluation and it is not used for runtime fault correction.

In Algorithm 1 `avgConnectivity` is a list consisting of average connectivities. Each element `avgConnectivity[k]` of this list is average connectivities for a 3×3 network with k broken links. Each element `avgConnectivity[k]`, is calculated by averaging over the connectivity values for each fault configuration with a set number of faults.



(a) Minimal path routing

(b) Non-minimal path routing

Figure 18: Comparison of average connectivity metric of TMs by number of available links, using minimal and non-minimal path routing

By calculating the average connectivity for all TMs, for all fault calculations, the resultant data can be plotted into a graph. This results in Figure 18, which illustrates the difference between average average connectivities for all usable TMs that are listed in Table 5.

Figure 18a shows the average connectivity metric of the TMs under minimal path routing for different number of working links. Figure 18b depicts the same data under non-minimal path routing. Under minimal path routing, the three lines in the figure correspond directly to the three classes of DoA in Table 5, where TMs with higher DoA provide better connectivity. However, the gap between the worst-performing and the best-performing groups of TMs is not substantial. In case of non-minimal path routing, the curves diverge more. The lines correspond to the seven classes of TMs, grouped by DoA_{Ex} in Table 5. The gap between the TMs with the highest DoA_{Ex} and the ones with the lowest DoA_{Ex} is close to twice as large as it was for DoA and minimal path routing. Additionally, comparing of the two figures shows, as expected, that under non-minimal path routing, the average connectivity is generally higher than in case of minimal path routing.

It can be seen from Figure 18 that there is a high correlation between the average connectivity and the DoA / DoA_{ex} of the TMs. This makes sense, since the more adaptive the TM is, the more alternative paths exist under that TM. Using a TM with larger number of alternative paths also increases the probability of the TM being able to bypass the broken link. Additionally, TMs, which have the same number of alternative paths (share the same adaptivity rating), have also the same average connectivity, since on average they can bypass the same number of broken links.

Unfortunately, Algorithm 4.6 does not scale very well in terms of performance, since the number of combinations of possible broken links increases exponentially with the total number of nodes in the NoC. This makes assessing the reliability of the TMs using Algorithm 4.6 very time consuming for large NoCs. However, the high correlation between adaptiveness and average connectivity of TMs means that because a TM with higher adaptivity value results in higher average connectivity, running this algorithm is not necessary during reconfiguration. The choice can be made by just calculating the connectivity under the current fault scenario for the TMs with the highest adaptivity.

4.7 Latency Evaluation

In this section, all the 50 usable TMs shown in Figure 17 are evaluated under synthetic traffic patterns using the Noxim [94] NoC simulator. The experimentation setup parameters are set as follows: A 4×4 2D full mesh NoC utilizing the Bonfire NoC routers introduced in Section 2.2 of this thesis. The system clock frequency is set to 1 GHz for all routers. During the experiments, synthetic random uniform traffic pattern is used. Packets are generated using Poisson distribution. The length of the packets is fixed to 8 flits, and the FIFO input buffers in the routers are configured as is 4 flits deep. Additionally, before the start of the experiments, a warm-up time of length 1000 cycles was used to allow the traffic patterns of the NoC to stabilize. The simulation was run for up to 20000 cycles.

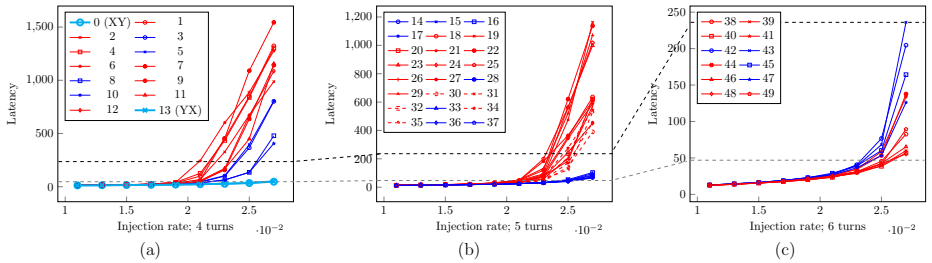


Figure 19: Latency results under random uniform traffic for (a) four, (b) five, (c) six-turn TMs

The average latency results are grouped based on the number of allowed turns to the groups of 4-turn, 5-turn and 6-turn TMs. Figure 19a-c show the average latency results for these TMs under random uniform traffic pattern (with packet injection rate ranging from 0.001 to 0.025). The curves are color coded in each figure to distinguish different classes of DoA_{Ex} (see Table 5).

The dotted lines in Figure 19a-c indicate the corresponding highest value of the Figure 19c and lowest value of Figure 19a, since the range of axes is different between Figure 19a-c.

It can be observed in Figure 19a that two of the TMs (0 and 13, highlighted with thick cyan colored line) that correspond to XY and YX routing, outperform the other TMs in terms of average latency. This conforms to the observations made in [95]. After those two TMs, both classes of 6-turn TMs and 5-turn TMs with lower DoA_{Ex} perform better than others.

4.8 Implications on Permanent Fault Tolerance in NoCs

4.8.1 Overview of the Routing Algorithm Reconfiguration Process

When a link in the NoC breaks due to a permanent fault, it can render a node, or even an entire region of the NoC isolated from rest of the network. This can have serious consequences, possibly leading to even the failure of the entire system. However, fortunately, in full-mesh NoCs, there exists built-in redundancy. Each node in the NoC is connected to at least two neighbors. Most nodes have three or even four connections. However, to take advantage of this redundancy, it is needed to have a suitable routing algorithm, which can route packages past the broken link.

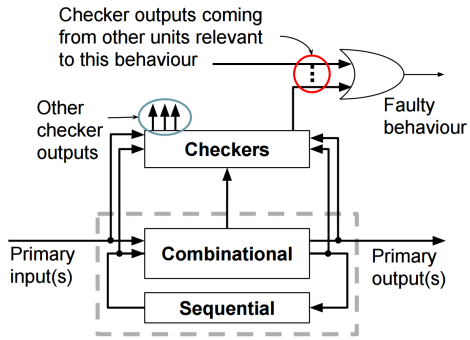


Figure 20: General structure of the fault detection, grouping and classification mechanisms for fault in control circuits

The broken links can be detected using checkers built into the router. Faults in the control part can be detected, for example, by using concurrent online checkers [55, 96] (Figure 20). This is described in more detail in [96]. Faults in the data-path can be detected using a parity checker, such as the one used in Chapter 3. Permanent faults can be distinguished from transient faults, like those caused by SETs, using fault classifier FSMs. [97, 98]. This topic is discussed in more detail in Publication IV.

Moreover, fault outputs from different checkers can be combined and abstracted into a turn fault, a fault occurring in one of the components on the path from an input port to an output port of the router (e.g., a West to North turn fault or even a straight path). This process is also illustrated in Figure 20. Additionally, it is important to understand that a single fault can also result in multiple turn faults. For instance, if a permanent fault occurs in the xbar, an entire output port will become non-functional. In this case, all turns that lead to that port will be affected. For example, if a West output port fails, both turns the S2W and N2W and the straight connection from East to West and the connection from the local port to West will fail. This information can information can be passed to the System Health Monitoring unit (SHMU), which can initiate reconfiguration of the routing algorithm based on the fault information received from checkers.

When the SHMU finds a TM that can restore communication to all nodes in the NoC by bypassing the broken link(s), it will perform the reconfiguration of the routing algorithm. To avoid deadlocks, the network needs to be either first cleared of all traffic, or a special algorithm for updating the routing algorithm at runtime, such as OSR Lite [99], needs to be used.

4.8.2 Choosing the Best Turn Model for a Fault Situation

Algorithm 2 Algorithm for choosing the best TM for the current fault situation

```
1: // Data structure for storing the TM with best connectivity
2: bestConnectivity = struct{connectivity=0, TM=None}
3:
4: // Read the fault information from checkers
5: currentFaultConfig = checkers.getFaultInfo()
6:
7: // The TMs with the highest DoA_ex should be checked first
8: sortedTMs = sort(usableSixTurnTMs, basedOn=DoA_ex)
9:
10: for TM in sortedTMs do
11:     RG = generateRG(TM, currentFaultConfig)
12:
13:     // Check if the TM has full connectivity under
14:     // current fault situation using Equation 1
15:     connectivity = RG.getConnectivity()
16:
17:     if (connectivity == FULL_CONNECTIVITY) then
18:         return TM
19:
20:     else if (connectivity > bestConnectivity.connectivity) then
21:         // If the TM does not provide full connectivity, save its connectivity value
22:         bestConnectivity.connectivity = connectivity
23:         bestConnectivity.TM = TM
24:
25: // No TM provides full connectivity. Select the TM with best connectivity
26: return bestConnectivity.TM
```

The information provided in this chapter is crucial for the SHMU to select the most suitable TM for the current fault situation. The most suitable TM can be chosen using the algorithm that is detailed in Algorithm 2. The algorithm is run whenever a new fault is detected by the checkers.

First, the fault information is be read from the checkers (Line 5). This way the SHMU has exact information regarding the broken link(s) in the network. Next, at Line 8 the SHMU loads all usable TMs with 6 turns and sorts them in descending order, based on DoE_{ex} (or DoE, in case minimal path routing is used. Only the 6-turn TMs are used, since they provide the best adaptivity, and thus the highest probability of being able to bypass the broken links. Other usable TMs with less turns are just more constrained subsets of the 6-turn TMs and provide lower adaptivity. The sorted list of the usable 6-turn TMs is stored in `sortedTMs` variable.

All TMs stored in `sortedTMs` are checked for connectivity, one-by-one. For each TM, starting with the ones with the highest adaptivity, a routing graph is generated (Line 11 in Algorithm 2). The generated routing graph is built for a specific TM, under the current fault configuration. Essentially, the edge on the RG, which represents the broken link, is removed. Finally, connectivity check is performed on the RG using Equation 1 (Line 15). If the RG with the fault information provides full connectivity under the TM (there is a connectivity between all node pairs in the network), that TM will be chosen by the SHMU. Otherwise, provided that the connectivity value is higher than the connectivity of any previously checked TM, it is saved. Then, the next TM in the `sortedTMs` list is checked for full connectivity.

If none of the checked TMs provide full connectivity, the TM with the highest connectivity value is chosen. Therefore, the system can continue working with lower performance, even if restoration of full connectivity is not possible.

4.9 Chapter Conclusions

In this chapter, new metrics for profiling and analyzing uniform NoC turn models have been proposed. The newly proposed connectivity metric enables to assess the TM's ability to provide full connectivity between all nodes in the NoC using routing graphs. Additionally, the previously known Degree of Adaptivity (DoA) metric was extended, resulting in the new metric called Extended Degree of Adaptivity (DoA_{ex}). The newly proposed metric enables to assess the adaptivity of TMs (the TM's ability to find alternative paths between node pairs in the NoC) for non-minimal path routing. The old metric could assess the ability for only minimal path routing.

Additionally, metrics for calculating average connectivity and latency of TMs were introduced. The average connectivity for k broken links is a metric, which represents the connectivity values of a TM, averaged over all fault configurations with k broken links. The average connectivity metric of a TM is a direct representation of that TM's reliability.

The new metrics were used to perform a full enumeration, evaluation and analysis of all 256 uniform TMs for 2D full-mesh NoCs. As a result, all 50 usable turn models that are deadlock-free and provide full connectivity, were identified. All usable TMs were then further classified and analyzed in terms of DoA, DoA_{ex}, average connectivity and latency. As a result, many new turn models with good performance where identified. Additionally, a discovery was made that there is a high correlation between the turn model's average connectivity, latency its (extended) Degree of Adaptivity. Finally, an algorithm for choosing the best TM for a fault configuration was proposed.

In contrast, prior to this work, only a small number of the TMs were described and analyzed in the literature. To the best of the author's knowledge, such an in-depth analysis of all TMs as performed in this work has not been done before.

5 Software TMR with Distributed Voting

This chapter introduces STROBES, a software-based fault tolerance algorithm. STROBES relies on a high-level fault model, which enables to implement fault tolerance fully in software, in a way, that the performance of the fault tolerance algorithm does not depend on the behavior of the protected application. As such, the proposed algorithm can be used to protect a wide range of applications by interfacing them with the STROBES library through a simple interface. In contrast, most similar approaches are limited to a very small range of applications with similar behavior or rely on modified or inherently fault tolerant hardware to protect the application.

Furthermore, the usage of the high-level, application agnostic fault model enables to simulate the STROBES algorithm independently of the application. This allows to fine-tune the algorithm's parameters to further optimize its performance and fault tolerance.

Finally, the fault correction capabilities of STROBES are formally proven using Markov analysis and fault injection experiments using a custom application independent high-level simulation tool.

This chapter is based on the following publication:

- **V**
  K. Janson, C. J. Treudler, T. Hollstein, J. Raik, M. Jenihhin, and G. Fey, "Software-level TMR approach for on-board data processing in space applications," in *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 147–152, April 2018

5.1 Introduction

Satellites and other space-based systems must operate reliably, without maintenance, under extreme conditions [20, 21] of outer space for years at a time. One of the worst problems for space-based systems is radiation. A radiation particle can randomly flip a bit in memory or cause a fault during computation [13–15]. This can lead to serious errors or even a complete failure [17–19, 38, 39] of the system. For example, tests performed on unprotected SRAM memories on satellites in space have shown as much as 15 random bit flips per day [18]. Admittedly, the tests reported in [18] were carried out on a system specifically designed for detecting such faults. Due to inherent fault masking that is present in every device, the fault rate in actual systems will be much lower. However, it is easy to imagine that over the normal mission length of a couple of years to a couple of decades the probability of a mission-critical error is very likely.

For this reason, space-based computer systems need to be protected against SEEs, particularly SEUs. The increased fault tolerance in space systems is normally achieved by using specially built hardware, which incorporates some form of redundancy. An example of such hardware would be commercial, specialized, highly reliable (hi-rel) components. Redundancy in hi-rel components is usually implemented at lower levels (e.g., flip-flops are triplicated) [15]).

However, while the hi-rel components provide better reliability when compared to COTS components, they also have many drawbacks. First, the usage of hi-rel components results in an increase in cost because the hi-rel components are considerably more expensive. Secondly, hi-rel components are usually built using older technologies. As such, they have a severely limited performance, compared to COTS components [100]. This motivates the use of COTS components in space applications. However, unlike the

hi-rel components, COTS components do not include built-in fault tolerance. Therefore, systems built from COTS components require additional mechanisms for protection against radiation-induced errors [19, 40–42].

The software-level TMR approach for Error handling in on-board data processing in space applications (STROBES) algorithm, proposed in this chapter, was designed in cooperation with German Aerospace Center (DLR) for use with their ScOSA platform [45].

The aim of DLR's ScOSA platform is to provide a scalable and modular computing platform for satellite-based systems. A secondary goal of the ScOSA platforms is to improve the performance by using networked COTS processing elements (PEs).

Therefore, a fully software-based fault tolerance algorithm was needed. An additional requirement was that the fault tolerance algorithm should be application agnostic, applicable to many applications with different behaviors with minimum amount of change to the code of the algorithm and the application.Finally, it should be possible to formally assess the fault tolerance of the proposed approach.

Therefore, the proposed fault tolerance algorithm is a fully software-based approach. Unlike many similar methods, presented in Section 5.2 it requires no changes to the hardware. This is especially important for systems that utilize COTS PEs.

The STROBES algorithm relies on a high-level fault model, described in Section 5.3. The fault model enables to detect faults that occur software running on a multiprocessor system. The fault detection is performed on a very high abstraction level, while staying agnostic regarding the implementation details and behavior of the software application that is used to protect against errors. As the STROBES fault tolerance algorithm is directly derived from the fault model, it also inherits the fault model's high-level fault detection capability.

As mentioned, the high-level fault model makes STROBES algorithm application agnostic. This means that its performance is not influenced by the behavior of the protected application. The performance of the algorithm depends on the worst-case parameters of the system that it is run on, the size of the application's state, and internal timeouts.

The values of these internal timeouts can be found through formal techniques and fine-tuned by analysis in high-level application independent high-level simulation of the STROBES algorithm. This approach is used in Section 5.8 to provide application-independent experimental data using a custom simulation environment, described in Section 5.7.

Since it is application agnostic, STROBES can be used to protect almost any application, regardless its functionality. It can be added to the application using a wrapper library. To add STROBES support, the application is to provide a mechanism for the STROBES library to monitor the application's state and to load the checkpoint data in order to perform a roll-back to a previous checkpoint, if needed. The simplest way to achieve this is in a single-threaded application is to make the variables that make up the application's state available to the library and insert a call to the STROBES library's synchronization function to the main loop of the application.

Therefore, it is easy to add support for STROBES to existing applications. The only requirement from the application's side is that it can handle the additional time overhead introduced by addition of STROBES. Additionally, there is no need to change the algorithm when the application is updated. All that is needed in this case is to re-compile and re-link the application with the STROBES library. As shown in Section 5.2, being application agnostic distinguishes STROBES from most other fully software-based

fault tolerance approaches. In most other approaches, the fault tolerance system is tightly coupled to a specific application or a small group of applications in a way that it cannot be easily separated.

Moreover, the reliability of the STROBES algorithm is formally proven using Markov chains (Section 5.6). The Markov analysis can also be used to assess the reliability of the system under different application parameters.

STROBES works by running the same application simultaneously on multiple interconnected PEs. Error detection and correction is performed using distributed majority voting on the application's state. Distributed voting helps to eliminate the voter as a single point of failure, while still providing consensus regarding the system's fault status. STROBES can correct errors in the application's state that were caused by transient faults, such as SEUs. Additionally, STROBES can also correct timing errors. Small timing errors, such as the ones caused by different network communication delays of different PEs, are corrected transparently, while a special mode is used for handling longer timing errors. The STROBES algorithm can also tolerate network errors, if there are still enough available and communicating PEs for the majority voting to succeed.

Last, but not least, STROBES' use is not limited to ScOSA-based satellite on-board computers. The STROBES algorithm can be also used to provide increased fault tolerance in terrestrial systems or even as an additional fault tolerance layer for systems that utilize hi-rel components.

This chapter is organized as follows: First, Section 5.2 gives an overview of related works. Then, in Section 5.3, the basic system structure for running the STROBES algorithm and the fault model used in this work are explained. This is followed by Section 5.4, which explains the detailed working of the STROBES algorithm. Next, Section 5.5 gives an overview on how fault handling and the synchronization of the PEs is done. Then, in Section 5.6, a reliability analysis of the STROBES algorithm is performed using the Markov model. Next, Section 5.7 gives an overview of the simulator used for obtaining the experimental fault handling performance results for the STROBES algorithm. The experimental results are presented in Section 5.8. Finally, the chapter is concluded in Section 6.

5.2 Literature Review

Many fault tolerance solutions for embedded systems require hardware modification. This is often done by implementing variations of hardware-based TMR solutions, such as [101–103], or by utilizing using soft-core PEs on field-programmable gate arrays (FPGAs) [104, 105]. Additionally, some approaches like [106–108] duplicate or triplicate parts of the processor's pipeline to achieve instruction level redundancy.

However, such solutions are not usable in a system such as ScOSA, whose main goals include developing a modular and easily extensible system. Additionally, the usage of high-performance COTS PEs rules out any changes to the PEs themselves. For this reason, in contrast to the approaches mentioned above, STROBES is designed to be a completely software-based solution. It requires no modification to the hardware it runs on.

Fully software-based fault tolerance algorithms, or so-called consensus algorithms, are not new. They have been around for decades and are widely used in data centers. Examples of such algorithms are Paxos [109] and Raft [110] and their derivatives [111]. However, these algorithms are usually specifically tailored for data center usage or not directly applicable to embedded systems. Most notably, such algorithms are not real-time-safe, which is requirement for many embedded systems.

The STROBES algorithm satisfies the real-time requirements by relying on deterministic timeouts for synchronization between PEs that run copies of the same application. Furthermore, the timeouts used in STROBES can be formally calculated and the algorithm can be simulated during the development phase using only basic data information about the system and no application details. The simulation can be used for design-space exploration in order to find the best results parameters for the algorithm for the specific system.

There exist other software-based fault tolerance algorithms that are designed for embedded systems. However, many approaches are only applicable for very specific parts of the software. For example, SHiFA [112] proposes dynamic process mapping to healthy processing elements in multi-core systems. However, this is done for only small computational kernels only which are dispatched and terminated. This approach is similar to the one in [113] which also takes anti-aging into account an additional parameter during task distribution.

Other solutions require modification to the underlying *operating system* (OS) to function. For example, [114] proposes a software-based TMR approach for critical tasks. However, fault-isolation and recovery are expected from the system it is run on. On the other hand, the authors of [115] present a real-time capable synchronization mechanism to be integrated into a real-time Operating System (RTOS) while protecting the entire application and being hardware agnostic. However, this approach requires customized OS and the application to be specifically written with this modified OS in mind.

Another well-known approach for providing software-based fault tolerance is compile-time instruction duplication, where redundant instructions are inserted into the protected application during compilation. Examples of such approaches are [116–118] and [119]. However, this approach has many drawbacks. Mainly, it increases the application's binary size and execution time considerably, while still providing only error detection, not correction. Many approaches attempt to address this issue by protecting only a limited number of critical instructions [120], which reduces fault coverage. Moreover, most software-based instruction duplication approaches can only detect single bit flips in the computational logic. However, gate-level fault injection experiments show that only 77% of errors in the processor's computation logic manifest themselves as single bit flips [120]. This means that 23% of all errors cannot be detected at all using such methods. In addition, these methods cannot usually detect errors in the memory or in the cache. Therefore, as concluded in [120], compile-time instruction duplication approaches are not suitable for use in systems where high levels of reliability is required.

Like the theoretical framework of [121], the STROBES algorithm makes no assumption of underlying synchronization mechanisms, but only relies on worst-case execution and communication times. However, unlike STROBES, the approach presented in [121] requires deterministic transmission times from the network.

Additionally, different from the asynchronous approaches for fault handling [122,123], timing assumptions are made throughout the implementation.

The STROBES algorithm provides a software-based reliability layer for unreliable PEs. It and does not require modifications to the operating system or the hardware it is run on. Unlike most other fault tolerance mechanisms found in the literature, it is application agnostic. Its performance does not depend on the behavior of the application it is protecting, but only on constraints set by the system it is running on and its own internal parameters. Hence, support for STROBES can be easily added to most applications using a light-weight wrapper library. Also, the application can be updated without changing the STROBES algorithm.

Those properties allow STROBES to provide software-based fault tolerance for critical applications running on unmodified COTS-based PEs. Alternatively, it can be used as an additional fault tolerance layer for applications running on hi-rel hardware. Unlike many more traditional consensus algorithms, STROBES is not limited to any specific application domain and supports real-time applications.

5.3 System Requirements and Fault Model

Any software can be represented as a finite-state machine (FSM) – a set of variables that make up the state of the application, and operations performed on that state. SEEs can cause error at the software level by either directly modifying the state by flipping bits in the memory or by causing a fault during computation, which leads to a wrong value being written to the state. This enables software-based solutions like the STROBES algorithm to manage the effects of transient hardware faults in the PEs at software level, since most errors will eventually be visible in the application state.

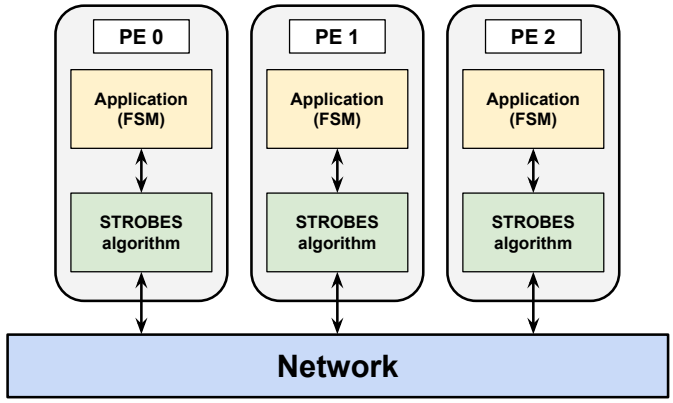


Figure 21: Abstract diagram of a system protected by the STROBES algorithm

STROBES works by running multiple copies of the protected application in parallel on n PEs in periodically occurring cycles (Figure 21). Each processing element PE_i ($i \in \{0 \ldots n-1\}$) runs an instance of the STROBES algorithm. While in this chapter we focus on the case study of TMR (i.e., $n = 3$), targeting a data-processing system with three PEs, the algorithm is scalable, and a larger number of simultaneous faults could be handled with more processing PEs ($n > 3$).

Essentially, the STROBES algorithm works by creating a *checkpoint* (CP) of the application state at regular intervals and performing a distributed majority voting on that CP. The specifics of the algorithm used for this are explained in Section 5.4. In order to support the distributed voting approach of, the interconnection network needs to support broadcasting. However, the network does not need to guarantee deterministic arrival times for data.

Fault tolerance is achieved by using a high-level fault model. The fault model defines three fault types:

1. $SF = SF_1, \ldots, SF_i$ – **Corruption of the application state**. Fault SF_i denotes that the application state of PE_i is corrupt at the time of creating the CP. Type-*SF* fault can occur for different reasons (e.g., bit-flip in the application memory due to radiation, fault in calculation, etc.). However, the source of the fault is not important as the behavior for handling the faults is the same.

2. $TF = TF_1, \ldots, TF_i$ – **Timing faults**. Fault TF_i in PE_i means that at the point of CP creation PE_i was either ahead or behind of other PEs in terms of software execution.

   Typical timing faults (Type-*TF*) include de-synchronized state of processing elements upon system start-up, clock drift during operation, different application run times on processing elements because of different loads and network delays. Small timing faults are an inherent in any distributed system and, in case of STROBES, are tolerated using timeouts. Large timing faults are more serious and are treated similarly to Type-*SF* faults.

3. $NF = NF_1, \ldots, NF_i$ – **PE stops processing**. A Type-NF_i fault denotes that PE_i has fell silent. Type-*NF* faults model not only severe PE failures but also network partitioning. This is because from a system's perspective, a stopped PE is indistinguishable from a PE which is unreachable due to network failures.

Additionally, the following conditions apply regarding faults which can be tolerated together in a STROBES system with three nodes ($n = 3$):

- A STROBES system can tolerate, at maximum, **a single Type-*SF* or Type-*NF* fault at a time**.
- There is **no limit to the amount of Type-*TF* faults at a time** it can handle.
- The two fault domains, $SF \cup NF$ and TF have, in practice, different root causes. As a result, the STROBES system **can handle any amount Type-*TF* faults at the same time as a single Type-*SF* or Type-*NF* fault**, as long as the timing difference between the PEs stays within the limits of the system timeout defined in Section 5.4.

The proposed fault model allows to abstract the faults in the multiprocessor system, while not only covering errors in application state (and by extension, its behavior), but also synchronization and network problems that can occur in distributed systems. The fault model allows to make the proposed STROBES algorithm application agnostic. Therefore, it can be used with any application that can handle the time delays, which are introduced by the STROBES algorithm. In contrast to similar works, STROBES can protect applications regardless of their behavior.

An application can be protected using the STROBES algorithm, by just adding a wrapper library. The left subfigure of Figure 22 shows a simple abstract application consisting of a set of variables and a main loop. The application performs some calculations in the *calculate()* function, using the variables declared earlier. The contents of *calculate()* are not relevant in this context.

The right subfigure of Figure 22 illustrates modifications that are needed to make application on the left STROBES compatible. In short, the main change to the application is to make the variables of the application's state available to the STROBES library. In the example presented in Figure 22, all variables are tied together into a struct. This struct is then passed to the STROBES library's synchronization function. The synchronization function is run once per every main loop cycle, although, if needed, it can also be run more often. However, this would introduce additional latency.

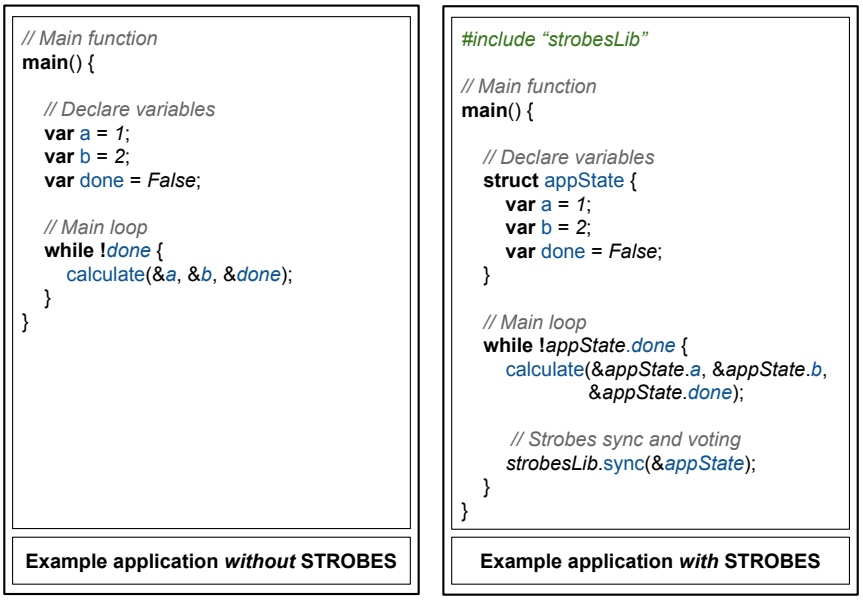


Figure 22: An example on adding the STROBES library to an application

5.4 STROBES Fault Handling Algorithm

5.4.1 Communication Between the Processing Elements

As a distributed algorithm, in order to achieve a consensus on system's fault status and synchronize the PEs, STROBES needs a to implement communication between the PEs. This subsection will give a short overview of the messaging system used by STROBES. Each message contains the following fields:

- **Type.** Message type identifies how should the receiver to react to the message. In STROBES there are defined six types of messages (see Table 6).
- **Cycle.** The sender's cycle counter value at the time of transmitting the message. This serves as a timestamp used to identify old messages and PEs that are out on sync with the rest of the system.
- **Payload (optional).** Data messages, like *Checksum* and *Checkpoint*, also carry the data as payload. Control messages do not have this field. Additionally, STROBES uses different message queues for data and control messages. This is needed to guarantee timely arrival of the control messages without losing any data. Both buffers are flushed in the beginning of every cycle.

In general, as explained in Table 6, the messages used in STROBES can be divided into the following three groups.

- **Checksums.** As explained in Section 5.4, the checksums of the application state are distributed once every cycle, following Algorithm 4. The checksum communication is used for Type-*SF* fault detection and for synchronization.

Table 6: Message types used for synchronization in the STROBES algorithm

Message Group	Type	Priority	Transmission Method	Comment
Checksum	*Checksum*	Low	Broadcast	-
End of Cycle Sync.	*NextCycle*	Low	Broadcast	-
Checkpoint Transfer	*CPRequest*	High	Unicast	CP transfer request
	CPAck	Low	Unicast	CP transfer Acknowledge
	Checkpoint	Low	Unicast	CP transfer
	CPTrFinish	Low	Broadcast	CP transfer is finished

- **End of cycle synchronization.** At the end of each cycle, the PEs will synchronize by broadcasting the *NextCycle* message (as explained in Table 6). When a *NextCycle* message has been received from all available PEs, the receiver continues with processing the next cycle. A PE will only wait for other PEs who did not send their checksum for a short timeout. If at the end of the timeout the PE has not received a *NextCycle* message from all other PEs, it will continue with processing the next cycle to avoid the system waiting indefinitely for a PE that is not available (has Type-*NF*) fault.
- **Checkpoint transmission.** Checkpoint transmission includes multiple synchronization using multiple messages to make sure that the faulty PE is ready to receive the CP and guarantee re-synchronization of the system after the Checkpoint transfer. The details of this can be seen in Section 5.4.

In addition, as shown in Table 6, the messages have been divided into high and low priority messages. Low priority messages will be processed by receiving PE when the algorithm execution arrives at the point where the message should be processed. On the other hand, high priority messages interrupt the receiving PE, causes it to address the high priority message. STROBES defines only the checkpoint transfer request message (*CPRequest*) as a high priority. When a PE receives a *CPRequest* messages, it means that the receiving PE has been identified as faulty and it should immediately continue to receive the correct CP. Also, CP transfer uses unicast messaging, while messages that are used for synchronization, such is sharing the checksums and the end of cycle synchronization message, are transmitted using broadcast.

5.4.2 Algorithm Description

The STROBES algorithm is run in parallel on three PEs. Each PE runs an instance of the algorithm, that is summarized in Figure 23. Each stage of the algorithm is explained in more detail below.

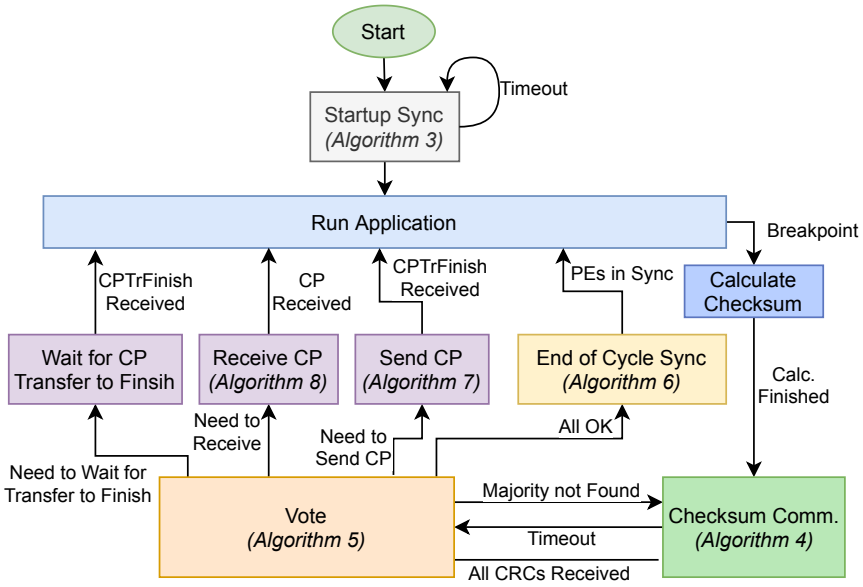


Figure 23: Diagram of the STROBES algorithm. Each PE runs an instance of this algorithm

Algorithm 3 Startup synchronization algorithm running in each PE

```
1:  typedef PEChkSum = (cycle, checksum)
2:  chkSumMap = map(PE_id, PEChkSum)
3:
4:  // Save the data for current PE
5:  // Since we are at startup, both cycle and checksum are 0
6:  chkSumMap[myID] = (0, 0)
7:
8:  // Loop until we have received checksums from all PEs
9:  while True do
10:     // Send an empty checksum
11:     net.broadcast(msgType=checksum, data=chkSumMap[myID])
12:
13:     // Repeatedly send checksum until all PEs in the system are synchronized
14:     while !commTimeout do
15:         if !net.buffer.isEmpty() then
16:             (type,cycle,PE_id,data) = net.buffer.pop()
17:             if type == checksum then
18:                 chkSumMap[PE_id] = (cycle, data)
19:
20:         else if chkSumMap.size() == totalPECount then
21:             // Send one more checksum
22:             net.broadcast(msgType=checksum, data=0)
23:             nextState = Run Application
24:             return
```

Startup Synchronization. When the STROBES-protected distributed embedded system is first booted up, each PE will come online at a slightly different time. For this reason, the system will boot into an the *Startup Sync.* stage, which is performed using Algorithm 3. Loosely, each PE broadcasts a dummy checksum e.g., all zeros (line 11) and then waits for dummy checksums from 2 other PEs (while loop at line 14). The content of these messages is ignored as they are used only for synchronization. When all three checksums have been received (line 20), each PEs will transmit an additional dummy checksum. This is needed for the PE that was initialized the latest, since otherwise it does not receive any checksums from other PEs because they were sent before the last PE was fully booted. Once all the PEs have received the second set of dummy checksums, the system is synchronized and ready for running the protected application. The algorithm uses dummy checksums instead of a generic initialization sequence to tolerate reintegration of a previously silent PE (a PE with a Type-*NF* fault). Because, in this case, the faulty PE will send out a checksum upon boot, it will be synchronized during the *Checksum Communication* step of the algorithm. In this case, the faulty PE will appear to the other PEs as as if it has both state (Type-*NF*) and timing (Type-*TF*) faults, which will result in one of the correct PEs sending the faulty PE the correct CP and thus the faulty PE being reintegrated into the system using mechanisms that are explained below.

Run Application. The *Run Application* stage in Figure 23 indicates parallel independent data processing by all PEs. The duration of this step (t_p) is application specific. The application is executed until a breakpoint is reached. The positioning of breakpoint is done during development time. While the details on where and how to place the breakpoints into the application are out of the scope of this thesis, a shorter execution time per each algorithm cycle will result in higher reliability. As shown in Section 5.6. The areas of the application's memory covered by the CP, and thus protected by STROBES, are also application specific and not defined by the STROBES algorithm.

Calculate Checksum. Once the breakpoint has been reached and the application has been paused, the current state of the application's memory is called the CP. Next, each node will calculate a checksum over its CP (*Calculate Checksum*). Any corruption in application's state is assumed to result in an erroneous checksum. The types of checkpoint corruption which can be detected depends on the type of checksum used. In this thesis, a 16-bit *Cyclic Redundancy Check* (CRC) checksum has been used as a proof of concept, however the checksum type is not specified by the algorithm.

Checksum Communication. The checksum communication stage is used for exchanging data and for synchronization using through timeouts. After a PE has finished calculating the checksum, it will broadcast it to all other PEs. (*Checksum Communication*). As a result of the checksum communication stage, each PE will have acquired the checksums from every other PE.

The *Checksum Communication* stage is detailed in Algorithm 4. After the checksum has been broadcast, the algorithm reads the receiving buffer (line 11) and stores any received checksum (line 13) until the receiving buffer is empty (line 10). This process continues until checksums from all PEs have been received or the timeout is reached (timeout occurs when some of the nodes do not send their checksum) (line 9). When either of the condition is met, the system will proceed to the *Vote* stage. Because the *Checksum Communication* stage implements predefined timeouts for waiting the checksums from other nodes, it also functions as the main synchronization mechanism for the STROBES algorithm and will be explained in more detail in Section 5.5.

Algorithm 4 Checksum communication algorithm running in each PE

```
 1: typedef PEChkSum = (cycle, checksum)
 2: chkSumMap = map(PE_id, PEChkSum)
 3:
 4: // Save the data for current PE
 5: chkSumMap[myID] = (currCycle, calcChkSum(CP))
 6: net.broadcast(msgType=checksum, data=chkSumMap[myID])
 7:
 8: // Wait until we receive checksums from all PEs or reach the timeout
 9: while (chkSumMap.size() < totalPECount and !commTimeout) do
10:     while !net.buffer.isEmpty() do
11:         (type, cycle, PE_id, data) = net.buffer.pop()
12:         if type == checksum then
13:             chkSumMap[PE_id] = (cycle, data)
14: nextState = Vote
15: return
```

If one PE stops processing, i.e., it suffers a Type-*NF* fault, the processing performance of the overall system degrades. All PEs will always wait first for the timeout to receive checksums. However, overall correct operation will continue until at least two PEs are functional.

Vote. In the *Vote* stage, each PE independently performs majority voting of over the checksums. The algorithm for performing the voting is shown in Algorithm 5. The algorithm starts with majority voting on the current cycle number for the received checksum messages to ensure that all PEs are synchronized with each other. Since it is assumed that a PE with faulty cycle ID will have also a faulty checksum, a PE with an erroneous cycle number will receive the correct CP (line 11). Next, majority voting determines the correct checksum. If all checksums are different, then the algorithm proceeds to the *Checksum Communication* step (line 16). If all checksums are the same everything is correct and the current cycle will be finished, the PE will activate the *End of Cycle Synchronization* stage (line 19). In case a fault is detected, a faulty PE will receive the correct CP (line 22), Algorithm 8; one PE with correct checksum will send the CP (line 26), Algorithm 8 and all other PEs with correct checksum will wait for this update to finish (line 29).

Algorithm 5 Distributed voting algorithm running in each PE

```
1: // chkSumMap is acquired by Algorithm 4
2: if chkSumMap.size() == 0 then
3:     nextState = Checksum Communication; return
4:
5: // Majority vote over the cycle numbers
6: cycleMajority = voteCycleNumber(chkSumMap)
7: if all cycle numbers are equal then
8:     nextState = Checksum Communication; return
9:
10: if currentPECycle != cycleMajority then
11:     nextState= Recv. checkpoint; return
12:
13: // Majority vote over checksums
14: chkSumMajority = voteChkSum(chkSumMap)
15: if all checksums are different then
16:     nextState = Checksum Communication; return
17:
18: if all checksums equal and chkSumMap.size() == totalPECount then
19:     nextState = End of Cycle Sync; return
20:
21: if currentPEChkSum != chkSumMajority then
22:     nextState = Recv. checkpoint; return
23:
24: minAddr = minimal address in correctPEs
25: if current PE's address == minAddr then
26:     nextState = Send checkpoint
27: else
28:     // Block and wait until CPTrFinish message is received (CP transfer is finished)
29:     net.blockingWaitForMsg(msgType=CPTrFinish)
30:
31:     cycleCounter += 1
32:     nextState = Run Application
33: return
```

Algorithm 6 End of cycle synchronization algorithm running in each PE

```
1: // Including our own message
2: msgRecvdCounter = 1
3:
4: net.broadcast(msgType=NextCycle)
5: // Wait only for PEs which sent checksum during the current cycle
6: while msgRecvdCounter < chkSumMap.size() do
7:     while !net.buffer.isEmpty() do
8:         (type, cycle, PE_id, data) = buffer.pop()
9:         if type == NextCycle then
10:            msgRecvdCounter += 1
11: cycleCounter += 1
12: nextState = RunApp
13: return
```

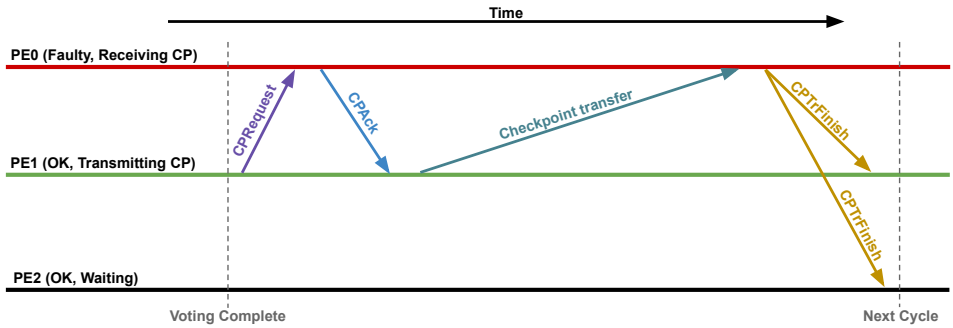


Figure 24: Diagram showing the communication between the PEs during CP transfer

End of Cycle Synchronization If voting succeeded and no problems, which require CP transmission were detected, each PE will enter the *End Cycle Sync.* stage (Algorithm 6). During this stage, each PE will broadcast a *NextCycle* message. Upon receiving the message from all other available PEs, the PE will continue to the next cycle and continue with the *Run Application* stage.

Algorithm 7 Correct checkpoint sending algorithm running in each PE

```
1: *CP = Pointer to application's checkpoint memory
2: // Send CPRequest message to the faulty node
3: net.transmit(destination=faultyNode.addr, msgType=CPRequest)
4:
5: // Block and wait until CPAck message is received from the faulty node
6: net.blockingWaitForMsg(msgType=CPAck)
7:
8: // CPAck received, transmit the checkpoint
9: net.transmit(destination=faultyNode.addr, msgType=Checkpoint, data=*CP)
10:
11: // Block and wait until CPTrFinish message is received
12: net.blockingWaitForMsg(msgType=CPTrFinish)
13:
14: cycleCounter += 1
15: nextState = Run Application
16: return
```

Algorithm 8 Algorithm for receiving the correct checkpoint

```
1: *CP = Pointer to application's checkpoint memory
2: global haveRecvdCPRequest
3:
4: // CPRequest is a high priority message and causes an interrupt which triggers
5: // this function. However, this function can also be run from the Vote function
6: // For this reason, we only wait for the CPRequest if we have not received it yet
7: if !haveRecvdCPRequest then
8:     // Wait for CPRequest
9:     net.blockingWaitForMsg(msgType=CPRequest)
10: // Respond with the CPAck message to the PE which sent the CPRequest
11: net.transmit(destination=msgGetSender(CPRequest), msgType=CPAck)
12:
13: // Block and wait until Checkpoint is received and saved
14: *CP = net.blockingWaitForMsg(msgType=Checkpoint)
15:
16: // Broadcast CPTrFinish to re-synchronize the system and continue with execution
17: net.broadcast(msgType=CPTrFinish)
18:
19: cycleCounter += 1
20: nextState = Run Application
21: return
```

Send CP / Receive CP If a fault is detected during the *Vote* stage, the correct PE with the lowest address will enter the *Send CP* stage (Algorithm 7) and send a copy of its CP to the faulty PE (*Receive CP*), Algorithm 8. The other healthy nodes will wait for the transmission to finish. The transfer process is depicted in Figure 24.

As shown in Figure 24, first, the correct PE with the lowest address (CP Sender), *PE 1* in Figure 24, will send an *CPRequest* message to the faulty node (CP Receiver), *PE 0* in Figure 24. This is shown in line 3 in Algorithm 8. The *CPRequest* is a high priority message. This means that when a PE receives this message, it will stop whatever is doing and imminently switch to the *CP Receive* stage. This is useful in case the faulty PE is not properly synchronized, which might result it not understanding on its own that it is faulty.

When the CP Receiver receives the *CPRequest* message, it will respond with *CPAck* message (line 11 in Algorithm 8). When the CP Sender receives the *CPAck* message, handshaking for CP transmission is complete. This is followed by the CP Sender transmitting the actual *Checkpoint* (line 9 in Algorithm 7), after which it will switch to waiting for the *CPTrFinish* message (line 12 in Algorithm 7).

Upon receiving the *Checkpoint*, the CP Receiver will overwrite its memory with the contents of the received CP. The transaction is finished by the CP Receiver broadcasting the *CPTrFinish* message (line 17 in Algorithm 8). This message will be received by all PEs in the system and will cause every PE, including the CP Receiver, to continue execution from the next cycle. Note that there is no additional end of cycle synchronization in this case, since the *CPTrFinish* message will guarantee synchronization.

5.4.3 Discussion on Real-Time Implementation of STROBES

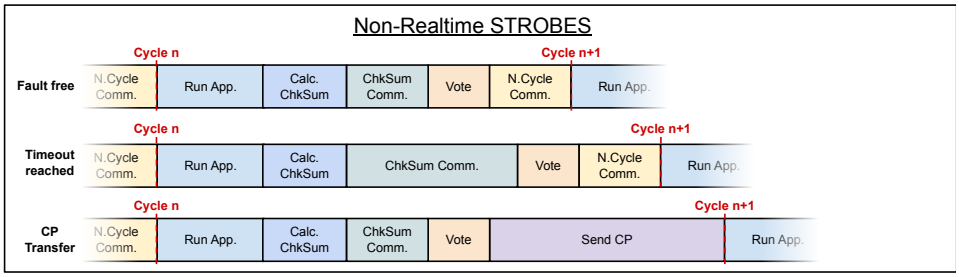
Real-time applications require predictable, deterministic timing. However, the baseline STROBES has a variable cycle length. The time spent in *Checksum Communication* stage greatly depends on the existence and the severity of Type-*TF* faults in the system because a PE will continue execution whenever it receives the checksums from all other PEs. Additionally, when there is a Type-*SF* fault in the system, the system will perform a CP transfer. However, this does not happen otherwise. The time spent transferring the CP depends on the network speed and checkpoint size, but realistically, it can take hundreds of milliseconds. This results in quite large variability in cycle range and while it has a positive impact on performance, it is not suitable for real-time applications.

Figure 25 visualizes the behavior of a single PE in the STROBES algorithm under different conditions for both the standard, non-real time, version of STROBES (Figure 25a) and its real-time adaptation (Figure 25b).

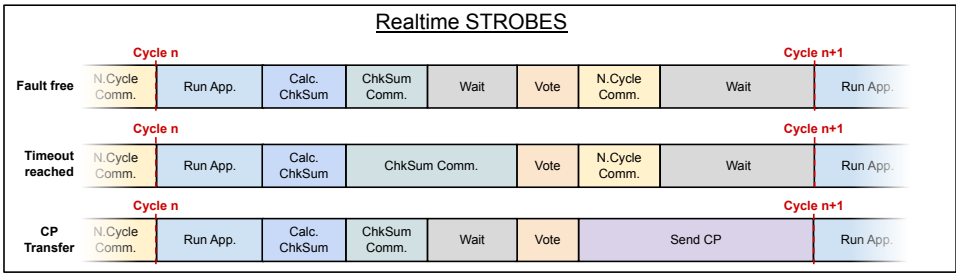
Both subfigures look at the same three cases. The "Fault free" case, represents the fault free situation where checksums from all PEs are received without much delay and majority voting does not result any errors, thus no CP transfer is needed. The "Timeout reached" case visualizes a system with a Type-*TF* fault, a situation where some of the PEs will not send their checksum before the timeout is reached. And finally, the "CP Transfer" case, representing a system with Type-*SF* fault which requires CP transfer to correct.

As seen in Figure 25a, the baseline version of STROBES has a high variability in cycle length and when there are no faults in the system, it will be much faster, compared to a faulty case. However, this is not suitable for real-time systems. For this reason, in the real-time version of STROBES, determinism is introduced by making the variable length components of the algorithm always equal to the maximum execution time (Figure 25b). This guarantees a consistent cycle length. In the figure, the added

padding is visualized using gray "Wait" blocks. As seen in Figure 25b, in the real-time version of the algorithm, the *Checksum Communication* stage always runs until the timeout is met, regardless if all checksums were received. The same is true for CP transmission, each PE will wait for the time required to transmit the CP at the end of the cycle no matter if the checkpoint transmission is performed or not. While this approach complies with the predictability requirement of real-time applications, it considerably slower than the baseline version. For this reason, the rest of the chapter concentrates on the baseline, non-real time version.



(a) Non-real-time version of STROBES



(b) Real-time version of STROBES

Figure 25: Differences between the real-time and non-real-time STROBES implementations. Each subfigure visualizes the behavior of a single PE under three different fault scenarios.

5.5 Fault Handling and Synchronization in STROBES

The STROBES algorithm uses two mechanisms for fault handling – timeouts in the *Checksum Communication* stage and transferring of the correct CP to the faulty PE. The two mechanisms have different use cases and the proper method for fault correction is chosen by the algorithm based on the type of the fault.

Checksum waiting timeouts (referred to just as "timeouts" from hereafter) are used for handling Type-*TF* faults, while CP transfer is mostly utilized for repairing Type-*SF* faults. However, sometimes both approaches are used together. This happens when the system has simultaneously both a Type-*TF* fault and a Type-*SF* fault.

There are also other faults, which manifest themselves similarly to having both a Type-*TF* fault and a Type-*SF* fault in the system, and can, thus, be also handled the same way. An example of such fault is a longer Type-*TF* fault, a timing fault where a PE is so much out of sync with the rest of the system that it is executing a completely different algorithm cycle and thus has a different memory CP than the rest of the system. The same situation happens when a PE with Type-*NF* fault is being re-integrated into

the system. In this case the fault manifests similarly to a long Type-*TF* fault since the PE will be desynchronized and will also have different memory contents. Therefore, in these cases, the faults are handled the same way as if there would be simultaneous Type-*TF* and Type-*SF* faults in the system.

This section further explains the fault handling approaches by using a simplified counting application as an example. The application memory protected by STROBES consists only of a single integer value. The application increases the value by one every cycle. While this example application is not very realistic use case for the STROBES algorithm, it is good for demonstrating the fault handling approaches used in the algorithm. The STROBES algorithm itself is application agnostic and works the same with more complex applications.

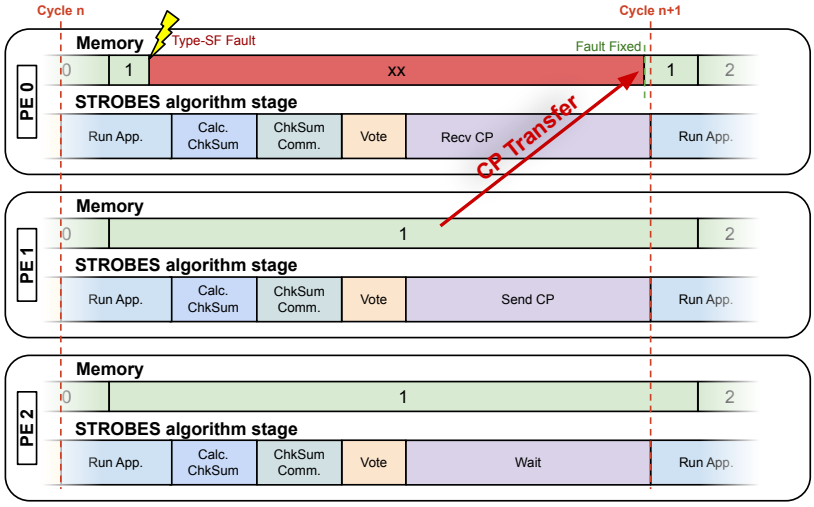


Figure 26: Correcting a Type-SF fault

Figure 26 demonstrates how the STROBES algorithm corrects a Type-*TF* fault. Figure 26 shows a case where there is a Type-*SF* fault in *PE 0*. During cycle n, the application increases the counter from 0 to 1, but then a Type-*SF* fault occurs and corrupts its memory. Corrupt memory is marked with red color and "xx" as the value in memory. All PEs continue executing normally until the *Vote* stage. During voting, the PEs will figure out through distributed majority voting that *PE 0* has a faulty CP. This will cause the correct PE with the lowest address, in this case *PE 1*, to transfer its CP to *PE 0*. This results in *PE 0* overwriting its memory with a correct CP and will, therefore, have the same value as other PEs in its memory once again. Afterwards, all PEs continue fully synchronized and with correct memory.

On the other hand, handling Type-*TF* faults depends on the length of the fault (how much out of synch the execution of the faulty PE is compared to the rest of the system). Certain amount of timing (Type-*TF*) faults always exists in every distributed system, including STROBES-based systems. This happens due different load on the PEs, different network delays, etc. For this reason, the STROBES algorithm includes timeouts for waiting for checksum messages sent during every cycle. As seen in Algorithm 4 in the previous section, all PEs will wait for the checksums from other PEs until they either receive checksums from all of them or until the timeout is reached. The timeout helps to guarantee that the PEs will wait for the messages for long enough every cycle

to compensate for the inherent timing differences. However, at the same time it enables the system to continue working with limited fault correction capabilities, instead of the entire system crashing, when a PE becomes unresponsive and does not send its checksum. An example of such unresponsive PE would be a PE which suffers from a Type-*NF* fault.

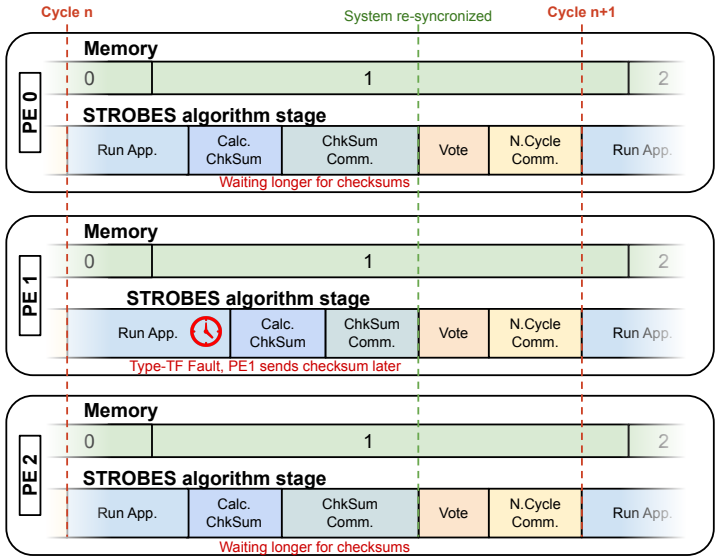


Figure 27: Correcting a Type-TF fault which is shorter than the communication timeout

Figure 27 illustrates correcting short Type-*TF* faults. In the figure, application execution for *PE 1* takes longer time than for other PEs. This can happen, for example, when PEs run also other applications in addition to the STROBES-protected application and therefore have different processor loads. However, short Type-*TF* faults caused by other inherent delays, such as different network performance between the PEs, are handled the same way. It can be seen in the figure, that *PE 0* and *PE 2* will wait for longer for the checksums until they receive them from all PEs, including *PE 1*. After the *Checksum Communication* stage, the system is re-synchronized. This means that short Type-*TF* faults will be handled completely transparently to the application.

Longer Type-*TF* faults, as seen in Figure 28 cannot be handled transparently and require a CP transfer to be corrected. In the figure, *PE 2* has a Type-*TF* fault, which is longer than the timeouts for waiting for the checksums. *PE 0* and *PE 1* wait for the checksum of *PE 2*, but do not receive it before the timeout is reached and thus continue executing the next cycle because they can still agree on the majority result amongst each other. Meanwhile, *PE 2*, the faulty PE, will not receive any checksums and will continue trying to re-synchronize, by repeatedly sending its checksum after every timeout and waiting for the checksums of other PEs. Once *PE 0* and *PE 1* reach the *Checksum Communication* stage of the next cycle, all PEs will synchronize, but majority voting will result *PE 2* being detected as faulty. This happens since it is still executing Cycle n while the other PEs have moved on to Cycle $n+1$. *PE 2* will be corrected and resynchronized the same way as it had both a Type-*TF* and a Type-*SF* fault. Correcting Type-*NF* faults works the same way because, like mentioned before, a Type-*NF* fault manifests as a PE being both out of sync with the rest of the system and having a different CP value.

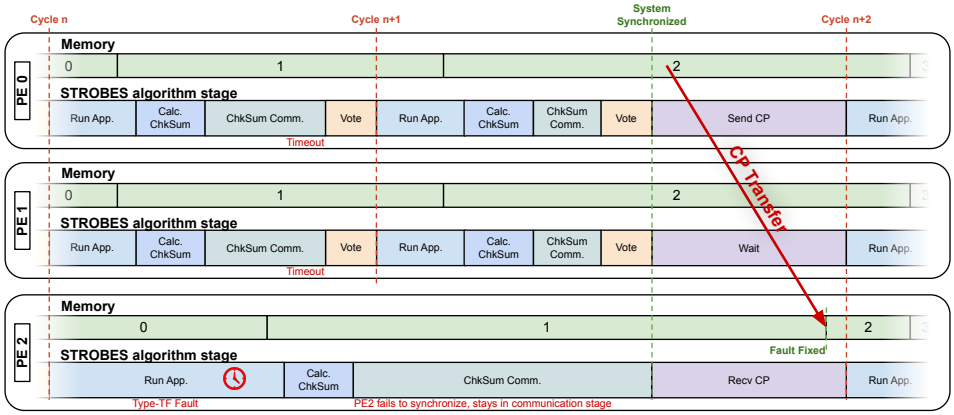


Figure 28: Correcting a Type-TF fault which is longer than the communication timeout

The example explained earlier in this section helps to specify the conditions for choosing a proper value for the timeout. As explained earlier, the timeout is needed for handling inherent Type-*TF* faults in the system, caused by different processor loads in the PEs, different network delays, etc. The timeout needs to be large enough to cover those delays. In other words, as illustrated by Equation 4, the minimum length of the timeout ($chksum_{timeout}$) needs to be longer than the sum of the application execution time per cycle under the worst-case processor load ($MAX(t_p)$), the checksum calculation time under the worst-case processor load ($MAX(t_c)$) and the maximum network delay ($MAX(network\_delay)$).

$$chksum\_timeout > MAX(t_p) + MAX(t_c) + MAX(network\_delay) \quad (4)$$

The maximum value for the timeout is not defined by STROBES. However, the longer the timeout value is, the longer Type-*TF* faults the STROBES algorithm can correct without a CP transfer (as shown in Figure 27). For this reason, the value to be chosen for the timeout depends on the application. A real-time application will always wait until the maximum timeout, so longer timeouts will considerably slow down the system, while, in that case the CP transfer will be "free", since the real-time version of the algorithm always waits for the time required for CP transfer anyway before continuing the execution, as explained in Section 5.4.3. A non-real-time system, on the other hand, would likely benefit from a longer timeout values, since in this case CP transfer costs a lot of time and longer timeouts reduce the need for CP transmission.

5.6 Reliability Assessment

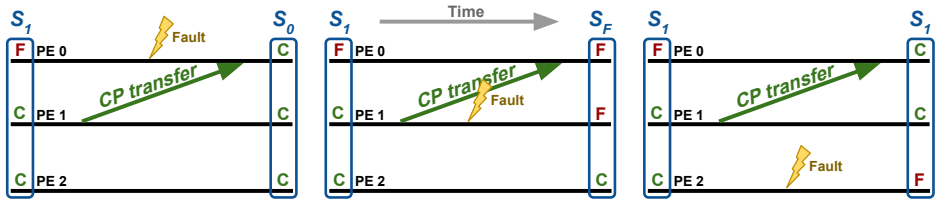
In this section, Markov reliability analysis of the STROBES algorithm is performed. The results of this analysis help to assess the feasibly of the STROBES algorithm for improving reliability of software applications. Additionally, it allows to better specify the constraints for the applications that can be protected using the proposed algorithm.

5.6.1 Definition of Failure Condition

In order to perform the reliability assessment for a system, first it is necessary to define a failure condition for the system under investigation. In a STROBES system, a failure occurs when the majority voting either fails or results in a wrong result. In any case, there must be at least two Type-*SF* faults in the system at the same time for this to happen. This failure can occur for the following reasons:

- Majority voting fails, all checksums differ.
  - Two faults occur in different PEs during the *Run Application* stage. The time the system spends in this stage is be defined as t_p.
  - System fails to correct a fault and starts the next cycle with a fault already present at the start of the *Run Application* stage. In this case, if another fault would occur during the time t_p, a majority decision would be impossible to achieve.
- Majority voting leads to a wrong decision. This can happen when a second fault occurs after *Run Application* stage in the PE that is used for repairing the initial fault. This case is discussed in more detail below.

However, it is important to note that the STROBES algorithm can still recover from two simultaneous Type-*SF* faults, but only if the second fault occurs after the *Run Application* stage finishes. However, in this case, the system is not aware of the second fault because majority voting is performed on the checksums generated over the CP that is taken of the application's state at the end of the *Run Application* stage. Therefore, the second fault will be detected only during the next cycle, provided that the initial fault is repaired during the active cycle. The success of the initial fault's repair under this condition depends on in which PE the second fault occurred.



(a) Additional fault in PE 0 *(b) Additional fault in PE 1* *(c) Additional fault in PE 2*

Figure 29: Effects of a second Type-SF fault occurring during repair time. In the example, the initial fault was in PE 0

In order to better illustrate the system's transitions based on the occurrence of the second fault, a simplified set of three fault states are defined:

- S_0 – System is fault free.
- S_1 – There is a single fault in the system. This is a faulty, but repairable state.
- S_F – System is in failed state.

All subfigures in Figure 29 represent a situation where the STROBES algorithm attempts to repair an already existing fault, while a second fault occurs during the repair process. However, the behavior of the system is identical, if the fault occurs at any time after the *Run Application* stage.

Therefore, in Figure 29, the system is already in state S_1 at the start of each diagram. In all subfigures, the initial fault has occurred in *PE 0*. Then *PE 1* transfers its CP to *PE 0* in order to repair it. *PE 2* waits idle until the CP transfer is finished. However, it can be seen from the figure that if a second fault were to occur during the repair, the final state of the system after the CP transfer is finished depends on in which PE the additional fault occurred. As such, there are three possible outcomes, each represented by different subfigure of Figure 29:

- Figure 29a explains the case where an additional additional bit flip occurs during the CP transfer in the faulty PE, *PE 0*. In this case, as seen in Figure 29a, the system will still be repaired because the CP transfer causes the memory contents of the faulty PE to be overwritten and both faults to be corrected.
- Figure 29b represents the case where the additional Type-*SF* fault occurs in the PE that is performing the CP transfer. In the figure, *PE 1* transfers its CP to *PE 0* to correct the fault in *PE 0*. However, during the transfer, an additional Type-*SF* fault occurs in *PE 1*. The additional fault in *PE 1* will cause the transfer to fail and to transfer a faulty CP to *PE 0*. By the end of the transfer, there will be two faults in the system, thus the system will degrade into the failed state, S_F.
- Figure 29c illustrates the situation where an additional fault occurs in the idle PE, which is waiting for the CP transfer to finish, but is not actively participating in it. In this case, the system will stay in state S_1 after repairing the first fault. The additional fault will be detected and repaired during next cycle. In the figure, *PE 1* transmits its CP to the faulty PE, *PE 0*, while a fault occurs in *PE 2*, which is idle. After the transfer is finished, the fault in *PE 0* is corrected, but the newly appeared fault in *PE 2* is still active.

Therefore, the failure condition can be summarized as that a STROBES system fails if any of the two situations occur:

- Two Type-*SF* faults occur during the *Run Application* stage
- One Type-*SF* fault occurs during *Run Application* stage and another Type-*SF* fault after the *Run Application* stage in the PE that is responsible for transmitting its CP to the PE that suffered the initial fault.

5.6.2 Markov Model Definition

By knowing the failure conditions and behavior of the STROBES algorithm, it is possible to define a Markov model, which represents STROBES' fault states and the transitions between the states. The Markov model can be then used to calculate the mean time to failure (MTTF) of the STROBES system.

Additionally, to simplify the analysis, it is assumed that the system is properly synchronized (no Type-*TF* or Type-*NF* faults). Also, the Markov model introduced in this section does not consider the case where there are three simultaneously active Type-*SF* faults in the system. While theoretically a plausible situation, the probability

of three Type-*SF* faults occurring during the time of a single algorithm cycle is very low and including this case in the model would not influence the results analysis much, while still increasing the complexity of the model considerably.

Table 7: Definitions of the fault states for a STROBES system with three PEs

State	Description	Fault Situation
S_0	No Type-*SF* faults	Working
S_1	One Type-*SF* fault	Repairable
$S_{1,1}$	One Type-*SF* fault	Repairable
$S_{1,2}$	Two Type-*SF* faults	Repairable
S_F	Two or more Type-*SF* faults	Failed

To better model the different behavior of the STROBES algorithm under faults, five fault states have been defined in the Markov model. These fault states are shown in Table 7 and also explained in more detail below:

- **State** S_0 – Fault free state
- **State** S_1 – One fault has occurred during the processing and communication stages
- **State** $S_{1,1}$ – After the *Vote* stage, there is only a single fault in the system. It is important to note that this state represents the actual fault state of the system after voting, not the state that the system "thinks" it is in as the result of the majority voting.
- **State** $S_{1,2}$ – One fault has occurred during the *Run Application* stage of the algorithm (time t_p). However another fault occurred during the *Calculate Checksum*, *Checksum Communication* or *Vote* stage in the PE that is not involved in repairing the first fault. As such, there are two concurrent Type-*SF* faults in the system. However, since the second fault is in the PE that is not involved in repairing the initial fault, the system is still in a repairable state. Like state $S_{1,1}$, fault state $S_{1,2}$ also does not represent the state the system "thinks" it is in because of majority voting result. It represents the actual fault state of the system after the *Vote* stage.
- **State** S_F – Failed state. At least two faulty PEs exist in a system in an unrepairable configuration

This model can be visualized using a Markov chain, which can be seen in Figure 30. Transition rates between the states are explained in Table 8.

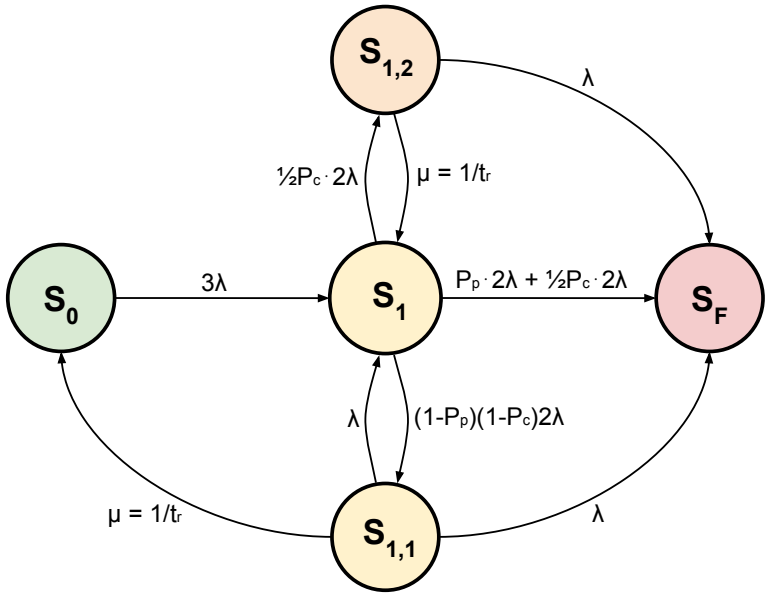


Figure 30: Diagram of a markov chain representing the STROBES algorithm's fault states and the transitions between them

Table 8: Markov model state transitions

Source State	Dest. State	Transition Rate	Comment
S_0	S_1	3λ	One fault occurs in any of three PEs
S_1	S_F	$P_p 2\lambda$	A second fault occurs during process execution
		$\frac{1}{2}P_c 2\lambda$	A second fault occurs during checksum transfer or voting in the PE that is needed for correcting the first fault
	$S_{1,2}$	$\frac{1}{2}P_c 2\lambda$	A second fault occurs during checksum transfer or voting in the PE that is *not* needed for correcting the first fault
	$S_{1,1}$	$(1-P_p)(1-P_c)2\lambda$	No additional fault occur during computation, checksum transfer and voting
$S_{1,1}$	S_0	$\mu = \frac{1}{t_r}$	All faults are repaired
	S_1	λ	A fault occurred during repair in the PE that is not involved in correcting the initial fault. As such, the system will continue executing the application with a single faulty PE
	S_F	λ	A fault occurred during repair in the PE that is involved in correction the initial fault
$S_{1,2}$	S_1	$\mu = \frac{1}{t_r}$	No additional fault occurred during repair. This results in one of the existing faults being repaired, while other fault will be repaired during the next cycle
	S_F	λ	An additional fault occurred during repair

Initially, the system is assumed to start up in a healthy state (S_0). However, a transition to state S_1 is made if a single Type-*SF* fault occurs during the *Run Application*, *Calculate Checksum*, *Checksum Communication* and *Vote* stages. This transition happens with fault rate of 3λ, where λ is the error rate that can be calculated

using Equation 5. The error rate depends on two parameters, the bit error rate (BER, in bit flips per second) and the size of the application memory that is protected by the STROBES algorithm, $size_{state}$. Since the fault can occur in any of the three PEs, the total amount of vulnerable memory area is three times larger than the state size of a single PE, hence the error rate for transitioning from state S_0 to state S_1 is 3λ.

$$\lambda = BER \cdot size_{state} \tag{5}$$

Once in state S_1, there are three transitions can be taken. The transition that is taken by the system depends on if a second fault occurs and if it does then where and when this happens.

Therefore, the system can transition from the state S_1 to either the failed state S_F, the repair state $S_{1,1}$, or to the repair state $S_{1,2}$. The conditions for these transitions are explained below.

- If a second fault occurs during the *Run Application* stage, it will always cause the majority voting to fail and the system transitions to state S_F.

  The probability P_p of a Type-*SF* fault occurring turing t_p, the time spent in the *Run Application* stage in the two remaining healthy PEs, is calculated using Equation 6. In the equation λ is the error rate, calculated using Equation 5. Because the faults can occur in any of the two fault-free PEs, lambda is multiplied by two. By using the probability P_p as a weight for the error rate, a transition rate of $P_p 2\lambda$ is derived.

- However, the second fault could also occur during time the algorithm is in the Calculate Checksum, *Checksum Communication* and *Vote* stages. The time spent in these stages, t_c, is calculated using Equation 7. The first component of the equation is the time spent on generating checksum over the CP ($\frac{size_{state}}{rate_{chksum}}$), where $size_{state}$ (in bits) is the CP size and $rate_{ckhsum}$ (in b/s) is the rate at which the checksum generation performed.

  Checksum transmission time ($\frac{size_{checksum} \cdot O_c}{BW}$) consists of the size of transmitted data, (the size of the checksum) in bits, multiplied by communication overhead O_c. The result is then divided by the network bandwidth BW, measured in b/s. The communication overhead O_c is a correction multiplier, which represents the additional data added by the communication protocol.

  The time spent in *Vote* state is not included in the equation since majority voting over three 16 or 32-bit checksum values happens nearly instantaneously in all modern systems.

  The P_c of a fault occurring during time t_c is calculated using Equation 8, where 2λ is the error rate over the states of the two working PEs.

  However, in case a second fault occurs during time t_c, the second fault is not detected by majority voting. In this case, state into which the system transitions to depends on in which PE the second fault occurs, as explained by the example in Figure 29. In total, in this case, two transitions can be taken, depending in which PE the second fault occurred in.

  - If the additional fault occurs in the PE that is responsible for repairing the initial fault (the case shown in Figure 29b), the system will move to the failed state S_F, since repair is not possible.

– However, if the second fault occurs in the PE that is not involved in the process of correcting the initial fault (Figure 29c), it is still possible to repair the fault. As such, the system will transition to the state $S_{1,2}$.

The two cases have an identical occurrence probability. For each transition, the probability is $\frac{1}{2}P_c$. Therefore, if a second fault occurred during time t_c, the transition rate for both transitions from S_1 to S_F and from S_1 to $S_{1,2}$ is $\frac{1}{2}P_c 2\lambda$.

- Finally, if there are no additional faults during the *Run Application, Calculate Checksum, Checksum Communication* and *Vote* stages, the system will move to state $S_{1,1}$. The probability of this is the probability that a second fault does not occur during the time t_p *and* that the second fault does not also occur during the time t_c, in any of the two PE that have remained fault-free. Therefore, the resulting error rate is $(1-P_p)(1-P_c)2\lambda$.

$$P_p = 1 - e^{-2\lambda \cdot t_p} \tag{6}$$

$$t_c = \frac{size_{state}}{rate_{chksum}} + \frac{size_{checksum} \cdot O_c}{BW} \tag{7}$$

$$P_c = 1 - e^{-2\lambda \cdot t_c} \tag{8}$$

In both states, $S_{1,1}$ and $S_{1,2}$, a repair is attempted. If only a single fault occurred during the *Run Application*, *Calculate Checksum*, *Checksum Communication* and *Vote* stages and the system is in state $S_{1,1}$, there are ia total of three possible transitions that can be taken. Each of them is explained in more detail below.

- **Repairing of the fault and returning to the fault free state S_0.** Clearly, the system will be repaired and return to state S_0 when no additional Type-*SF* faults occur during the CP transmission time t_r. This can happen with repair rate $\mu = \frac{1}{t_r}$.
- **Moving back to state S_1.** When a single Type-*SF* fault occurs during the time t_r in the idle PE, which is not involved in the CP transfer, the initial fault will be corrected, but the newly-occurred fault will stay in the system. As such, the system will continue operating with one faulty PE until the fault is detected during the *Vote* stage during the next cycle. It can then be corrected afterwards. Therefore, in this case, the system will transition to state S_1. This case is illustrated in Figure 29c. It happens with error rate λ, as it is equal to a Type-*SF* fault occurring in exactly one PE during time t_r.
- **Degradation into state S_F.** The system can further degrade into the failed state S_F when a single Type-*SF* fault occurs during recovery in the PE that is transmitting its CP to the faulty PE (Figure 29b). This causes the transfer to fail and a situation where there are two Type-*SF* faults in the system after the CP transfer. This can also happen with fault rate λ, since the additional Type-*SF* fault occurs in exactly one PE.

$$t_r = \frac{size_{state} \cdot O_c}{BW} \tag{9}$$

If the system is in state $S_{1,2}$, there are two outcomes. If one of the faults gets repaired, then the system moves to state S_1. However, should there be another fault during the repair, the system will enter the failed state S_F.

- **Repairing the first fault and returning state S_1.** If the system is in state $S_{1,2}$, then there are already two faults in the system. However only a single fault can be repaired at a time since if the system is in state $S_{1,2}$, it knows only about the existence of the first fault. However, in this state, repair is still possible because the second fault occurred in the PE, which is not involved in the process of repairing the first fault (the situation illustrated in Figure 29c). Therefore, a successful repair in state $S_{1,2}$ results in the system continuing the execution of the application with one faulty PE. The remaining fault will be detected during the *Vote* stage of the next cycle, after which, also the remaining fault can be corrected.

  Therefore, if the repair of the initial fault is successful, the system will move to state S_1. This transition is taken with repair rate $\mu = \frac{1}{t_r}$, where t_r is the time to repair. Time to repair is, in case of STROBES, equal to the time required for transferring the correct CP to the faulty PE. t_r can be calculated using Equation 9.

- **Degradation into state S_F.** However, if another Type-*SF* fault occurs during the repair, the repair will fail, and the system will transition to the failed state S_F. This transition happens with fault rate λ, since there is exactly one PE, where the additional fault could occur.

5.6.3 Reliability Calculation

The reliability $R(t)$ of the system is defined as the probability that the system is working correctly at a time interval $[0,t]$ [46]. In the context of a Markov process, reliability can be redefined as the probability of a system being *only* in the non-failed states (S_0, S_1, $S_{1,1}$, or $S_{1,2}$) during the time interval $[0,t]$. However, the probability of the system being in only those states during time interval $[0,t]$ is equal to the probability that system does *not* enter the failed state (S_F) during that time interval, which is easier calculate.

Generally, the probabilities of a system being in a fault state S_n at time t can be calculated using Equation 10. In the equation, M is the fault state transition matrix and $P(t)$ is a vector of elements $P_n(t)$ ($n \in 1 \dots 3$). P_n represents the probability of the system being in state S_n at time t. The transition matrix M can derived from the Markov chain, shown in Figure 30. Knowing the transition matrix M, the matrix multiplication can be expressed as in Equation 11. This representation is equivalent to a system of ordinary differential equations presented in Equation 12.

$$\frac{\partial}{\partial t} P(t) = M \cdot P(t) \tag{10}$$

$$\begin{bmatrix} \frac{\partial}{\partial t} P_0(t) \\ \frac{\partial}{\partial t} P_1(t) \\ \frac{\partial}{\partial t} P_{1,1}(t) \\ \frac{\partial}{\partial t} P_{1,2}(t) \\ \frac{\partial}{\partial t} P_F(t) \end{bmatrix} = \begin{bmatrix} -3\lambda & 0 & \frac{1}{t_r} & 0 & 0 \\ 3\lambda & -P_p 2\lambda - P_c 2\lambda - (1-P_p)(1-P_c)2\lambda & \lambda & \frac{1}{t_r} & 0 \\ 0 & (1-P_p)(1-P_c)2\lambda & -2\lambda - \frac{1}{t_r} & 0 & 0 \\ 0 & \frac{1}{2} P_c 2\lambda & 0 & -\lambda - \frac{1}{t_r} & 0 \\ 0 & P_p 2\lambda + \frac{1}{2} P_c 2\lambda & \lambda & \lambda & 0 \end{bmatrix} \cdot \begin{bmatrix} P_0(t) \\ P_1(t) \\ P_{1,1}(t) \\ P_{1,2}(t) \\ P_F(t) \end{bmatrix} \tag{11}$$

$$\begin{cases} \frac{\partial}{\partial t}P_0(t) = -3\lambda P_0(t) + \frac{1}{t_r}P_{1,1}(t) \\ \frac{\partial}{\partial t}P_1(t) = 3\lambda P_0(t) - (P_p 2\lambda + P_c 2\lambda + (1-P_p)(1-P_c)2\lambda)P_1(t) + \lambda P_{1,1}(t) + \frac{1}{t_r}P_{1,2}(t) \\ \frac{\partial}{\partial t}P_{1,1}(t) = (1-P_p)(1-P_c)2\lambda P_1(t) - (2\lambda + \frac{1}{t_r})P_{1,1}(t) \\ \frac{\partial}{\partial t}P_{1,2}(t) = \frac{1}{2}P_c 2\lambda P_1(t) - (\lambda + \frac{1}{t_r})P_{1,2}(t) \\ \frac{\partial}{\partial t}P_F(t) = (P_p 2\lambda + \frac{1}{2}P_c 2\lambda)P_1(t) + \lambda P_{1,1}(t) + \lambda P_{1,2}(t) \end{cases} \tag{12}$$

In order to find the probability of the system being in the failed state S_F at time t, or the probability $P_F(t)$ in Equation 12, the system of equations was solved symbolically using MATLAB [124]. The initial conditions for solving the equation were set to $P_0(0) = 1,\ P_1(0) = 0,\ P_{1,1}(0) = 0,\ P_{1,2}(0) = 0,\ P_F(0) = 0$, representing that the system was fully functional at time $t = 0$. Next, reliability $R(t)$ can be calculated as the probability of a system not being in a faulty state at time t, as shown in Equation 13.

$$R(t) = 1 - P_F(t) \tag{13}$$

However, a more intuitive metric than reliability itself is the Mean Time To Failure (MTTF). In case of a system that cannot be repaired, MTTF is defined as the mean time from system's startup until its failure [46]. In case a STROBES system, it can be defined as the mean time from system's startup until the time the system ends up in the faulty state, S_F. Mathematically, MTTF is the area under the reliability curve [46], represented as an improper integral with limits from zero to infinity, as shown in Equation 14. However, as proven in [125], for faster calculation, MTTF can computed using the Laplace transform. The equations used for MTTF calculations can be seen in Equation 14, where $R^*(s)$ represents the Laplace transform of the reliability $R(t)$ function.

$$MTTF = \int_0^\infty R(t)dt = \lim_{s \to 0} R^*(s) \tag{14}$$

However, in order to solve the equation numerically, values for λ, t_r, and the probabilities P_p and P_c need to be calculated, which depend on many constants. The values that are used for the constants in this thesis are shown in Table 9. The constants roughly estimate an onboard computer system based on 32-bit 40MHz PEs, connected over the SpaceWire [126] network. The fault rate of 10^{-7} bitflips/s is an approximate for a satellite located in low Earth orbit. The values for these constants were provided by DLR. Finally, the results were acquired by numerically solving the MTTF equation (Equation 14) in MATLAB.

Table 9: Constants used for reliability estimation

Constant	Value	Comment	Description
BER	10^{-7} bitflips/s	BER for low Earth orbit	Bit error rate
$rate_{chksum}$	$8 \cdot 10^7$ b/s	40MHz 32-bit CPU	Checksum generation rate
O_c	1.28 times	Est. for SpaceWire	Transmission overhead
BW	50 Mb/s	Est. for SpaceWire	Network bandwidth

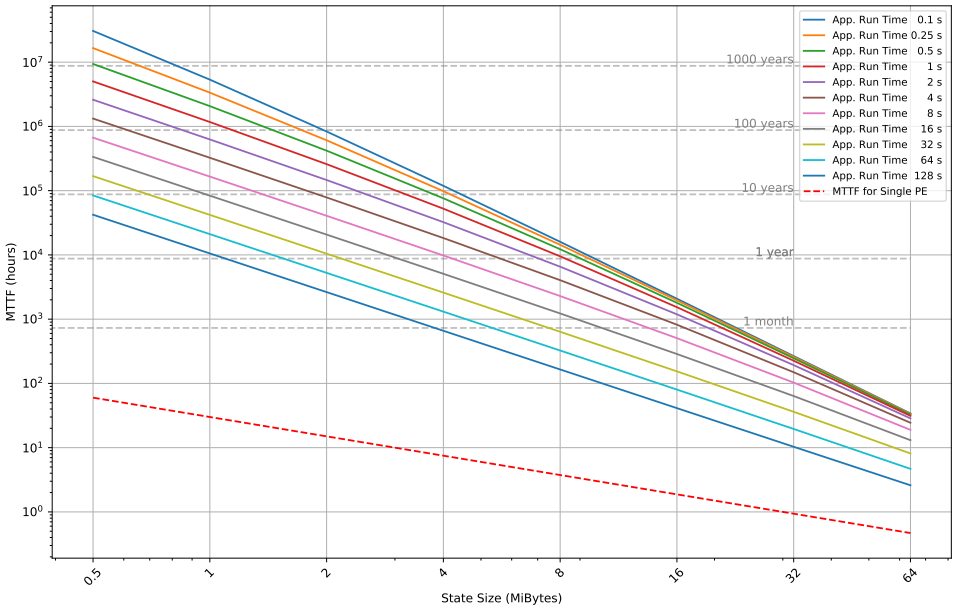


Figure 31: MTTF of STROBES algorithm for different state sizes and application running times per cycle

For comparison, MTTF was also calculated for an unprotected application, running on a single PE. Because the unprotected system cannot be corrected, it will fail whenever a Type-*SF* fault occurs. Additionally, since there is only a single PE, it will fail with failure rate λ.

The MTTF of the unprotected system can be calculated as the inverse of its failure rate λ (Equation 15). This is a well-known simplification [46] of the reliability calculation procedure described earlier for the STROBES system, which applies to simple systems without any repair mechanisms or redundancies.

$$MTTF_{unprot} = \frac{1}{\lambda} \tag{15}$$

The MTTF for the STROBES algorithm for different combinations of state size and application run time per cycle have been visualized Figure 31. Both axes of the figure are logarithmic. The dashed red line represents the unprotected system with a single PE.

As illustrated in Figure 31, the STROBES algorithm works the best for the shorter application run times per cycle and for smaller state sizes. This is caused by the fact that the larger state size not only translates into a longer CP transmission time t_r, but also more time is required to calculate a checksum over a larger state. This, however, further increases the chance of a Type-*SF* fault occurring during a cycle.

Application run time per cycle influences MTTF the same way; the shorter it is, the more often the voting happens, thus reducing the chance of two simultaneous Type-*SF* faults in a system.

Additionally, state size influences the MTTF of the algorithm much more than the application runtime. This observation is in accordance with the model, since increasing the state size n times also increases the failure rate λ by n times. However, increasing

the application running time per cycle n times, does not result in similar increase in the failure rate.

To conclude, STROBES can be reliably used to protect applications with a relatively small state size of under four MiB. However, this is enough for quarantining fault tolerant operation of most control applications. To provide error correction on the data produced by the applications, other methods, such as ECCs, can be used.

Additionally, it can be seen from Figure 31 that shorter application running times per cycle (t_p) provide better fault tolerance. This is because shorter t_p values also result in more frequent error checking. As such, there exists a tradeoff between applications performance and fault tolerance and the value chosen for t_p depends also on the timing requirements of the protected application. However, it is advised to minimize t_p for improved reliability, especially for applications with larger states.

5.7 Simulator

The STROBES algorithm is application agnostic. Protecting an application can be done by just installing a lightweight wrapper to be around the application. Moreover, as explained in previous sections, the performance of the STROBES algorithm only depends on three parameters:

- **Application running time per algorithm cycle.** The shorter the application running time, the more often are synchronization and voting performed. This results in higher MTTF, as explained in Section 5.6. On the other hand, shorter application running time also results in poorer application performance, since the system spends less time on running the application and more time on synchronization, compared to a longer application run time.
- **Size of the application's memory state (CP size).** Larger CP size results in lower MTTF value, As explained in Section 5.6. This happens because the fault sensitive memory area is larger. However, this is a parameter that depends solely on the application. In order to optimize the fault handling capability of STROBES, applications with larger CP size need to run the check more often. This dependence between application run time and CP size is illustrated well by Figure 31.
- **Checksum waiting timeout.** Longer timeout results in better of Type-*TF* fault handling capability. However, a longer timeout value can also decrease the system's performance. While the minimum timeout value is specified by Equation 4, the optimal value depends on different inherent delays in the system.

For this reason, the performance of a STROBES system can be analyzed in a simulated environment. However, it can be also used to perform design space exploration for finding parameters, which result in the best performance. The design space exploration can be done in parallel with the application development and needs only be done once – when changes are made to the application during development, the performance of the STROBES algorithm will usually not change. An exception to this rule is when the CP size or application's running time per algorithm cycle are considerably changed. In addition to the three application and STROBES-related parameters, specified above, the simulation also needs to have information on the worst-case timing parameters of the system the STROBES algorithm is run on:

- **Checksum calculation rate.** The rate at which the checksum is calculated ($rate_{checksum}$ in Table 9).
- **Data transmission overhead factor.** The extra bits added to the sent data by communication protocol, shown as O_c in Table 9
- **Network bandwidth.** The speed at which transfer data transfer is performed. Shown as BW in Table 9

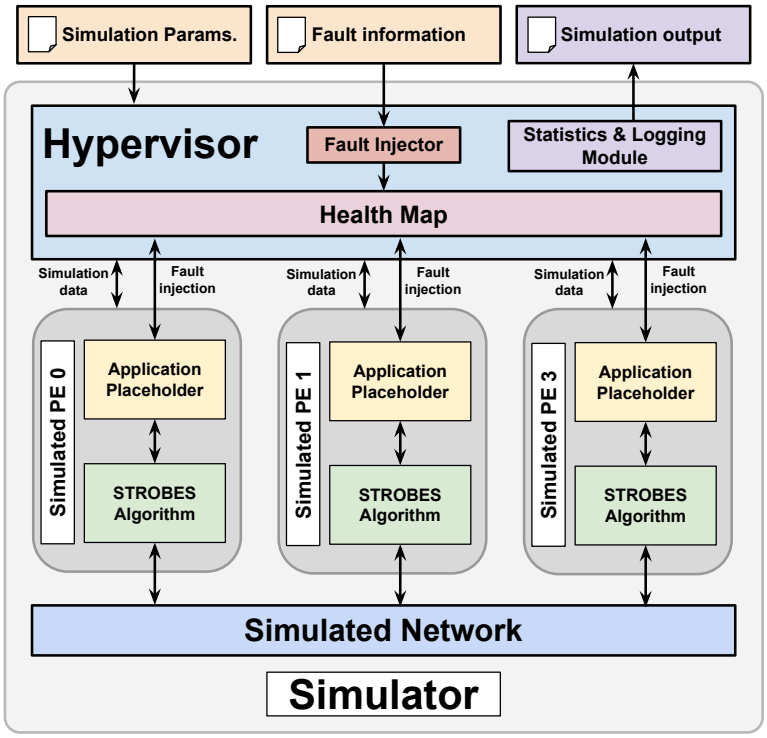


Figure 32: Architecture of the STROBES simulator

Architecture of the simulation tool is shown in Figure 32. The simulator takes in the simulation parameters that is explained above and fault injection information and produces the simulation output. Simulation output data contains statistical information, such different performance parameters of the algorithm, which can be used to generate the results in Section 5.8. Additionally, the simulation output also contains logs and other debugging information.

The simulation tool runs three simulated PEs, connected to simulated network with broadcast support. Each PE runs independently the STROBES algorithm, where the application is replaced by a time delay, combined with a fault information of its CP. This approach is possible since the STROBES algorithm is application agnostic. Its performance does not depend on the application's functionality, but only the on the time the application is run per one algorithm cycle.

Since the application is emulated by a delay, the application also does not have a real state, any real memory allocated to it for protection. However, while calculating the checksum or transmitting the CP to a faulty PE, the simulation tool also considers the

state size, which is specified in simulation parameters. For example, it will take longer to transmit a larger CP. This level of simulation of the application state is sufficient since the exact fault location inside the application state is not important for STROBES. The STROBES algorithm uses checksums over the entire CP for determining if the CP (and, as extension, the application's state) is faulty by comparing it to the checksums received from other PEs. It is assumed that a checksum over a faulty CP will always differ from a checksum taken over a correct CP.

For this reason, the fault information of the state is stored as a binary value in the system health map, located in the hypervisor. The hypervisor monitors the fault information of the states of each PE and, if needed, also performs fault injection. However, hypervisor's functions cover more than only fault handling. The hypervisor is also responsible for setting up, starting and finally stopping the simulation. Because the hypervisor can monitor all components involved in the simulation, it also logs the simulation data and performs statistics for the simulation. The logging functionality is implemented inside the hypervisor since it has a bird's eye view of the entire simulation. The hypervisor always knows the exact state of all the PEs. It knows if all the PEs are synchronized or not and the fault status of all the PEs, at any given time during the simulation.

The event-based simulation is run in discrete time, with the resolution of $1\ ms$. This resolution is high enough for modelling possible delays while still guaranteeing acceptable simulation performance. The time for checksum calculation and CP transfer is calculated the similarly to Section 5.6. More specifically, the time spent on checksum calculation ($t_{chkCalc}$) is calculated using Equation 16. In the equation $size_{cp}$ represents the size of the CP over which the checksum is calculated, in bits and $rate_{chksum}$ is the speed of checksum calculation, in b/s.

$$t_{chkCalc} = \frac{size_{cp}}{rate_{chksum}} \tag{16}$$

The time spent on transmitting a message, such as a checksum or a CP ($t_{msgTrans}$), is calculated using Equation 17. In the nominator is the amount of data to be transmitted, consisting of the message size ($size_{msg}$) in bits, multiplied by communication overhead factor O_c. The communication overhead factor is a parameter related to the communication protocol and represents the additional bits introduced by the protocol. For example, the overhead factor for the Spacewire protocol is 1.28. Spacewire is a communication protocol that is often used in for communication between PEs in satellites. The final data size in bits is then divided by network bandwidth BW, in b/s.

$$t_{msgTrans} = \frac{size_{msg} \cdot O_c}{BW} \tag{17}$$

5.8 Experimental Results

In order to evaluate its capabilities for fault correction, the STROBES algorithm was simulated using the simulation tool introduced in Section 5.7. During the simulations, the time that the STROBES algorithm spent on correcting the faults was recorded. It is important to note that when timing-related faults (Type-*TF* and Type-*NF*) where injected the timer was started when the PE that had the timing fault injected was ready to synchronize with the rest of the system. The timer was stopped once all PEs were completely synchronized and their state contents were identical. As such, only the additional time spent by the STROBES algorithm correcting the fault is considered, not the delay caused by the fault. Additionally, since the simulation tool uses its own internal simulation time, the results do not depend on load of the computer it is running on.

Algorithm 9 An algorithm, which explaining the set of experiments that were run for acquiring the experimental data

```
1: for faultType in list[typeSF, typeTF, typeNF] do
2:     for appRunTime in list[100, 250, 500] do
3:         for stateSize in list[0.5, 1, 2, 4] do
4:             for chksumTimeout in range(appRunTime + 1, appRunTime 3) do
5:                 if faultType == typeSF then
6:                     // In case of Type-SF fault, do not inject delay
7:                     listOfDelayFaults = [0]
8:                 else // Type-TF or Type-NF faults
9:                     listOfDelayFaults = range(0, appRunTime 3)
10:                for delayFault in listOfDelayFaults do
11:                    sim.reset()
12:                    sim.run(appRunTime, stateSize, chksumTimeout,
13:                            injectFault=faultType, delayFaultTime=delayFault)
14:                    sim.saveResult()
```

In order to generate the experimental results provided in this section, hundreds of thousands of experiments were run with different parameter sets for each of the three fault types (Type-*SF*, Type-*TF* and Type-*NF*). The application running time values per cycle of 100 ms, 250 ms and 500 ms and state sizes of 0.5 MiB, 1 MiB, 2 MiB and 4 MiB were used during the simulation. Those values are in the range of the most realistic applications to be protected by STROBES, since according to Figure 31, they result in high MTTF values.

Additionally, each combination of state size and application running time was simulated with a range of checksum waiting timeout values, ranging from $(t_p + 1)\ ms$ to $(3 \cdot t_p)\ ms$, where t_p is the application running time per cycle. Moreover, in case of Type-*TF* and Type-*NF* faults, the amount of delay was ranged from $0\ ms$ to $(3 \cdot t_p)\ ms$. The set of experiments is visualized using Algorithm 9. Finally, the simulation results were collected, and the worst case and average correction times were extracted from the results for each fault type.

5.8.1 Type-SF faults

To benchmark the Type-*SF* fault correction performance of the STROBES algorithm, during each simulation a Type-*SF* fault was injected. The injection was performed by force-writing a wrong value into the memory of one PE. The timer that was used for fault correction time measurements was started then the faulty PE tried to re-synchronize with the rest of the system. The timer is started and stopped by the simulator's hypervisor (see Section 5.7). The timer was stopped after the fault was corrected using CP transfer and the system had been re-synchronized.

State Size (MiB)	Application Run Time (ms)	Fault Correction Time (ms)
0.5	100	16
1	100	29
2	100	55
4	100	106
0.5	250	16
1	250	29
2	250	55
4	250	106
0.5	500	16
1	500	29
2	500	55
4	500	106

Table 10: Type-SF fault correction times for different application running times and state sizes

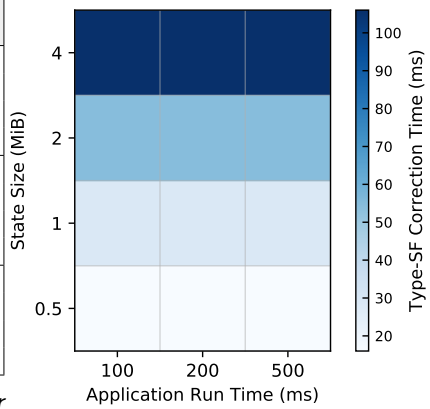


Figure 33: Type-SF fault correction time

Results of the Type-*SF* fault injection experiment can be seen Table 10. The results are further visualized in Figure 33. From the data it is visible that, as expected, the correction time of for Type-*SF* faults depends only on the state size. This makes sense, since the larger the state, the more time must be spent on transferring the correct CP to the faulty PE.

5.8.2 Type-TF faults

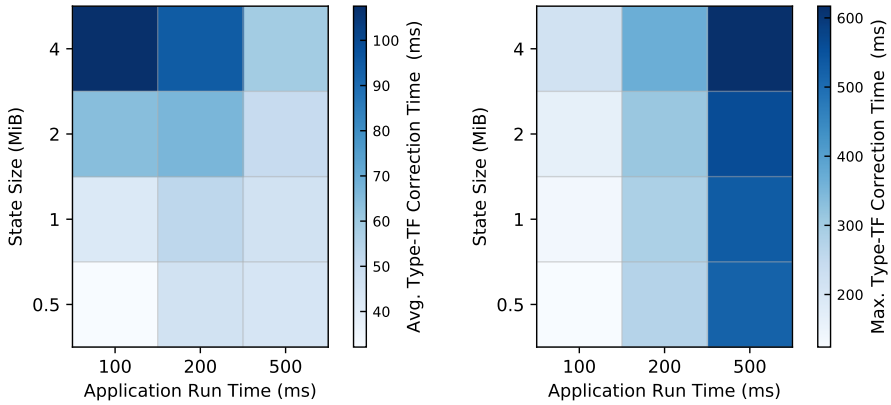
Next, to profile the STROBES algorithm's performance under Type-*TF* faults, injections of the timing faults were done. For simulating a Type-*TF* fault, the application execution time t_p of the PE suffering from the fault was extended by the duration of the timing fault during the simulation. The fault correction time was measured from the point when all PEs detected that one PE is not synchronized to until the point where all PEs were re-synchronized. The fault injection experiments were carried out exhaustively for all Type-*TF* faults in the range of $0\ ms$ to $(3 \cdot t_p)\ ms$.

The average and maximum correction times for Type-*TF* faults can be seen in Table 11 and in Figure 34. It is important to note that the scale for the color axes for Figure 34a and Figure 34b is different. It can be seen from the data that there is a large difference in the average and Type-*TF* maximum fault correction times.

In Figure 34a, the fault correcting delay is the highest for the shortest application running times. At a first glance, this somewhat counter-intuitive. However, it is important to keep in mind that, as explained in Figure 27 and Figure 28 in Section 5.5 of this chapter, Type-*TF* faults can only be corrected without performing a CP transfer

Table 11: Minimum, average and maximum Type-TF fault correction times for different application running times and state sizes

State Size (MiB)	Application Run Time (ms)	Average (ms)	Max (ms)
0.5	100	32.13	124
1	100	42.49	137
2	100	64.46	165
4	100	107.57	220
0.5	250	46.47	273
1	250	52.90	286
2	250	67.00	314
4	250	94.71	370
0.5	500	44.72	521
1	500	46.33	534
2	500	50.67	562
4	500	59.21	617



(a) Average Type-TF fault correction time

(b) Maximum Type-TF fault correction time

Figure 34: Dependance of Type-TF fault correction time on application running time per cycle and state size. Please note the difference in scale for the color axis for the subfigures.

if the length of the fault is shorter than $t_p + chksum\_timeout$. t_p is the application running time per cycle. In order to correct Type-*TF* faults, which last longer, a CP transfer is necessary. As such, a shorter t_p results in a higher probability that the system cannot be synchronized without a CP transfer. When the CP transfer is performed, obviously, as observed about correcting Type-*SF* faults, transferring a larger CP constitutes to a longer correction time.

The maximum Type-*TF* fault correction times visualized in Figure 34b is quite different from the average correction times. The maximum Type-*TF* fault correction time depends heavily on the application running time. The maximum correction times occur when the length of the fault is slightly longer than $t_p + chksum\_timeout$. In this case, the timeout passes, and the other two non-faulty PEs will continue execution of the next cycle without the faulty PE. Even if the faulty PE tries to synchronize right after the timeout ends, it cannot do so before the next synchronization phase of the

STROBES algorithm. The synchronization phase occurs after the non-faulty PEs have finished their *Run Application* stage. Therefore, the longer is t_p, the longer the faulty PE needs to wait to be synchronized.

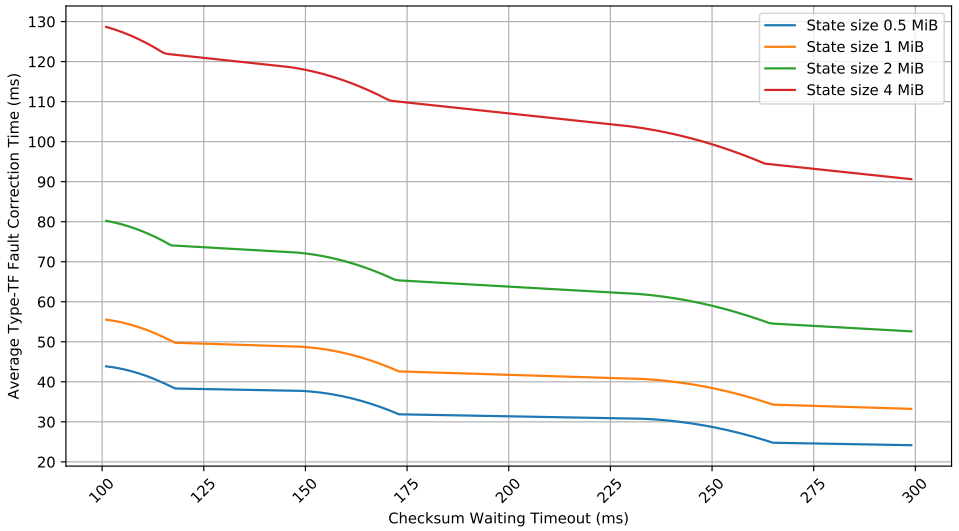


Figure 35: Dependance of the average Type-TF fault correction time on checksum waiting timeout. Application running time per cycle is fixed to 100 ms

Figure 35 shows the dependance of the average Type-*TF* fault correction time on checksum waiting timeouts. All experiments in the figure are shown for application running time of 100 ms.

As seen in Figure 35, the average Type-*TF* fault correction time reduces with higher checksum waiting timeout values. This can be explained by the fact that a longer checksum waiting timeout increases the probability that the STROBES algorithm can correct the fault without CP transfer. Additionally, it reduces the probability of the worst-case scenarios that are visualized in Figure 34b.

5.8.3 Type-NF faults

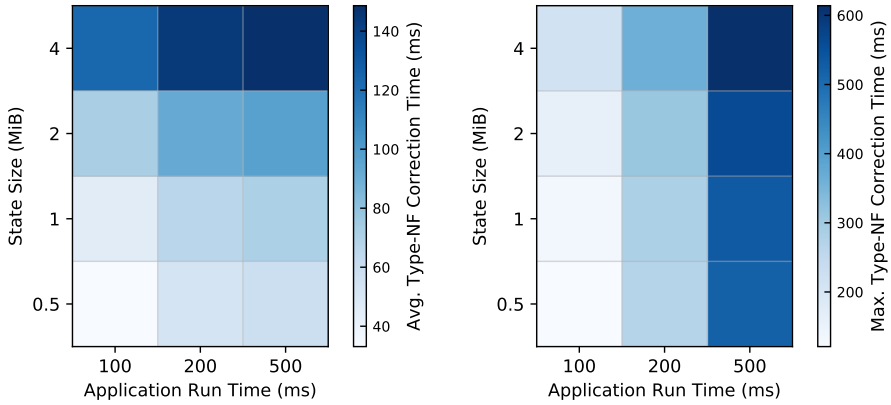
Type-*NF* faults represent the situation when the faulty PE is completely out of the sync with the rest of the STROBES system. This can happen either due to a communication problem, reset of a PE, etc. Type-*NF* fault is essentially a situation where the faulty PE has both a Type-*TF* fault and a Type-*SF* fault simultaneously. This results in the system always performing a CP transfer, no matter how large the timing fault component is. This is in contrast with Type-*TF* faults where CP transfer is not needed if the fault is shorter than $t_p + chksum\_timeout$.

The results of Type-*NF* fault correction experiments can be seen in Table 12 and are visualized in Figure 36. Since Type-*NF* faults are basically a combination of Type-*SF* and Type-*TF* faults, the fault correction time has a large correlation with correction times for both fault types.

For example, as seen in Table 12, the minimum Type-*NF* fault correction times are equal to the correction times of a Type-*SF* fault correction times, seen in Table 10. This is because the minimal Type-*NF* fault correction time only occurs in exceptional case, when the timing component of the Type-*NF* fault is zero. In such circumstances,

Table 12: Minimum, average and maximum Type-NF fault correction times for different application running times and state sizes

State Size (MiB)	Application Run Time (ms)	Min (ms)	Average (ms)	Max (ms)
0.5	100	16	33.10	121
1	100	29	46.10	134
2	100	55	72.36	161
4	100	106	123.89	214
0.5	250	16	53.33	270
1	250	29	66.32	283
2	250	55	92.58	310
4	250	106	144.08	364
0.5	500	16	58.25	521
1	500	29	71.25	534
2	500	55	97.37	561
4	500	106	148.63	614



(a) Average Type-NF fault correction time

(b) Maximum Type-NF fault correction time

Figure 36: Dependance of Type-NF fault correction time on application running time per cycle and state size. Please note the difference in scale for the color axis for the subfigures.

only a normal CP transfer is performed, as additional synchronization is not needed.

Additionally, the maximum correction time (Figure 36b) for Type-*NF* faults is similar to the maximum correction time of Type-*TF* faults. This is again caused by the fact that the reasons behind worst case synchronization time situations for both fault types are the same – the faulty PE cannot be synchronized during the same cycle it becomes available. Additionally, because the correction of extreme case of Type-*TF* faults which produce the maximum fault correction times involves requires CP transfer, also the fault correction methods are the same

However, the average fault correction times (Figure 36a) remain different for Type-*NF* faults, compared to other fault types. The correction time mostly depends on the state size. This is so because during the correction of Type-*NF* faults, a CP transfer is performed always.

The slight increase in Type-*NF* fault correction time as the application running time increases comes from the simulations that do not manage to synchronize during the first cycle (such as the maximum correction time experiments). This increases the average correction time slightly for longer application running times. The same pattern can be seen also in Type-*TF* fault correction data, but it is only visible for experiments run with smaller state size. For larger state sizes, the anomaly caused by conditional CP transfer in Type-*TF* correction data masks that pattern.

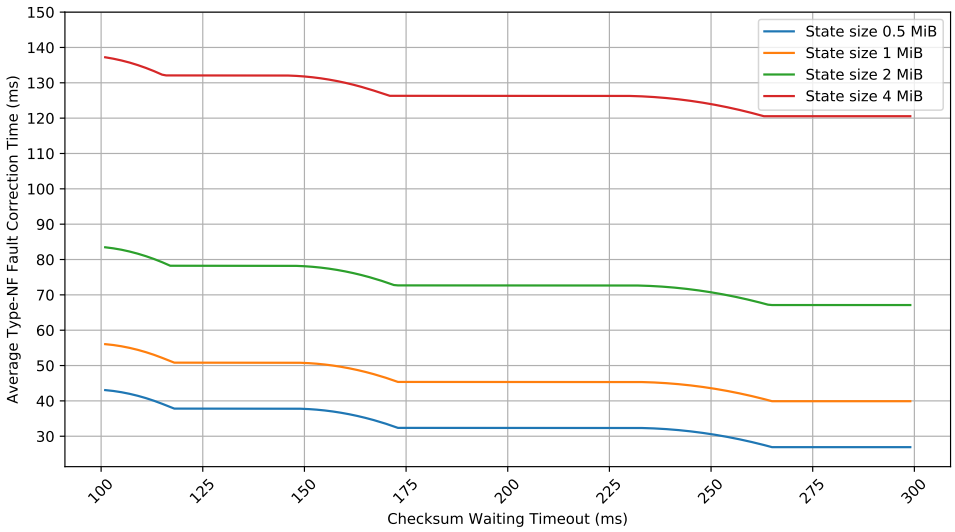


Figure 37: Dependance of the average Type-NF fault correction time on checksum waiting timeout. Application running time per cycle is fixed to 100 ms.

The dependence of the average Type-*NF* correction time on checksum waiting timeout for application running time of 100 ms per each cycle, can be seen in Figure 37. When compared to the same data for Type-*TF* correction times (Figure 35), it can be seen that the patterns are very similar. However, for Type-*NF* faults, the average correction times are slightly higher and are not influenced as much by the checksum waiting timeout, as for Type-*TF* faults.

This difference can once again be explained by the conditional CP transfer during the correction of Type-*TF* faults. For Type-*TF* faults, a longer checksum waiting delay increases the chance of a de-synchronized PE to be re-synchronized without CP transfer, thus lowering the average correction time. However, during the correction of Type-*NF* faults, CP transfer is performed in any case and it does not depend on the synchronization time of the PE. This makes the effects of a longer checksum waiting timeout much less pronounced.

5.8.4 Additional Observations

Some additional observation about the optimal set of parameters for the STROBES algorithm can be made based on the experiments conducted in this section. The performance of the STROBES algorithm depends on five parameters: size of the application's state, application running time per cycle, checksum waiting timeouts and faults in the system.

Obviously, the faults are not under the control of the designer of such system. The same could be said for the application's state size, which largely depends on the application's functionality. However, the other parameters, application running time per cycle and checksum waiting timeout, can be optimized to increase the STROBES system's fault correction performance.

Application running time: As discussed in Section 5.6, the application running time per cycle should be kept short. This increases the frequency of majority voting, which results in an increase in system's reliability. However, the results provided in Section 5.8.2 suggest that coupling very short application running intervals with a larger state size can have negative effect on Type-*TF* fault correction performance. As seen in Figure 34a, if the state smaller than 2 MiB, using application running time of 100 ms is probably the best option. However, if the state size is equal or exceeds 2 MiB, increasing the application running time to 200 ms results in better average Type-*TF* fault correction time.

Checksum waiting timeout: Equation 4 in Section 5.5 gives the minimal values for the checksum waiting time. It should be longer than the application running time and cover worst-case network delays. There is no upper limit to checksum waiting timeout. It is seen in Figure 35 that a longer checksum waiting timeout results in faster Type-*NF* correction. However, in case of a PE failure, a longer timeout has a negative effect on the system's performance because the non-faulty PEs will wait for the failed PE to re-synchronize for the full length of the timeout every cycle. For this reason, a very long checksum waiting time is also not reasonable.

5.9 Chapter Conclusions

In this chapter, a software-level fault tolerance approach, called STROBES, was proposed. As a completely software-based approach, STROBES can be used to provide fault tolerance for applications running on unmodified, non-fault tolerant, COTS hardware. The benefit of COTS components is that they are cheaper and faster than specialized fault tolerant components.

Additionally, since STROBES uses a high-level fault model, it is application agnostic. Its performance does not depend on the functionality of the protected application, but only on the worst-case timing constraints of the system it is run on. The parameters can be found formally and fine-tuned through high-level application-independent simulation. In this chapter, the architecture of the simulation tool was also provided.

In order to make an application work with STROBES, only a light-weight wrapper needs to be added. Consequently, it is easy to add STROBES to existing applications and the application can also be seamlessly updated after deployment without making any changes to STROBES' behavior or code. In contrast, most similar software-based fault tolerance mechanisms are limited to a very specific class of applications or require an extensive modification of the hardware or the operating system.

Fault tolerance is achieved in STROBES by running three versions of the same application simultaneously on three PEs. Faults are detected using distributed majority checking. STROBES can detect and correct errors in application state and handle timing errors and network problems.

The reliability of the STROBES algorithm was formally proven using Markov analysis. The analysis gave further insight into the capabilities and limitations of the algorithm. STROBES works best with applications whose state size remains small, under four MiB. Additionally, STROBES has a higher MTTF when shorter application running times are used. This makes sense since a shorter application running time per cycle results in

more frequent fault checking and repair, thus reducing probability of multiple Type-*SF* faults occurring during a cycle.

The information gathered from the Markov analysis was further analyzed in simulated fault injection experiments. In the experiments, the STROBES algorithm was run with a range of parameters under different types of faults. The analysis of the simulation results yielded the following conclusions about selecting the most optimal values for the two tunable parameters of the STROBES algorithm:

- **Application running time per cycle.** Generally, shorter application running times per cycle improve the fault tolerance, as it results in more frequent error checking. However, the experiments show that very short application running times per cycle can increase the time that is required for correcting timing errors. This is especially visible in cases where the application's state size is equal to or exceeds 2 MiB. In this case, the application running time per cycle of 200 ms results on average a faster timing error correction than application running time of 100 ms. However, it is also important to note that increasing the application running time will also decrease the MTTF of the system. Therefore, there is a trade-off between timing fault correction speed and the reliability of the system. The selection of the optimal value of application running time per cycle depends largely on timing and reliability constraints and requirements imposed on the system by the specification of the application.
- **Checksum waiting timeout.** The second parameter, the checksum waiting time, influences not only the correction times of the Type-*TF* faults, but also the re-integration times of the PEs, which have suffered from Type-*NF* faults. The results of the fault simulation experiments suggest that longer values for the checksum waiting timeout result in faster error correction times. However, very long checksum waiting times have a negative effect on soft error handling capabilities and the performance of the application, and should be thus avoided. Therefore, the most optimal value for the checksum waiting timeout depends greatly on the timing and reliability requirements implied by the application's specification. The values can be found and fine-tuned using high-level simulations.

The experiments proved STROBES' ability to guarantee fault tolerance of the protected application, and gave even further insight into how different parameters of the algorithm influence its fault correction performance. As a result of the experiments, additional information about the optimal parameters for the STROBES algorithm was provided.

6 Conclusions

The ever-growing complexity of electronic devices and the shrinking of transistor's feature size have made systems more susceptive to faults. These faults can be either permanent or transient. Permanent faults can be the result of manufacturing defects, but they can also occur during the product's lifetime. For example, due to aging or physical damage to the device. On the other hand, transient faults are almost always caused by environmental problems, most notably cosmic radiation, which does not only influence space-based technology. Soft errors, caused by radiation, have a noticeable effect on the operation of critical terrestrial devices, such as servers in data centers.

Repairing of permanent faults is hard. The faulty component needs to be either replaced or worked around to restore the lost functionality. The effects of transient faults, on the other hand, can be repaired more easily. Often all that is required to correct soft errors is to overwrite the faulty value in a register or memory with a known good value. However, if the effects of transient faults are not mitigated in time, they can still result in a system failure.

Modern day multiprocessor SoCs consist of many processing elements, which are connected using some form of interconnect. Recently, NoCs have emerged as the new connectivity paradigm for SoCs with high core counts, since they provide better scalability that the traditional bus-based approaches. However, the data that is transmitted over the NoC links is very sensitive to errors, since it contains information that is needed for routing the packet. An error on the link can result, in worst case, in congestion of the NoC. In this case, no further communication is possible between the nodes until the situation is resolved.

These errors on NoC links can, for example, be the result of transient faults, such as SETs, in combinational logic at the output port of the sending router. In order to guarantee reliable functioning of NoC-based devices, it is necessary to detect and recover from such errors before they can cause any harm. A mechanism for soft error mitigation for NoC links was proposed in the first contribution of this thesis:

- The proposed mechanism provides very fast soft error correction times for NoC links, normal operation can be resumed at the next raising clock edge after the transient fault has disappeared. Therefore, the increase in latency is minimal. Even in case of unrealistically high fault rates of 10 million to up to 80 million faults per second, only very small increases in latency were seen, compared to the fault free experiment.

- Additionally, the proposed mechanism uses existing redundancy in the input FIFOs of the NoC router instead of additional re-transmission buffers that are used by most similar approaches. As such, the increase in static power consumption, compared to the unprotected baseline router, is only 0.03 mW. Dynamic power consumption increased, on average, by 0.127 mW, or 1.85%.

- The proposed mechanism imposes 18% area overhead on the baseline router and increases the critical path delay by 7.8%. The results were acquired using 40 nm TSMC CMOS technology.

The mechanism provided in the first contribution can mitigate the effects of transient faults on NoC links. However, a link between the NoC routers may also fail permanently. For this reason, the routing algorithm that is used in the NoC needs to be robust so that the system could continue working with broken NoC links by finding alternative

path between the NoC nodes. Additionally, a formal way of comparing the different routing algorithms in terms of adaptivity and its reliability is needed for choosing the best turn model for a fault scenario. This leads to the second contribution:

- In the second contribution, comparison and analysis of all uniform 2D turn models was performed. To the best of the author's knowledge, an analysis of this extent has not been done before. Prior to this work, only limited number of turn models have been described in the literature. Additionally, many new metrics for turn model comparison were introduced.

- It can be formally proven that there are in total of 256 uniform 2D turn models. As the result of the analysis carried out in this work, all 50 usable turn models were identified. All other turn models, except the usable ones have either a deadlock in them or they do not provide full connectivity in the NoC and, therefore, are not usable.

- All the 50 usable turn models were further analyzed for their degree of adaptivity, the number of alternative paths they provide. The degree of adaptivity was calculated for both minimal and for non-minimal path routing. It was discovered that all 50 turn models can be grouped into three groups based on their adaptivity in case minimal path routing is used and into seven groups for non-minimal path routing. This can be done, since multiple turn models share the same adaptivity value.

- Additionally, the analysis resulted in the discovery of many new turn models that, to the author's best knowledge, have not been described before in the literature. This includes three new turn models, which have the highest adaptivity for non-minimal path routing and six turn models that are in the group that achieves the best adaptivity under minimal path routing.

- Next, average connectivity (turn model's response to broken links) and latency of all usable turn models was measured. A high correlation between average connectivity, latency and adaptivity of the turn models was shown. In case of average connectivity and latency, the exact same groups of turn models formed, as for adaptivity. Additionally, more adaptive turn models have simultaneously higher average connectivity and lower latency. Therefore, turn model's degree of adaptivity also acts as a good measure for its ability to handle link failures and its latency.

- Changing a turn model can often restore connectivity between the nodes in the network in case a link in the NoC has failed by finding alternative paths between all node pairs. Hence, an algorithm that can be used by the fault manager to identify the best turn model for a fault scenario, was presented.

However, faults can occur not only in the interconnection network, but also in the processing elements themselves. In this case, it is important that mission critical software continues to work reliably. This leads to the final contribution of the thesis on a software-based, application agnostic fault tolerance algorithm.

- Because it uses a high-level fault model, the proposed fault tolerance algorithm does not require modifications to the hardware and is application agnostic. Its performance depends only on constraints set by the system that the algorithm is

run on and its internal parameters, not on the behavior of the protected application. This is in contrast with other similar approaches that require changes to the hardware or can be used to protect only a single application or a small range of applications with similar behavior. The proposed approach is more generic.

- The algorithm works by running the same application simultaneously on three processing elements and performing distributed majority voting over the state of the application. Fault detection and correction is performed in cycles. The application is run for a certain time. Then fault detection and correction follow. In addition to soft errors in the memory used by the protected application (its state), the algorithm can also correct timing errors between the processing elements, and it can survive a silent processing element, which can happen, for example, due to a network failure.

- Soft error tolerance capabilities of the proposed algorithm were verified using Markov analysis. The analysis proved the fault tolerance capabilities of the proposed algorithm. Additionally, it can be seen from the results of the Markov analysis that the proposed algorithm can provide good tolerance against soft errors in applications with state sizes of up to 4 MiB, if the error checking frequency is high. In general, the more often error checking is performed, the higher is the MTTF. However, higher error checking frequency results in running the application for less time per each cycle. This would have a negative effect on the application's performance. The optimal point between the application's performance and fault tolerance depends on the application and on the system the algorithm is run on.

- Therefore, a custom high-level, application independent simulation tool was created. The simulation tool allows to simulate the algorithm in an application independent setting for finding optimal parameters for the algorithm.

- In this thesis, the simulation tool was used to further analyze the fault correction capabilities of the proposed algorithm, especially under delay faults that were not considered during Markov analysis. The simulation results also give further insight into selection of more optimal values for the two internal parameters of the algorithm, the application running time per cycle and checksum waiting timeout:

  - *Application running time per cycle.* As proven by Markov analysis, shorter application running times per cycle (and thus, more frequent error checking) generally improves the fault tolerance. However, the fault simulation experiments show that if the application running time per cycle is too short, it increases the average time that is required for correcting timing errors in the system. For example, in cases where the application's state size is equal to or exceeds 2 MiB, the application running time of 200 ms results, on average, faster correction of timing errors than application running time of 100 ms. However, it is important to remember that increasing the application running time also decreases MTTF of the system. Thus, there exists a trade-off between timing fault correction speed and the system's reliability. The optimal value of application running time per cycle depends largely on timing and reliability constraints and requirements imposed by the application's specification.
  - *Checksum waiting timeout.* The checksum waiting timeout parameter effects timing error correction times, but also the time it takes to re-integrate a

silent note after it comes online (e.g., after the network failure was repaired). Fault simulation results suggest that longer checksum waiting timeout values result in faster error correction times. However, like with application running times, very long checksum waiting timeouts are not recommended, since they have negative effect on applications performance and soft error correction capabilities. This is because when a processing element falls silent, the proposed fault tolerance algorithm will always wait until the checksum timeout is reached. As such, the most optimal value for the checksum waiting timeout is, like for application running time, highly dependent on the timing and reliability requirements imposed by the application's specification and it can be fine-tuned using high-level simulation.

List of Figures

1 Changes of CPU parameters over 48 years [25, 26] 11
2 Fault tolerance approaches presented in this thesis 14
3 Effects of a particle strike on a transistor 21
4 High-level structure diagram of a 3×3 NoC-based SoC utilizing the full mesh topology 24
5 Example turn models. Red arrows denote a disabled turn, black arrows represent an enabled turn 25
6 The packet format used by the Bonfire NoC 27
7 Credit-based flow control mechanism 28
8 Block schematic of the open-source Bonfire router [58] 29
9 Description of the logic-based distributed routing (LBDR) method [61] .. 30
10 Simplified version of the proposed RT mechanism. The yellow lightning symbols represent the faults that can be corrected using the proposed mechanism. 37
11 Circular buffer implementation of a FIFO 38
12 FIFO implementation for the optimized version of the proposed RT mechanism that utilizes the built-in redundancy in int circular buffer implementation of the input FIFO 39
13 Average packet latency of a 4×4 2D-mesh network under different packet (PIR) and fault injection rates (FIR) on the links with fault duration of 10% of the clock cycle. Orange line represents fault-free behavior of the RT mechanism 40
14 Comparison of dynamic and static power consumption (in mW) of the proposed and baseline 4×4 network under different packet injection rates 41
15 Routing graph for the XY routing algorithm 46
16 3×3 full-mesh NoC 46
17 Visualization of all 50 usable uniform 2D TMs, deadlock free TMs that provide full connectivity. The forbidden turns are drawn in red. 47
18 Comparison of average connectivity metric of TMs by number of available links, using minimal and non-minimal path routing 50
19 Latency results under random uniform traffic for (a) four, (b) five, (c) six-turn TMs 51
20 General structure of the fault detection, grouping and classification mechanisms for fault in control circuits 52
21 Abstract diagram of a system protected by the STROBES algorithm 59
22 An example on adding the STROBES library to an application 61
23 Diagram of the STROBES algorithm. Each PE runs an instance of this algorithm 63
24 Diagram showing the communication between the PEs during CP transfer 67
25 Differences between the real-time and non-real-time STROBES implementations. Each subfigure visualizes the behavior of a single PE under three different fault scenarios. 70
26 Correcting a Type-*SF* fault 71
27 Correcting a Type-*TF* fault which is shorter than the communication timeout 72
28 Correcting a Type-*TF* fault which is longer than the communication timeout 73

29 Effects of a second Type-*SF* fault occurring during repair time. In the example, the initial fault was in PE 0 74
30 Diagram of a markov chain representing the STROBES algorithm's fault states and the transitions between them 77
31 MTTF of STROBES algorithm for different state sizes and application running times per cycle 82
32 Architecture of the STROBES simulator 84
33 Type-SF fault correction time 87
34 Dependance of Type-TF fault correction time on application running time per cycle and state size. Please note the difference in scale for the color axis for the subfigures 88
35 Dependance of the average Type-TF fault correction time on checksum waiting timeout. Application running time per cycle is fixed to 100 ms .. 89
36 Dependance of Type-NF fault correction time on application running time per cycle and state size. Please note the difference in scale for the color axis for the subfigures 90
37 Dependance of the average Type-NF fault correction time on checksum waiting timeout. Application running time per cycle is fixed to 100 ms. 91

List of Tables

1 Comparison of approaches for protecting NoC links against the effects of soft errors 34
2 Area overhead of the proposed mechanism 40
3 Critical path delay overhead of the proposed mechanism 40
4 List of TMs that have previously been described in the literature 48
5 DoA and DoA_{Ex} for all usable 2D routing algorithms. Previously known TMs are underlined 48
6 Message types used for synchronization in the STROBES algorithm 62
7 Definitions of the fault states for a STROBES system with three PEs ... 76
8 Markov model state transitions 77
9 Constants used for reliability estimation 81
10 Type-SF fault correction times for different application running times and state sizes 87
11 Minimum, average and maximum Type-TF fault correction times for different application running times and state sizes 88
12 Minimum, average and maximum Type-NF fault correction times for different application running times and state sizes 90

References

[1] K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "AWAIT: An ultra-lightweight soft-error mitigation mechanism for Network-on-Chip links," in *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–6, July 2018.

[2] K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "Handling of SETs on NoC links by exploitation of inherent redundancy in circular input buffers," in *2018 16th Biennial Baltic Electronics Conference (BEC)*, pp. 1–4, Oct 2018.

[3] S. P. Azad, B. Niazmand, K. Janson, T. Kogge, J. Raik, G. Jervan, and T. Hollstein, "Comprehensive performance and robustness analysis of 2D turn models for Network-on-Chips," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May 2017.

[4] S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in Network-on-Chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 48–53, April 2017.

[5] K. Janson, C. J. Treudler, T. Hollstein, J. Raik, M. Jenihhin, and G. Fey, "Software-level TMR approach for on-board data processing in space applications," in *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 147–152, April 2018.

[6] J. Malburg, K. Janson, J. Raik, and F. Dannemann, "Fault-aware performance assessment approach for embedded networks," in *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 1–4, April 2019.

[7] S. R. Nassif, N. Mehta, and Y. Cao, "A resilience roadmap," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 1011–1016, March 2010.

[8] V. Huard, E. Pion, F. Cacho, D. Croain, V. Robert, R. Delater, P. Mergault, S. Engels, P. Flatresse, N. R. Amador, and L. Anghel, "A predictive bottom-up hierarchical approach to digital system reliability," in *2012 IEEE International Reliability Physics Symposium (IRPS)*, pp. 4B.1.1–4B.1.10, April 2012.

[9] J. Fang, S. Gupta, S. V. Kumar, S. K. Marella, V. Mishra, P. Zhou, and S. S. Sapatnekar, "Circuit reliability: From physics to architectures: Embedded tutorial paper," in *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 243–246, Nov 2012.

[10] S. Mahapatra, P. B. Kumar, T. R. Dalei, D. Sana, and M. A. Alam, "Mechanism of negative bias temperature instability in cmos devices: degradation, recovery and impact of nitrogen," in *IEDM Technical Digest. IEEE International Electron Devices Meeting, 2004.*, pp. 105–108, Dec 2004.

[11] F. Oboril and M. B. Tahoori, "Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level," in *IEEE/IFIP International*

Conference on Dependable Systems and Networks (DSN 2012), pp. 1–12, June 2012.

[12] S. Khan, S. Hamdioui, H. Kukner, P. Raghavan, and F. Catthoor, "Bti impact on logical gates in nano-scale cmos technology," in *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 348–353, April 2012.

[13] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[14] R. Velazco and F. J. Franco, "Single event effects on digital integrated circuits: Origins and mitigation techniques," in *2007 IEEE International Symposium on Industrial Electronics*, pp. 3322–3327, 2007.

[15] European Cooperation for Space Standardization, *ECSS-Q-HB-60-02A – Techniques for radiation effects mitigation in ASICs and FPGAs handbook*. ESA Requirements and Standards Division, 2016.

[16] B. E. Pritchard, G. M. Swift, and A. H. Johnston, "Radiation effects predicted, observed, and compared for spacecraft systems," in *IEEE Radiation Effects Data Workshop*, pp. 7–13, 2002.

[17] T. Goka, H. Matsumoto, and N. Nemoto, "See flight data from japanese satellites," *IEEE Transactions on Nuclear Science*, vol. 45, pp. 2771–2778, Dec 1998.

[18] B. D. Sierawski, R. A. Reed, K. M. Warren, A. L. Sternberg, R. A. Austin, J. M. Trippe, R. A. Weller, M. L. Alles, R. D. Schrimpf, L. W. Massengill, D. M. Fleetwood, G. W. Buxton, J. C. Brandenburg, W. B. Fisher, and R. Davis, "Cubesat: Real-time soft error measurements at low earth orbits," in *2017 IEEE International Reliability Physics Symposium (IRPS)*, pp. 3D–1.1–3D–1.6, April 2017.

[19] L. A. Aranda, P. Reviriego, and J. A. Maestro, "Towards a fault-tolerant star tracker for small satellite applications," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 1–1, 2020.

[20] D. N. Baker, "How to cope with space weather," *Science*, vol. 297, no. 5586, pp. 1486–1487, 2002.

[21] D. N. Baker, S. G. Kanekal, X. Li, S. P. Monk, J. Goldstein, and J. L. Burch, "An extreme distortion of the van allen belt arising from the 'hallowe'en' solar storm in 2003," *Nature*, vol. 432, pp. 878–881, Dec 2004.

[22] R. Edwards, C. Dyer, and E. Normand, "Technical standard for atmospheric radiation single event effects, (see) on avionics electronics," in *2004 IEEE Radiation Effects Data Workshop (IEEE Cat. No.04TH8774)*, pp. 1–5, 2004.

[23] K. W. Harris, "Asymmetries in soft-error rates in a large cluster system," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 336–342, 2005.

[24] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 397–404, Sep. 2005.

[25] K. Rupp, "42 Years of Microprocessor Trend Data." https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, Jun 2018. Accessed: 2020-10-26.

[26] K. Rupp, "Microprocessor Trend Data Github Page." https://github.com/karlrupp/microprocessor-trend-data, Jul 2020. Accessed: 2020-10-26.

[27] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam: Morgan Kaufmann, 5 ed., 2012.

[28] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, pp. 684–689, 2001.

[29] F. Thoma, M. Kuhnle, P. Bonnot, E. M. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schuler, K. D. Muller-Glaser, and J. Becker, "Morpheus: Heterogeneous reconfigurable computing," in *2007 International Conference on Field Programmable Logic and Applications*, pp. 409–414, 2007.

[30] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pp. 88–598, 2008.

[31] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, S. Vora, and E. Wang, "Skylake-sp: A 14nm 28-core xeon® processor," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 34–36, 2018.

[32] T. Burd, N. Beck, S. White, M. Paraschou, N. Kalyanasundharam, G. Donley, A. Smith, L. Hewitt, and S. Naffziger, ""zeppelin": An soc for multichip architectures," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 133–143, 2019.

[33] P. Conway and B. Hughes, "The amd opteron northbridge architecture," *IEEE Micro*, vol. 27, no. 2, pp. 10–21, 2007.

[34] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. Shoaib Bin Altaf, N. Enright Jerger, and G. H. Loh, "Modular routing design for chiplet-based systems," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 726–738, 2018.

[35] "ITRS International Technology Roadmap for Semiconductors. Process integration, devices, and structures (PIDS)," 2009.

[36] J. W. McPherson, "Reliability trends with advanced cmos scaling and the implications for design," in *2007 IEEE Custom Integrated Circuits Conference*, pp. 405–412, Sep. 2007.

[37] M. Alam, K. Roy, and C. Augustine, "Reliability- and process-variation aware design of integrated circuits — a broader perspective," pp. 4A.1.1 – 4A.1.11, 05 2011.

[38] R. Harboe-Sorensen, E. Daly, F. Teston, H. Schweitzer, R. Nartallo, P. Perol, F. Vandenbussche, H. Dzitko, and J. Cretolle, "Observation and analysis of single event effects on-board the soho satellite," in *RADECS 2001. 2001 6th European Conference on Radiation and Its Effects on Components and Systems (Cat. No.01TH8605)*, pp. 37–43, Sep. 2001.

[39] N. V. Kuznetsov, "The rate of single event upsets in electronic circuits onboard spacecraft," *Cosmic Research*, vol. 43, no. 6, pp. 423–431, 2005.

[40] K. Weide-Zaage, P. Eichin, Chen Chen, Yupeng Zhao, and Lifan Zhao, "Cots - radiation effects approaches and considerations," in *2017 Pan Pacific Microelectronics Symposium (Pan Pacific)*, pp. 1–8, Feb 2017.

[41] S. Esposito, C. Albanese, M. Alderighi, F. Casini, L. Giganti, M. L. Esposti, C. Monteleone, and M. Violante, "Cots-based high-performance computing for space applications," *IEEE Transactions on Nuclear Science*, vol. 62, pp. 2687–2694, Dec 2015.

[42] M. Pignol, "Cots-based applications in space avionics," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 1213–1219, March 2010.

[43] A. J. P. Theuwissen, "Influence of terrestrial cosmic rays on the reliability of ccd image sensors—part 1: Experiments at room temperature," *IEEE Transactions on Electron Devices*, vol. 54, pp. 3260–3266, Dec 2007.

[44] T. J. O'Gorman, "The effect of cosmic rays on the soft error rate of a dram at ground level," *IEEE Transactions on Electron Devices*, vol. 41, pp. 553–557, April 1994.

[45] K. Höflinger, S. Müller, T. Peng, M. Ulmer, D. Lüdtke, and A. Gerndt, "Dynamic fault tree analysis for a distributed onboard computer," in *2019 IEEE Aerospace Conference*, pp. 1–13, 2019.

[46] E. Dubrova, *Fault-Tolerant Design*. 978-1-4614-2113-9: Springer-Verlag New York, first ed., 2013.

[47] J. Wilkinson and S. Hareland, "A cautionary tale of soft errors induced by sram packaging materials," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 428–433, 2005.

[48] T. C. May and M. H. Woods, "A new physical mechanism for soft errors in dynamic memories," in *16th International Reliability Physics Symposium*, pp. 33–40, 1978.

[49] F. Wang and V. D. Agrawal, "Single event upset: An embedded tutorial," in *21st International Conference on VLSI Design (VLSID 2008)*, pp. 429–434, 2008.

[50] R. C. Baumann and E. B. Smith, "Neutron-induced boron fission as a major source of soft errors in deep submicron sram devices," in *2000 IEEE International Reliability Physics Symposium Proceedings. 38th Annual (Cat. No.00CH37059)*, pp. 152–157, 2000.

[51] V. Savulimedu Veeravalli, T. Polzer, U. Schmid, A. Steininger, M. Hofbauer, K. Schweiger, H. Dietrich, K. Schneider-Hornstein, H. Zimmermann, K.-O. Voss, B. Merk, and M. Hajek, "An infrastructure for accurate characterization of single-event transients in digital circuits," *Microprocessors and Microsystems*, vol. 37, no. 8, Part A, pp. 772 – 791, 2013. Special Issue DSD 2012 on Reliability and Dependability in MPSoC Technologies.

[52] M. Ottavi, D. Gizopoulos, and S. Pontarelli, *Dependable multicore architectures at nanoscale*. Springer, 2018.

[53] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, pp. 43–98, 1956.

[54] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1, pp. 293–307 vol.1, Feb 1996.

[55] P. Saltarelli, B. Niazmand, J. Raik, V. Govind, T. Hollstein, G. Jervan, and R. Hariharan, "A framework for combining concurrent checking and on-line embedded test for low-latency fault detection in noc routers," in *Proceedings of the 9th International Symposium on Networks-on-Chip*, NOCS '15, (New York, NY, USA), pp. 6:1–6:8, ACM, 2015.

[56] D. L. How and S. Atsatt, "Sectors: Divide conquer and softwarization in the design and validation of the stratix® 10 fpga," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 119–126, 2016.

[57] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, "2.2 amd chiplet architecture for high-performance server and desktop products," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 44–45, 2020.

[58] "Project Bonfire Network-on-Chip Repository." `https://github.com/Project-Bonfire`, 2015. Accessed: 2020.03.14.

[59] S. Kumar, A. Jantsch, J. P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pp. 105–112, 2002.

[60] L. Benini and G. D. Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, pp. 70–78, Jan 2002.

[61] J. Flich and J. Duato, "Logic-based distributed routing for NoCs," *IEEE Computer Architecture Letters*, vol. 7, pp. 13–16, Jan 2008.

[62] J. Duato, S. Yalamanchili, and N. Lionel, *Interconnection Networks: An Engineering Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[63] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pp. 278–287, 1992.

[64] G. Schley, N. Batzolis, and M. Radetzki, "Fault localizing End-to-End flow control protocol for Networks-on-Chip," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 454–461, Feb 2013.

[65] E. A. Rambo, C. Seitz, S. Saidi, and R. Ernst, "Designing Networks-on-Chip for high assurance real-time systems," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 185–194, Jan 2017.

[66] E. Wachter, V. Fochi, F. Barreto, A. Amory, and F. Moraes, "A hierarchical and distributed fault tolerant proposal for NoC-based MPSoCs," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2017.

[67] M. Ali, M. Welzl, S. Hessler, S. Hellebrand, and S. And, "An efficient fault tolerant mechanism to deal with permanent and transient failures in a network on chip," *International Journal of High Performance Systems Architecture*, vol. 1, 01 2007.

[68] S. Shamshiri, A. Ghofrani, and K. T. Cheng, "End-to-end error correction and online diagnosis for on-chip networks," in *2011 IEEE International Test Conference*, pp. 1–10, Sept 2011.

[69] S. Ogg, B. Al-Hashimi, and A. Yakovlev, "Asynchronous transient resilient links for NoC," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, (New York, NY, USA), pp. 209–214, ACM, 2008.

[70] A. P. Frantz, F. L. Kastensmidt, L. Carro, and E. Cota, "Dependable Network-on-Chip router able to simultaneously tolerate soft errors and crosstalk," in *2006 IEEE International Test Conference*, pp. 1–9, 2006.

[71] S. R. Naqvi, V. S. Veeravalli, and A. Steininger, "Protecting an asynchronous NoC against transient channel faults," in *2012 15th Euromicro Conference on Digital System Design*, pp. 264–271, Sept 2012.

[72] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das, "Exploring fault-tolerant Network-on-Chip architectures," in *International Conference on Dependable Systems and Networks (DSN'06)*, pp. 93–104, June 2006.

[73] D. DiTomaso, T. Boraten, A. Kodi, and A. Louri, "Dynamic error mitigation in NoCs using intelligent prediction techniques," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[74] A. Dutta and N. A. Touba, "Reliable network-on-chip using a low cost unequal error protection code," in *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, pp. 3–11, Sept 2007.

[75] X. Chen, Z. Lu, Y. Lei, Y. Wang, and S. Chen, "Multi-bit transient fault control for NoC links using 2D fault coding method," in *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pp. 1–8, Aug 2016.

[76] A. P. Frantz, M. Cassel, F. L. Kastensmidt, É. Cota, and L. Carro, "Crosstalk- and SEU-aware Networks on Chips," *IEEE Design Test of Computers*, vol. 24, pp. 340–350, July 2007.

[77] S. Pasricha and Y. Zou, "A low overhead fault tolerant routing scheme for 3d networks-on-chip," in *2011 12th International Symposium on Quality Electronic Design*, pp. 1–8, March 2011.

[78] M. Ebrahimi, M. Daneshtalab, J. Plosila, and H. Tenhunen, "Mafa: Adaptive fault-tolerant routing algorithm for networks-on-chip," in *2012 15th Euromicro Conference on Digital System Design*, pp. 201–207, Sep. 2012.

[79] R. Salamat, M. Ebrahimi, and N. Bagherzadeh, "An adaptive, low restrictive and fault resilient routing algorithm for 3d network-on-chip," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 392–395, March 2015.

[80] F. Dubois, A. Sheibanyrad, F. Pétrot, and M. Bahmani, "Elevator-first: A deadlock-free distributed routing algorithm for vertically partially connected 3d-nocs," *IEEE Transactions on Computers*, vol. 62, pp. 609–615, March 2013.

[81] P. Lotfi-Kamran, A. M. Rahmani, M. Daneshtalab, A. Afzali-Kusha, and Z. Navabi, "Edxy – a low cost congestion-aware routing algorithm for network-on-chips," *Journal of Systems Architecture - Embedded Systems Design*, vol. 56, pp. 256–264, 07 2010.

[82] J. Khichar and S. Choudhary, "Fault aware adaptive routing algorithm for mesh based NoCs," in *2017 International Conference on Inventive Computing and Informatics (ICICI)*, pp. 584–589, Nov 2017.

[83] J. Khichar, S. Choudhary, and R. Mahar, "Fault tolerant dynamic xy-yx routing algorithm for network on-chip architecture," in *2017 International Conference on Intelligent Computing and Control (I2C2)*, pp. 1–6, June 2017.

[84] K. Tatas, S. Sawa, and C. Kyriacou, "Low-cost fault-tolerant routing for regular topology nocs," in *2014 21st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 566–569, Dec 2014.

[85] A. M. Shafiee, M. Montazeri, and M. Nikdast, "An innovational intermittent algorithm in networks-on-chip (noc)," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 2, no. 9, pp. 2907 – 2909, 2008.

[86] A. DeOrio, D. Fick, V. Bertacco, D. Sylvester, D. Blaauw, J. Hu, and G. Chen, "A reliable routing architecture and algorithm for nocs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, pp. 726–739, May 2012.

[87] T. Putkaradze, S. P. Azad, B. Niazmand, J. Raik, and G. Jervan, "Fault-resilient noc router with transparent resource allocation," in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, July 2017.

[88] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[89] Y. Miura, K. Shimozono, S. Watanabe, and K. Matoyama, "An Adaptive Routing of the 2-D Torus Network Based on Turn Model," in *Computing and Networking (CANDAR), 2013 First International Symposium on*, pp. 587–591, Dec 2013.

[90] S. Mubeen and S. Kumar, "Designing efficient source routing for mesh topology network on chip platforms," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pp. 181–188, Sept 2010.

[91] A. Patooghy and H. Sarbazi-Azad, "Performance comparison of partially adaptive routing algorithms," in *20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, vol. 2, pp. 5 pp.–, April 2006.

[92] J. Duato, "A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 1320–1331, Dec. 1993.

[93] S. P. Azad, B. Niazmand, P. Ellervee, J. Raik, G. Jervan, and T. Hollstein, "Socdep2: A framework for dependable task deployment on many-core systems under mixed-criticality constraints," in *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–6, June 2016.

[94] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 162–163, July 2015.

[95] G. Ascia, V. Catania, M. Palesi, and D. Patti, "Neighbors-on-path: A new selection strategy for on-chip networks," in *2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pp. 79–84, Oct 2006.

[96] 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 *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 60–71, IEEE Computer Society, 2012.

[97] J. Silveira, C. Marcon, P. Cortez, G. Barroso, J. a. M. Ferreira, and R. Mota, "Scenario preprocessing approach for the reconfiguration of fault-tolerant noc-based mpsocs," *Microprocess. Microsyst.*, vol. 40, pp. 137–153, Feb. 2016.

[98] J. Silveira, M. Bodin, J. M. Ferreira, A. C. Pinheiro, T. Webber, and C. Marcon, "A fault prediction module for a fault tolerant noc operation," in *Sixteenth International Symposium on Quality Electronic Design*, pp. 284–288, March 2015.

[99] A. Strano, D. Bertozzi, F. Triviño, J. L. Sánchez, F. J. Alfaro, and J. Flich, "Osr-lite: Fast and deadlock-free noc reconfiguration framework," in *2012 International Conference on Embedded Computer Systems (SAMOS)*, pp. 86–95, July 2012.

[100] S. Habinc, A. Sakthivel, J. Ekergarn, A. Bjorkengren, R. Pender, S. Landstrom, F. Cordero, J. Mendes, T.-M. Ho, and K. Stohlmann, "Mascot on-board computer based on GR712RC," in *Data Systems in Aerospace (DASIA)*, 2013.

[101] A. Patooghy, S. G. Miremadi, A. Javadtalab, M. Fazeli, and N. Farazmand, "A solution to single point of failure using voter replication and disagreement detection," in *2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pp. 171–176, Sep. 2006.

[102] C. L. Chen, Y. H. Chen, and T. Hwang, "Communication driven remapping of processing element (PE) in fault-tolerant NoC-based MPSoCs," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 666–671, 2017.

[103] V. Fochi, L. L. Caimi, M. Ruaro, E. Wächter, and F. G. Moraes, "System management recovery protocol for MPSoCs," in *IEEE International System-on-Chip Conference (SOCC)*, pp. 367–374, 2017.

[104] X. Chen, W. Yang, M. Zhao, and J. Wang, "Hls-based sensitivity-inductive soft error mitigation for satellite communication systems," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 143–148, July 2016.

[105] S. Tanoue, T. Ishida, Y. Ichinomiya, M. Amagasaki, M. Kuga, and T. Sueyoshi, "A novel states recovery technique for the tmr softcore processor," in *2009 International Conference on Field Programmable Logic and Applications*, pp. 543–546, Aug 2009.

[106] T. Sato, "Exploiting instruction redundancy for transient fault tolerance [microprocessor applications]," in *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 547–554, 2003.

[107] S. Kumar and A. Aggarwal, "Self-checking instructions — reducing instruction redundancy for concurrent error detection," in *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 64–73, 2006.

[108] R. Gong, K. Dai, and Z. Wang, "Transient fault tolerance on chip multiprocessor based on dual and triple core redundancy," in *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 273–280, 2008.

[109] L. Lamport, "Paxos made simple," in *ACM SIGACT News*, vol. 32, pp. 18–25, 2001.

[110] D. Ongaro and J. Ousterhout, "Search of an understandable consensus algorithm," in *Proc. 2014 USENIX Annual Technical Conference (ATC '14)*, pp. 305–319, 2014.

[111] J. Quiané-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "Rafting mapreduce: Fast recovery on the raft," in *2011 IEEE 27th International Conference on Data Engineering*, pp. 589–600, 2011.

[112] M. Fattah, M. Palesi, P. Liljeberg, J. Plosila, and H. Tenhunen, "SHiFA: System-level hierarchy in run-time fault-aware management of many-core systems," pp. 1–6, 2014.

[113] C. Bolchini, A. Miele, and D. Sciuto, "An adaptive approach for online fault management in many-core architectures," pp. 1429–1432, 2012.

[114] J. W. Alexander, B. J. Clement, K. P. Gostelow, and J. Y. Lai, "Fault mitigation schemes for future spaceflight multicore processors," pp. 358–367, 2012.

[115] H. Fu, M. Cai, L. Fang, P. Liu, and J. Dong, "Research on rtos-integrated tmr for fault tolerant systems," in *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, vol. 3, pp. 750–755, July 2007.

[116] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, pp. 243–254, 2005.

[117] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," *SIGPLAN Not.*, vol. 45, p. 385–396, Mar. 2010.

[118] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-directed instruction duplication for soft error detection," in *Design, Automation and Test in Europe*, pp. 1056–1057 Vol. 2, 2005.

[119] L. Tan, Q. Tan, and J. Xu, "Automatic instruction-level recovery by duplicated instructions and checkpointing," in *2012 5th International Conference on BioMedical Engineering and Informatics*, pp. 1304–1307, 2012.

[120] C. K. Chang, G. Li, and M. Erez, "Evaluating compiler ir-level selective instruction duplication with realistic hardware errors," in *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pp. 41–49, 2019.

[121] J. F. Hermant and G. L. Lann, "Fast asynchronous uniform consensus in real-time distributed systems," *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 931–944, 2002.

[122] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[123] C. Honvault, M. Le Roy, P. Gula, J. C. Fabre, G. Le Lann, and E. Bornschlegl, "Novel generic middleware building blocks for dependable modular avionics systems," in *European Dependable Computing Conference*, pp. 140–153, Springer, 2005.

[124] "MATLAB website."
`https://se.mathworks.com/products/matlab.html`, 2016.
Accessed: 2020.06.10.

[125] M. Amiri and V. Přenosil, "General solutions for mttf and steady-state availability of nmr systems," in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–4, 2014.

[126] European Cooperation for Space Standardization, *ECSS-E-ST-50-12C Rev.1 – Techniques for radiation effects mitigation in ASICs and FPGAs handbook*. ESA Requirements and Standards Division, 2016.

Acknowledgements

I would like to express my gratitude to my PhD advisor, Professor Thomas Hollstein, and to my co-advisor, Professor Jaan Raik for their support, guidance, and time throughout my PhD studies.

I would also like to thank our department head, Margus Kruus, for his support with administrative issues.

Special thanks go to my colleges and co-authors in the Department of Computer Engineering, especially to Siavoosh Payandeh Azad, Behrad Niazmand, René Pihlak and Professor Maksim Jenihhin for the support, help, and the discussions we had during my PhD studies.

Additionally, I would thank my co-authors from German Aerospace Center (DLR), especially professor Görschwin Fey, Carl Johann Treudler and Jan Malburg for their help and support and for letting me stay in DLR in Bremen and work with them.

I would also like to acknowledge the organizations, which have supported me throughout my PhD studies: IUT 19-1 of the Estonian Ministry of Education and Research, the Estonian IT Academy program, the EU H2020 RIA IMMORTAL and the EU Twinning Action TUTORIAL project.

Last, but not least, I would like to thank my family for all the support I received from them over the years.

Abstract
Techniques for Robust Routing, Communication and Computation in Multiprocessor Systems

Over the previous decades, computing technology has evolved at an incredible pace. This is largely a result of the ongoing decrease in the size of the technology node. This process has allowed to build increasingly more complex multiprocessor systems-on-chip, while keeping the cost and power consumption constant. Unfortunately, devices that are built using smaller technologies also become more susceptive to faults.

Faults in electronics can be roughly categorized into permanent and transient (temporary) faults. Permanent faults include, for example, manufacturing defects and faults that are caused by aging, the degradation of the transistors over lifetime due to different physical phenomena. Permanent faults are difficult to correct, since in order to restore the lost functionality, the faulty component needs to be either replaced or worked around of. Transient faults, on the other hand, do not result in permanent damage to the device. One of the largest causes of transient faults in electronics is cosmic radiation. If an energetic, charged particle hits a memory element, it can cause a soft error (a random bit flip). Alternatively, soft errors can also occur when a transient fault in the combinational logic is read into a memory element. Soft errors can be easily corrected by overwriting the erroneous value with a correct one. However, if the soft error is not corrected in time, before the erroneous value is used, it can still lead to a failure.

Faults that are caused by cosmic radiation are an especially large problem for satellites, since they orbit outside of the protection of the Earth's magnetic field. However, radiation is not an issue for only satellites, as it also has a large impact on the operation of airplanes and many terrestrial devices, such as servers in data centers. Therefore, to guarantee dependable operation of mission critical computer systems even under faults, fault tolerance has become an essential component of these systems.

On an abstract level, a multiprocessor system consists of processing elements, which communicate to each other using the interconnect, and the software that runs on the processing elements. Faults that occur in the interconnects usually manifest as communication errors. In this case, the data either arrives corrupt or never arrives at all at its destination. On the other hand, faults in the processing elements generally manifest themselves at the software layer by either causing a direct change in the application's state or resulting in a computational error. Therefore, fault tolerance should be implemented at multiple parts of the system. This dissertation presents three fault tolerance approaches for handling different types of errors at different parts of the system.

First, a design for a relaxed transmission mechanism is presented to handle soft errors on the links of network-on-chip based interconnects. These errors could otherwise lead to congestions in the interconnection network or cause a packet re-transmission, which causes additional latency. Soft errors at the network-on-chip links are caused by radiation-induced faults in the combinational part at the output of the sending router. Compared to similar approaches, the proposed mechanism does not need additional re-transmission buffers and provides a very fast error recovery time, since fault-free operation can be resumed at the next raising clock edge after the transient fault, which caused the soft error, has disappeared.

Next, for a first time, an exhaustive analysis of all uniform 2D turn model-based network-on-chip routing algorithms is performed. Previously, only a limited number

of popular routing algorithms have been analyzed. The analysis in this thesis resulted in discovery of many new turn model-based routing algorithms, which have not been described before in the literature. Moreover, in this thesis, additional metrics are provided for qualitative comparison of turn model-based routing algorithms. These metrics allow to use a novel an approach, proposed in this thesis, to find an optimal routing algorithm reconfiguration to maximize the use of healthy NoC routing resources, based on any specific pattern of permanent faults.

Finally, a new software-based fault tolerance algorithm has been proposed that enables fault tolerant execution of code on non-fault tolerant processing elements. The proposed approach is based on a high-level fault model that allows implementing fault tolerance completely in software in a manner that the fault tolerance algorithm's performance does not depend on the behavior of the protected application. Therefore, the proposed algorithm can be used for protecting a wide range of different applications by interfacing the applications with a library that contains the fault tolerance algorithm using a simple interface. In contrast, most similar fault tolerance approaches are limited to a very small range of applications that have similar behavior or require either modified or inherently fault tolerant hardware.

Additionally, the usage of the high-level and application agnostic fault model enables simulating the proposed fault tolerance algorithm independently of the protected application. This allows further fine-tuning the algorithm's parameters in order to optimize its performance and fault tolerance. The fault correction ability of the proposed approach is proven by Markov analysis and through fault simulations.

While the proposed approaches work independently, they can be used together for increased fault tolerance.

Lühikokkuvõte
Robustse marsruutimise, side ja arvutuse tehnikad mitmeprotsessorilistes süsteemides

Viimaste kümnenditega on arvutitehnika väga kiiresti arenenud. Selle üks peamisi põhjuseid on võimekus toota üha väiksemate mõõtmetega transistore. Nii saab ehitada üha integreeritumaid ja keerukamaid kiipsüsteeme samas, kui seadmete voolutarve ja hind püsivad muutumatutena. Paraku aga muutuvad transistoride suuruse kahanedes seadmed vastuvõtlikumaks ka riketele.

Üldjoontes saab rikked liigitada püsivateks ja ajutisteks. Püsivad rikked on näiteks tootmisdefektid ja sellised kahjustused, mis on tingitud transistori vananemisest ehk selle aeglasest lagunemisest mitme füüsikalise nähtuse tõttu. Püsivaid rikkeid on keeruline parandada: rikkis komponendi peab kas asendama või süsteemi nii ümber seadistama, et olemasolevaid ressursse kasutades saaks rikke tõttu kaotatud funktsionaalsuse vähemalt osaliselt taastada.

Ajutised rikked seevastu ei põhjusta seadmeile püsivaid kahjustusi. Üks suurim ajutiste rikete põhjus elektroonikas on kosmiline kiirgus. Kui energeetiline laetud osake tabab mäluelementi, võib see põhjustada selles mäluelemendis pehme vea, muutes biti väärtuse vastupidiseks. Teiseks võivad pehmed vead ilmneda siis, kui kiirguse poolt tekitatud ajutine rike kombinatoorses loogikas loetakse mäluelementi sisse. Pehmeid vigu saab hõlpsasti parandada, kirjutades mäluelemendis vale väärtuse õigega üle. Küll aga võib ka pehme viga põhjustada ohtliku tõrke kui ajutisest rikkest tingitud pehmet viga ei parandata enne vigases mälupesas oleva väärtuse kasutamist.

Kosmilisest kiirgusest põhjustatud rikked on eriti suur probleem satelliitide jaoks, kuna nende orbiidid paiknevad väljaspool Maa magnetvälja kaitset. Samas ei ole kiirgus ainult satelliitide probleem, sellel on suur mõju ka lennukite ja ka paljude maapealsete seadmete toimimisele. Üks selliste süsteemide näiteks on serverid andmekeskustes. Seega, et tagada missioonikriitiliste arvutisüsteemide usaldusväärne töö isegi rikete korral, on rikketaluvus muutunud nende süsteemide oluliseks komponendiks.

Abstraktsemal tasandil saab hajusad sardsüsteemid jagada arvutuselementideks, neid ühendavaks võrguks ja tarkvaraks, mis arvutuselementides töötab. Rikked ühendusvõrgus avalduvad peamiselt ühendus- või suhtlusvigadena arvutuselementide vahel. Sel juhul jõuavad andmed sihtkohta kas vigaselt või ei jõua üldse. Samas ilmnevad arvutuselementide rikked sageli alles tarkvaratasandil, põhjustades tarkvara oleku otsese muutuse või valearvutuse. Seetõttu tuleks rikketaluvusmehhanisme rakendada süsteemi mitmes osas. Selles doktoritöös esitatakse kolme rikketaluvuse meetodit erinevat tüüpi rikete käsitlemiseks süsteemi eri osades.

Esiteks tutvustatakse rahuliku edastamise (*relaxed transmission*) lahendust, mille eesmärk on tulla toime pehmete vigadega kiipvõrgu marsruuterite vahelistes ühendustes. Sellised vead võivad vastasel juhul põhjustada kiipvõrgus ummikuid või paketi taasedastamist, mis suurendab latentsust. Suur osa pehmeid vigu kiipvõrgu ühendustel on põhjustatud radiatsiooni tulemusel tekkinud riketest kobinatoorses loogikas, mis asub saatva marsruuteri väljundis. Võrreldes sarnaste lähenemisviisidega ei vaja pakutav mehhanism täiendavaid ülekandepuhvreid ja tagab riketest väga kiire taastumise, kuna rikkevaba talitlust saab jätkata selle tõusva taktisignaali serva ajal, järgneb pehme vea tekitanud ajutise rikke kadumisele.

Järgmisena esitatakse esimest korda süvaanalüüs kõigist kahemõõtmelisel pöördemudelil põhinevatest kiipsüsteemi marsruutimisalgoritmidest. Varem on analüüsitud vaid piiratud arvu populaarseid marsruutimisalgoritme. Selles väitekirjas esitatud analüüsi

tulemusel avastati palju uusi pöördemudelipõhiseid marsruutimisalgoritme, mida pole kirjanduses varasemalt mainitud. Lisaks esitatakse väitekirjas lisamõõdikud, et pöördemudelil põhinevaid marsruutimisalgoritme kvalitatiivselt võrrelda. Need mõõdikud võimaldavad kasutada väitekirjas pakutud uudset lähenemisviisi, mis aitab leida iga püsivate rikete kombinatsiooni kohta optimaalseima marsruutimisalgoritmi, mis lubaks kiipvõrkudes töökorras olevaid marsruutimisressursse maksimaalselt ära kasutada.

Lõpuks pakutakse välja uus, tarkvarapõhine veataluvusalgoritm, mis kindlustab arvutiprogrammide veakindluse. See lahendus aitab tagada missioonikriitilise tarkvara rikkevaba töö arvutuselementides, millele pole riistvaralisi rikketaluvusmehhanisme sisseehitatud. Seetõttu saab pakutud algoritmi kasutada paljude eri rakenduste kaitsmiseks, ühendades rakendused lihtsa liidese abil teegiga, mis sisaldab rikketaluvuse algoritmi. Enamik teisi sarnaseid rikketaluvusalgoritme on kasutatavad üksnes väga väikese hulga rakendustega, mis käituvad sarnaselt või vajavad kas muudetud või juba olemuslikult rikkekindlat riistvara.

Lisaks võimaldab kõrgetasemelise ja kaitstavast tarkvararakendusest sõltumatu rikkemudeli kasutus simuleerida esitletavat rikketaluvusalgoritmi, sõltumatult sellega kaitstavast rakendusest. Nii on võimalik algoritmi parameetreid täpsemalt häälestada, et optimeerida selle jõudlust ja rikketaluvust. Pakutud rikketaluvusalgoritmi võime rikkeid tuvastada ja parandada on tõestatud Markovi analüüsi ja rikkesimulatsioonide abil.

Kuigi esitletud lahendused töötavad sõltumatult, on neid võimalik kasutada parema rikketaluvuse saavutamiseks ka koos.

Appendix 1

I

K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "AWAIT: An ultra-lightweight soft-error mitigation mechanism for Network-on-Chip links," in *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–6, July 2018

AWAIT: An Ultra-Lightweight Soft-Error Mitigation Mechanism for Network-on-Chip Links

Karl Janson, René Pihlak, Siavoosh Payandeh Azad, Behrad Niazmand, Gert Jervan, Jaan Raik
Department of Computer Systems, Tallinn University of Technology,
email: {karl.janson, rene.pihlak, siavoosh.azad, behrad.niazmand, gert.jervan, jaan.raik}@ttu.ee

***Abstract*—Networks-on-Chip have become a widely accepted communication paradigm for many-core Systems-on-Chip. However, with the ever-shrinking transistor size, the network's sensitivity to transient faults on the physical links cannot be ignored since even a single transient fault can lead to a network-wide congestion and a system failure. This paper proposes the AWAIT mechanism, an ultra-lightweight transient fault mitigation mechanism for Network-on-Chip links. The proposed mechanism covers all single event transients. The experimental results show that the AWAIT mechanism prevents network-wide failure even in harsh environments (up to 80 million random faults on links per second). The AWAIT mechanism is also scalable and imposes only 5.1% area overhead with very negligible critical path delay overhead.**

***Index Terms*—Network-on-Chip, Fault tolerance, Transient fault mitigation**

I. Introduction

The ever-decreasing transistor feature size has enabled the integration of large number of cores on a single die. However, the traditional bus-based interconnection infrastructure is not scalable enough and becomes the bottleneck in inter-core communication. Network-on-Chip (NoC) has emerged as a solution to this problem. However, the decreasing feature size also makes the system much more susceptible to both environmental faults (*e.g.*, faults caused by radiation) and also introduces new problems, such as Negative-Bias Temperature Instability (NBTI) and Hot Carrier Injection (HCI) [1]. The aforementioned problems add the need for integrated reliability mechanisms. In this paper we concentrate on improving the reliability of the inter-router links.

This paper proposes AWAIT, a scalable mechanism which is able to handle an extensive amount of transient faults (see Section VI) by relying on a single parity bit for fault detection, together with only a few additional gates per link for transient fault mitigation. This approach enables the detection and correction of all single bit link faults while requiring eight times less data overhead per flit and 10 times less area for 32-bit links than Hamming code, which only provides single bit correction [2].

The AWAIT mechanism pauses system operation in the affected part of the system until the faults disappear and does not require retransmission of data. Applying this method would result in only a slight increase in latency in the presence of faults and in case of transient faults will always guarantee correct arrival of the data.

978-1-5386-3344-1/17/$31.00

This paper focuses only on Single Event Transient (SET) faults in NoC links (due to the use of a parity checker for fault detection). Moreover, it is important to note that the AWAIT mechanism is not limited to the usage of the parity checker but can be also combined with other fault detection (or correction) mechanisms, if needed. The behavior in the presence of permanent faults is out of the scope of this paper.

The rest of this paper is organized as follows. Section II provides a literature review. Sections III and IV provide details of the baseline router and an analysis of the NoC's behavior in different operation environments. Section V will provide details on the proposed AWAIT mechanism and Section VI will provide experimental results which illustrate the efficiency of the proposed mechanism. Finally, section VII will conclude the paper.

II. Literature Review

Many works have investigated the problem of transient fault mitigation for physical links in NoCs. However, most of the mechanisms proposed in the literature are very costly in terms of latency, area or data overhead. In general, all fault tolerance mechanisms used for NoC links can be categorized as End-to-End (E2E) or Hop-by-Hop (HBH) mechanisms, based on the fault detection and mitigation granularity. In HBH mechanisms, the packet/flit is examined every time the data is transferred from one router to another. However, in E2E mechanisms, a fault is detected only once the packet is ejected from the network. HBH mechanisms have usually lower latency than the E2E based mechanisms, since in E2E mechanisms the entire packet needs to be retransmitted in case a fault, which cannot be corrected in the endpoint, is detected.

In order to mitigate transient faults in physical links in a NoC, three main approaches exist in the literature:

A. Data Retransmission

1) Packet Retransmission (P-RET): Works such as [2], [6], [3], [4] and [5] have proposed methods based on End-to-End (E2E) packet retransmission. This approach suffers from high latency caused by the retransmission; once a faulty packet is detected, often by an Error Detecting Code (EDC), a NACK packet should be sent to the sender of the faulty packet for requesting a retransmission. During this process, the receiving node might need to also discard the non-faulty packets in order to preserve the packet order. It is important to note that such E2E packet retransmission mechanisms require

TABLE I: Comparison of related works

Approach	Used Method	HBH/E2E	No Extra Buffers	Targeted Fault Model	Correction Latency
[2]	P-RET	E2E	✓	SETs + 50% of METs	$\gg$ 1 clk
[3]	P-RET	E2E	✓	SETs	$\gg$ 1 clk
[4]	P-RET	E2E	✓	SETs + METs	$\gg$ 1 clk
[5]	P-RET	E2E	✗	SETs + METs	$\gg$ 1 clk
[6]	P-RET	E2E	✗	SETs	$\gg$ 1 clk
[7]	ECC	E2E	✓	SETs	N.A.
[8]	ECC	HBH	✗	SETs	N.A.
[9]	ECC	HBH	✗	SETs	N.A.
[10]	F-RET	HBH	✓	SETs	N.A.
[11]	F-RET	HBH	✗	SETs + METs	3 clks
[12]	RT + ECC + EDC	HBH	✗	SETs + METs (Prediction)	2 clks
[13]	P-RET	HBH	✗	SETs + Some DET	N.A.
[14]	ECC	HBH	✗	SETs + METs	N.A.
[15]	ECC	HBH	✗	SETs	$\gg$ 1 clk
Proposed	**RT**	**HBH**	✓	**SETs + 50% of METs**	**0-1 clk**

distributed packet dropping mechanisms in order to prevent a network-wide failure.

On the other hand, [13] has proposed HBH packet retransmission using store-and-forward switching. However, store-and-forward requires large input buffers since each router has to buffer the entire packet before forwarding it, which is very inefficient use of network buffers and chip area.

2) Flit retransmission (F-RET): In contrast to packet retransmission, works such as [11] and [10] have proposed HBH flit retransmission in case of receiving a faulty flit. These works provide much lower latency compared to E2E approaches but suffer from the need for additional retransmission buffers in the router.

B. Error Correction Code (ECC)

Another class of fault tolerance mechanisms for NoC links is Error Correction Codes (ECCs). Works such as [8], [9] and [15] have provided solutions for correcting single transient upsets, while [14] has used more sophisticated error correction techniques to cover multiple event transients. Similar to retransmission mechanisms, ECCs are also implemented either as HBH [8], [9], [14] and [15] or as E2E [7]. However, the main problem with those approaches is the large area overhead of such mechanisms while providing only limited fault coverage.

C. Relaxed Transmission (RT)

In contrast to above mentioned approaches which try to correct the fault, Relaxed Transmission (RT) uses the transient nature of the SETs and METs in order to mitigate them. The main idea of this approach is to inform one party in the transmission to wait until the faults disappear. A relaxed transmission mechanism has been proposed in [12], where the upstream router predicts the faults and informs the downstream router to delay data sampling. Even though the RT method is very efficient in mitigating faults, prediction of transient faults which have a random nature, will not be effective. Hence, in [12] also other methods have been used alongside RT.

This paper proposes an ultra-lightweight, HBH, relaxed transmission based transient fault mitigation approach called AWAIT which does not require any additional buffers. The proposed mechanism can handle transient faults immediately by pausing the operation of the faulty components without data loss and guarantees returning to normal, fault free operation in zero to one clock cycle after the fault disappears. This is achieved by utilizing a parity checker as the fault detection mechanism which allows to detect and mitigate all SETs. However, the proposed AWAIT mechanism is not limited to using parity checker but can be used together with any other fault detection mechanism. Since packet retransmission is not required, the AWAIT mechanism is also power efficient.

A side-by-side comparison of different fault tolerance mechanisms for physical links in NoCs proposed in the literature can be seen in Table I.

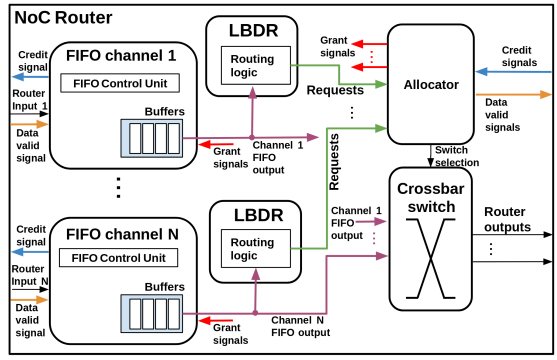
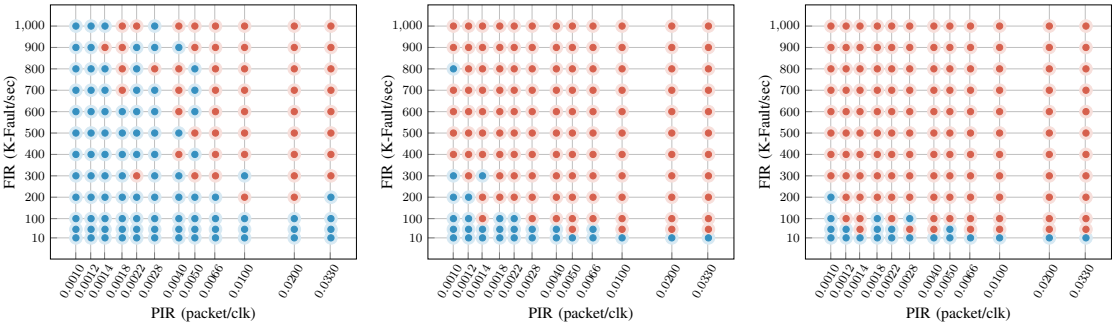


Fig. 1: Block diagram of the baseline Bonfire router

III. Bonfire Router

In order to prove the applicability of the AWAIT mechanism in a real design, it was implemented for inter-router links in a 4×4 NoC utilizing the Bonfire router [2]. An overview of the baseline Bonfire NoC router (without any fault tolerance mechanisms) can be seen in Fig. 1. The Bonfire router utilizes wormhole switching and 32-bit flit size.

The Bonfire router uses a credit-based flow control mechanism, where the transmitter includes a credit counter to keep track of the free slots in the receiver's input buffer. When



(a) Fault duration: 10% of clock period (b) Fault duration: 100% of clock period (c) Fault duration: 200% of clock period

Fig. 2: Effect of faults on a 4×4 2D-mesh baseline network during 100,000 clock cycles

initialized, the credit counter is set to equal to the number of slots in the receiver's input buffers. The value in the credit counter is decremented each time a flit is sent (and an additional slot in the receiver's input buffer gets occupied). When a slot frees up in the receiver's input buffer, the receiver will issue a credit signal to the transmitter, which causes the credit counter in the transmitter to increase. When credit counter reaches zero, the receiver's input buffer is full, and the transmitter will stop transmitting. If the value in the credit counter is larger than zero, data can be transmitted one flit per clock cycle.

Each input port of the Bonfire router consists of an input buffer, implemented as a First-In-First-Out (FIFO) and a routing computation unit which is implemented as Logic-Based Distributed Routing (LBDR) [16]. The main advantage of using LBDR compared to routing tables is its scalability. Additionally, since LBDR describes the network topology and routing algorithm using a fixed number of connectivity and routing bits, it is possible to easily mark the links as healthy or faulty, by disabling routing to faulty links. It is also easy to change the routing algorithm using the routing bits.

The output ports of the router are allocated by the allocator unit based on the routing decisions from LBDR. The allocator unit uses the Round-Robin policy for prioritizing multiple requests to the same output port. Finally, data is transferred from the input port to the output port using a crossbar switch.

IV. Network Failure Analysis

On-chip networks are very sensitive to faults on the network's physical links. A single faulty value stored in a register may lead to a network wide failure. We define a *network failure* as the situation were the entire network or part(s) of the network is congested (and the system cannot recover from it) due to inability to route a packet. Such a faulty value stored in a register can be one of the following:

- **Fault in the flit type:** might lead to a network-wide congestion
- **Fault in the destination address:** might lead to a network-wide congestion
- **Fault in other header information:** a fault in source address, packet length, etc. may cause problem at the application layer but has no effect on the network behavior, thus the effect of these faults can be ignored for the network.
- **Fault in the payload:** has no-effect on the network behavior. It might cause a problem at the application layer. The effect of these faults can be ignored for the network.

In this work, we have performed fault injection experiments to evaluate the effects of these faults on the network's behavior. The faults were randomly injected into the network links by temporarily forcing signals in the simulation to faulty values at fault rates ranging from ten thousand to one million faults per second. In order to see the effects of SETs with different lengths in the network, three different fault length scenarios were investigated where the fault length varied from 10% of the clock period to 100% and 200% of clock period (the fault effectively stayed on the link for two clock cycles). For this investigation, in total 1500 experiments were performed using random traffic pattern with different Packet Injection Rates (PIR) and Fault Injection Rates (FIR).

Fig. 2 depicts behavior of a 4×4, 2D-mesh network with wormhole switching using 4-flit FIFO buffers, without any fault tolerance mechanisms under different packet and fault injection rates (during a period of 100 thousand clock cycles). The fault injection rate is between ten thousand to one million faults per second. The red dots mark a network failure while blue dots depict the cases where the network successfully transmitted all of the injected packets. Each data point in this plot is based on 5 experiments with the same fault rate and packet injection rate but with different seeds for the random traffic generators. A network failure is declared only if at least 3 out of 5 experiments for that data point failed. As expected, with increasing packet injection rate and fault injection rate, the probability of a fault hitting a sensitive part of the packet increases. The experiments also show that the duration of fault's presence in the network has a great effect on the network behavior. This is due to the fact that faults

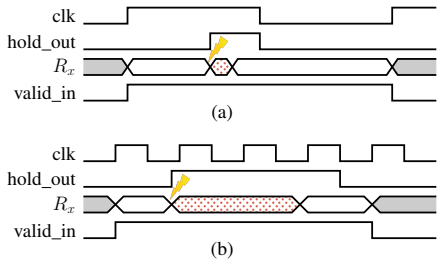


Fig. 3: Status of the signals during a) short (10% clk period) and b) long (200% clk period) transient fault

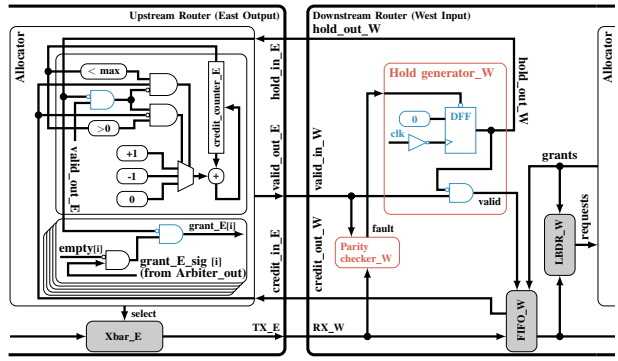


Fig. 4: Schematic of the AWAIT mechanism

with longer presence have higher chance of being registered in the FIFO buffers.

These experiments highlight the network's sensitivity to transient faults and the need for a scalable mitigation mechanism.

V. AWAIT Mechanism

In this paper AWAIT, a new light-weight fault mitigation mechanism is proposed to address transient faults on network links. The AWAIT mechanism operates on hop-by-hop (HBH) basis which enables fault correction with considerably lower latency when compared to end-to-end (E2E) packet retransmission. Also, unlike approaches like HBH flit/packet re-transmission, the proposed mechanism does not require additional buffers.

The AWAIT fault mitigation approach for network links is based on the transient nature of the SETs. The upstream router holds correct data when a transient fault occurs on the router link. Once the transient fault disappears, the data on the link comes back to its normal value and can be read by the downstream router's FIFO. In order to wait out the transient fault, three steps are required:

1) The downstream router should detect the presence of fault(s).
2) The downstream router should stop sampling data in case of a fault.
3) The upstream router must keep the transmitted data on the link until it is certain that the downstream router has received the correct, fault-free data.

In order to provide step one, a simple fault detection system – a parity checker – is used. Parity checker is especially useful since it can detect all single faults.

In order to support step two, the process of storing the data into downstream router's FIFO is stopped if a fault is detected. This is achieved by replacing the *valid_in* signal of the FIFO module with a new signal: $\texttt{valid} = (\texttt{valid\_in} \bigwedge \overline{\texttt{fault}})$.

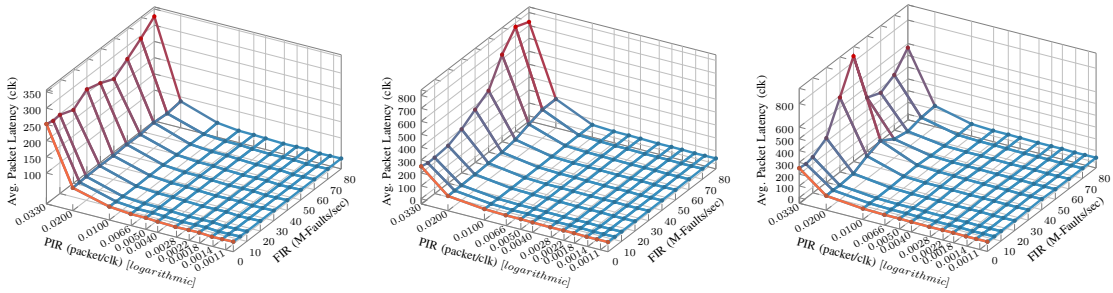
Finally, to support step three, a hold signal is propagated from the downstream router to the upstream router, informing it about the presence of a fault on the link. Upon receiving this signal, the upstream router's allocator unit halts transmission and maintains the value on the link. The AWAIT mechanism does not require any additional buffers in the system and is entirely governed via controlling of the grant signals. The *hold* signal can only be cleared once the fault has disappeared. It will be done on the next falling clock edge after the disappearance of the fault, enabling the system to register the correct data on the next rising clock edge.

Fig. 3 depicts the behavior of the mechanism for a short (10% of the clock period) and a long (200% of the clock period) transient fault, respectively. Rx signal in the figure represents the data receiving line of the router. The duration while the Rx signal had a faulty value is colored red. The *valid_in* signal describes the valid signal received from the upstream router (showing upstream router maintaining the value on the link). It is important to note that if the fault disappears before the falling edge of the system clock, the effects of the fault will be mitigated in the same clock cycle. However, if the fault disappears after the falling edge of the clock, the correct data will be latched one clock cycle later during the next falling clock edge.

Fig. 4 shows the block diagram of the output port of an upstream router, the communication link, and the input port of a downstream router along with the added circuitry. For simplicity, Fig. 4 shows only the East output of the upstream router, connected to the West input of the downstream router. Packets are being transmitted from the upstream router to the downstream router.

The parity checker in the downstream router's input checks the parity of the received data. The output of the parity checker is used to trigger the hold signal in the downstream router. When a fault is detected, the *hold_out* signal is instantly set to *"1"*. Using a flip-flop ensures that the *hold_out* signal stays at the *"1"* value, using asynchronous reset of the D flip-flop, until the fault has disappeared. Once the fault has disappeared, at the next falling edge of the system's clock, the value of *hold_out* signal is set to *"0"*.

Additional components are added to the mechanism in order to control the credit counters and the grant generation in the allocator unit. Upon receiving the hold signal, allocator unit suppresses the grants issued to the FIFO unit which is



(a) Fault duration: 10% of clock cycle (b) Fault duration: 100% of clock cycle (c) Fault duration: 200% of clock cycle

Fig. 5: Average packet latency of 4×4 2D-mesh network under different PIR and FIR on the links

accessing the output channel. This means that FIFO will not perform any read operations and the latest data will be kept on its output. Allocator unit will also keep the select signal of the Xbar intact, ensuring that the data is kept unchanged on the link and it will not update the value of the credit counter as long as the hold signal is present. The additional hardware for supporting these mechanisms is minimal and does not exceed a few gates. Next section will provide experimental results for illustrating the AWAIT mechanism's efficacy under different fault injection campaigns.

VI. Experimental Results

This section details the experiments for validating the effectiveness of the proposed AWAIT mechanism and providing support for its efficacy in mitigating SETs. Later in this section the overheads of the AWAIT mechanism are investigated in terms of area, power and critical path delay.

The experiments were carried out on a 4×4, 2D-mesh NoC with wormhole switching using 4 flit deep FIFO buffers. Each router in the network is a Bonfire credit-based router equipped with the AWAIT mechanism for handling transient faults. For measuring the latency of the AWAIT mechanism, 1500 experiments were performed using random uniform traffic pattern with different Packet Injection Rates (PIRs) and Fault Injection Rates (FIRs) – up to 80 Million random faults per second over the network's physical links. The experiments shows no failing scenario and validates the effectiveness of the AWAIT approach. In other words, despite of the existence of transient faults on the router's link at run-time, the number of received packets over the network match the number of sent packets. Fig. 5 depicts the behavior of the fault-tolerant NoC equipped with the AWAIT mechanism under different fault campaigns on network links. Similar to experiment on the baseline router (see Fig. 2), three different fault duration scenarios were used. Each data point in this plot is the average latency of 5 experiments, all of which use the same FIR and PIR, but have different seeds for the random packet generators. The experiments illustrate that the network is still operational under extreme fault conditions (fault injection rate from 5-Million to 80-Million faults per second) with acceptable latency overhead. The experiments show that while the network is not saturated, the length of the faults does not have much effect on the additional network latency from fault handling. However, once the network exceeds saturation point, the additional latency becomes more visible. The orange line in Fig. 5 marks the reference, fault-free experiment.

TABLE II: Area overhead of the AWAIT mechanism

	Area (μm^2)	Area Overhead
Baseline router	8289.38	–
AWAIT router	8719.56	5.1 %

Table II shows the area overhead imposed to the baseline router by the AWAIT mechanism. Both designs are implemented in Register Transfer Level (RTL) in VHDL and are synthesized using TSMC 40 nm CMOS technology standard cell library in Synopsys Design Compiler. It is important to note that the fault detection mechanism (parity checker) imposes the majority of the reported combinational area overhead (3.82%) due to the additional XOR gates required to perform the fault detection. It should be noted, however, that this area overhead is still very small when compared to the overhead implied by an error correction code such as Hamming [2]. The rest of the mechanism, including the additional gates imposed by the proposed mechanism, incurs only 1.3% overhead to the router's area, which makes the solution scalable.

Fig. 6 shows the comparison of the dynamic and static power consumption of a 4×4 network augmented with the AWAIT mechanism against the baseline router under different PIRs. In order to obtain the power results for each PIR, the design architecture was simulated in ModelSim. Then, the simulation traces were collected and the switching activity of the components and signals were stored. Further, the annotated switching activities are fed into the synthesis tool to calculate the power results. The AWAIT mechanism has only a negligible total power consumption overhead (around 2.89% on average), compared to the baseline router. This is mostly caused by increase in the dynamic power consumption.

Table III shows the critical path delay overhead of the AWAIT mechanism compared to the baseline router. The proposed mechanism does not impose overhead to the router's critical path delay. Due to the critical path delay of the AWAIT mechanism the faults that occur very close to the clock edge

TABLE III: Critical path delay overhead of the AWAIT mechanism

	Critical-Path Delay (ns)	Overhead
Baseline router	1.96	–
AWAIT router	1.94	$\approx 0\%$

will be latched into the FIFO buffers. However, in systems with frequency of 250 MHz and below these effects are negligible. Additionally, it should be noted that since the area and timing overhead of the AWAIT mechanism is very small, it can be coupled with other fault mitigation mechanisms which can handle faults already latched into the buffer (such as a packet dropping mechanism [2]) to solve the issue. The source code for both the baseline and the AWAIT router design along with all the experimental result scripts are maintained as an open-source project [17].

VII. Conclusion

Network-on-Chip has become a widely accepted communication paradigm, replacing the bus-based communication mediums in modern multi/many-core System-on-Chips. However, the shrinking transistor's feature size and increase in the number of components integrated on a single chip, makes the systems growingly more susceptible to transient faults. On-chip networks are particularly sensitive to transient faults on the physical links between the nodes. This paper presented the ultra-lightweight AWAIT mechanism which can tolerate all single event transients on network links. The proposed AWAIT mechanism adds only 5.1% overhead to the area and only a negligible overhead to the critical path delay of the routers, which makes it a scalable solution, as it adds transient fault tolerating capability to the design by augmenting it with only a few gates. Experimental results show the effectiveness of the AWAIT mechanism even in extreme environments. As future work, the authors plan to extend this work to cover permanent faults on the router links as well. Moreover, the mechanism would be extended to cover faults occurring in the control part of the router.

Acknowledgment

The work has been supported by EU's Twinning Action TUTORIAL, Estonian Institutional research grant IUT 19-1, Estonian center of excellence in IT EXCITE, and Estonian IT Academy programme.

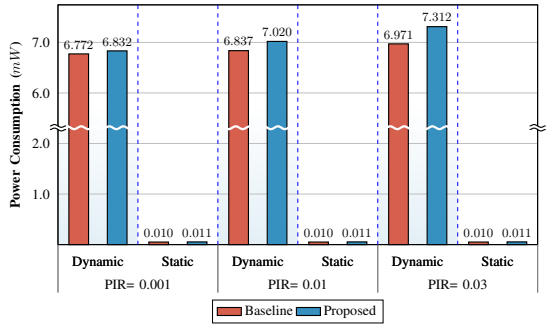


Fig. 6: Comparison of dynamic and static power consumption (in mW) of the proposed and baseline 4×4 network under different packet injection rates

References

[1] D. Lorenz, G. Georgakos, and U. Schlichtmann, "Aging analysis of circuit timing considering NBTI and HCI," in *2009 15th IEEE International On-Line Testing Symposium*, June 2009, pp. 3–8.

[2] S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in Network-on-Chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, April 2017, pp. 48–53.

[3] G. Schley, N. Batzolis, and M. Radetzki, "Fault localizing End-to-End flow control protocol for Networks-on-Chip," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2013, pp. 454–461.

[4] E. A. Rambo, C. Seitz, S. Saidi, and R. Ernst, "Designing Networks-on-Chip for high assurance real-time systems," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, Jan 2017, pp. 185–194.

[5] E. Wachter, V. Fochi, F. Barreto, A. Amory, and F. Moraes, "A hierarchical and distributed fault tolerant proposal for NoC-based MPSoCs," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2017.

[6] M. Ali, M. Welzl, S. Hessler, S. Hellebrand, and S. And, "An efficient fault tolerant mechanism to deal with permanent and transient failures in a network on chip," *International Journal of High Performance Systems Architecture*, vol. 1, 01 2007.

[7] S. Shamshiri, A. Ghofrani, and K. T. Cheng, "End-to-end error correction and online diagnosis for on-chip networks," in *2011 IEEE International Test Conference*, Sept 2011, pp. 1–10.

[8] S. Ogg, B. Al-Hashimi, and A. Yakovlev, "Asynchronous transient resilient links for NoC," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2008, pp. 209–214.

[9] A. P. Frantz, F. L. Kastensmidt, L. Carro, and E. Cota, "Dependable Network-on-Chip router able to simultaneously tolerate soft errors and crosstalk," in *2006 IEEE International Test Conference*, 2006, pp. 1–9.

[10] S. R. Naqvi, V. S. Veeravalli, and A. Steininger, "Protecting an asynchronous NoC against transient channel faults," in *2012 15th Euromicro Conference on Digital System Design*, Sept 2012, pp. 264–271.

[11] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das, "Exploring fault-tolerant Network-on-Chip architectures," in *International Conference on Dependable Systems and Networks (DSN'06)*, June 2006, pp. 93–104.

[12] D. DiTomaso, T. Boraten, A. Kodi, and A. Louri, "Dynamic error mitigation in NoCs using intelligent prediction techniques," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[13] A. Dutta and N. A. Touba, "Reliable network-on-chip using a low cost unequal error protection code," in *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, Sept 2007, pp. 3–11.

[14] X. Chen, Z. Lu, Y. Lei, Y. Wang, and S. Chen, "Multi-bit transient fault control for NoC links using 2D fault coding method," in *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, Aug 2016, pp. 1–8.

[15] A. P. Frantz, M. Cassel, F. L. Kastensmidt, É. Cota, and L. Carro, "Crosstalk- and SEU-aware Networks on Chips," *IEEE Design Test of Computers*, vol. 24, no. 4, pp. 340–350, July 2007.

[16] J. Flich and J. Duato, "Logic-based distributed routing for NoCs," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 13–16, Jan 2008.

[17] "AWAIT," https://github.com/Project-Bonfire/AWAIT, 2018.

Appendix 2

II

K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "Handling of SETs on NoC links by exploitation of inherent redundancy in circular input buffers," in *2018 16th Biennial Baltic Electronics Conference (BEC)*, pp. 1–4, Oct 2018

Handling of SETs on NoC Links by Exploitation of Inherent Redundancy in Circular Input Buffers

Karl Janson, René Pihlak, Siavoosh Payandeh Azad, Behrad Niazmand, Gert Jervan, Jaan Raik
Department of Computer Systems, Tallinn University of Technology,
email: {karl.janson, rene.pihlak, siavoosh.azad, behrad.niazmand, gert.jervan, jaan.raik}@ttu.ee

***Abstract*—The miniaturization of nanometer technologies beyond the sub-micron domain has jeopardized the reliability of on-chip network links, making them more susceptible to Single Event Transients (SETs) during system's run-time. Using retransmission approaches has been proposed in the literature for handling SETs in Network-on-Chip (NoC) links. However, those approaches either suffer from significant latency overhead or impose additional retransmission buffers, which consume more area. This paper proposes the ReUSE mechanism as a transient fault mitigation mechanism for Network-on-Chip links. The mechanism takes advantage of the inherent redundancy in the input buffers of NoC routers and reuses these for SET mitigation on the NoC links. By using a parity checker for fault detection, the approach can cover all SETs during run-time and return to normal operation in maximum one clock cycle after the disappearance of the fault. The experimental results show that the proposed mechanism prevents network-wide failure even in harsh environments with up to 80 million random faults on links per second. The ReUSE mechanism imposes 18% area overhead and 7.8% critical path delay overhead to the baseline NoC router.**

***Index Terms*—Network-on-Chip, Fault tolerance, Transient fault mitigation**

I. Introduction

The ongoing trend of miniaturization of the semiconductor technology beyond the sub-micron domain has made digital circuits more susceptible to faults, including Single Event Transients (SETs) [1]. As SETs are also caused by evironental factors, such as radiation, systems operating in harsh environments with high radiation levels, such as space applications, are especially at risk of SETs. Such faults occurring during run-time can affect the performance of embedded systems, including Network-on-Chip (NoC)-based System-on-Chips (SoCs). A faulty link in a NoC can corrupt the data transmitted between processing elements over the network. Thus, it is important to handle such SETs in inter-router links at run-time.

This paper focuses on management of SETs in NoC links. This is done by implementing a very low latency hop-by-hop (HBH) retransmission mechanism. The proposed ReUSE mechanism does not use additional registers and can cover 100% of SETs inter-router links by using the inherent redundancy provided by the unusable memory slot found in the circular buffers [2] used in the NoC router input ports. The detection of SETs at each router input is performed online via a single bit parity checker. Experimental results show the effectiveness of the proposed mechanism in terms of area overhead, critical path delay overhead and power consumption.

The rest of this paper is organized as follows: Section II provides information about related works, Section III provides details on the proposed ReUSE mechanism and Section IV will provide results of the experiments which help to illustrate the proposed mechanism's efficiency. Finally, section V will conclude the paper.

II. Related Work

The issue of Single Event Transients (SETs) on the NoC links has been addressed in several papers. Depending where and when SET detection and/or correction takes place, the approaches can be divided into End-to-End (E2E) and Hop-by-Hop (HBH) mechanisms.

In case of E2E approaches, such as [3], [4] and [5], fault detection and/or correction takes place at the final destination, while HBH approaches such as [6], [7] and [8] are designed to avoid fault propagation by fault mitigation in each router. The E2E approaches, however, are unable to handle errors in the packet header before the packet has reached the destination, meaning, they cannot guarantee the arrival of the packet to the final destination where fault detection/correction takes place and, moreover, can cause network congestions due to misrouting. Therefore, the proposed approach deploys a HBH approach.

In addition to that, the SET fault tolerance mechanisms can be further divided into three groups based on the type of mechanism used: *1)* Error correction codes (ECC); *2)* Relaxed transmission (RT); *3)* Packet/flit retransmission (R/F-RET).

Error correction code based approaches such as [9], [6] and [10] usually imply a large area and data overhead.

Relaxed transmission based approaches such as AWAIT proposed in [11] detect a fault on the link by using a fault detection mechanism, such as a parity checker. When a fault is detected the system is paused until the SET disappears. However this approach cannot handle faults which happen near the raising clock edge, since fault detectors (such as parity) will not provide the result instantaneously due to signal propagation delays in the gates. If the time between the SET and the next rising clock edge is shorter than the fault detection latency, the faulty value will be latched into the router's input buffer. With the use of high clock rates in modern systems the probability of a fault being latched into the buffer raises noticeably, thus limiting the maximum safe speed of systems using the RT approach.

Retransmission-based approaches use fault detection mechanisms and trigger retransmission of the data in case a fault is detected.

978-1-5386-7312-6/18/$31.00

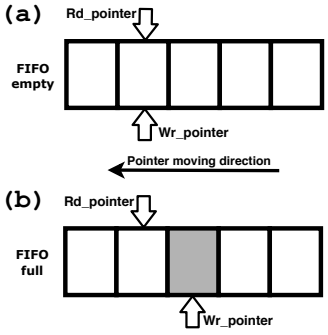


Fig. 1: A FIFO which is implemented as a circular buffer (a) empty, (b) full

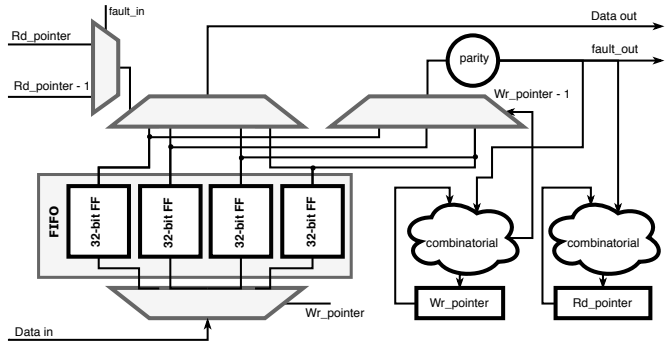


Fig. 2: Schematic of ReUSE FIFO implementation

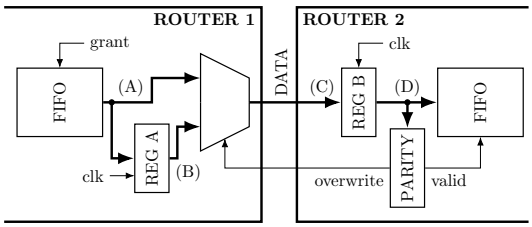


Fig. 3: Schematic of the dual register version of the ReUSE mechanism

Retransmission approaches can be categorized as packet retransmission (P-RET) or flit retransmission (F-RET). HBH P-RET approaches, such as [12] suffer from large area and slow speed, since routers need large buffers to store the entire packet. Additionally, in case of a fault, the entire packet needs to be retransmitted. In case of F-RET approaches, like [13] and [14], only one flit is retransmitted in case of the fault, thus requiring much smaller buffer and completing retransmission much faster. However, usually those approaches still have a relatively large overhead, since they require additional registers for retransmission.

In addition, the approach proposed in [15] has some similarities with the proposed mechanism. It uses Razor flip-flops[16] to both perform online error detection and to restore the correct value of a signal after detection. The goal of the approach is to tolerate SETs on NoC inter-router links. However, a delayed clock is used in order to obtain the delayed sample of the signal.

The ReUSE approach proposed in this paper is able to detect 100% of SETs by utilizing a parity checker. However, as it is based on HBH flit retransmission, it can, unlike [11], handle all detected faults while guaranteeing returning of normal, fault free, system operation in at most one clock cycle after disappearance of the fault. Moreover, the ReUSE mechanism does not use any additional registers, but relies on inherent redundancy already found in FIFO input buffers of NoC routers, thus not requiring increase in the area.

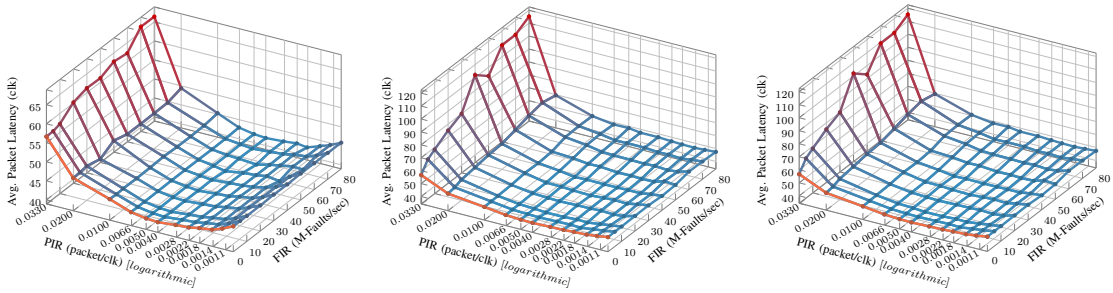
III. ReUSE Mechanism

A. General Concept

ReUSE is a HBH flit retransmission mechanism for handling SETs on the NoC links. A simplified concept of the ReUSE mechanism utilizing two additional registers can be seen in Fig. 3. For simplification, signals used for flow control, and also router components responsible for routing decisions and arbitration are not shown in the figure. In the figure data is transmitted from *Router 1* to *Router 2*.

On the transmitter's (*Router 1*) side, each clock cycle, a flit is transferred to *Router 2* (receiver) (A) and also stored into register *REG A*. Thus, value on the *DATA* line depends on the multiplexor's (MUX) select line. If (A) input is selected, *DATA* will have the current flit read from the FIFO on it, however, if the (B) input is selected, *DATA* will be set equal to the flit which was read from the FIFO during the previous clock cycle and stored into *REG A*. By combining this with a mechanism for pausing the operation of the transmitting router, it is possible to keep the already sent flit on the DATA line until a fault disappears. While the pausing mechanism is not specified in this paper, an example of such a mechanism can be seen in [11].

On the receiver side (*Router 2*) every time a flit is received over link (C), it is stored into a register *REG B*. The value in the register is checked by a parity checker. If the parity checker detects a fault, it will cause the sending router to be paused and the MUX to be switched to input (B) using the "overwrite" line, thus holding the current flit on the *DATA* line. The *DATA* line is sampled to *REG B* every clock cycle and rechecked by the parity checker. Once the fault has disappeared, the operation of *Router 1* will be resumed and the MUX will be switched back to input (A), and writing to receiving router's FIFO is enabled by a signal from the parity checker. Since the data in register *REG B* is updated each clock cycle, the mechanism is guaranteed to return to normal operation on the next rising edge of the clock after the fault has disappeared.



(a) Fault duration: 10% of clock cycle (b) Fault duration: 100% of clock cycle (c) Fault duration: 200% of clock cycle

Fig. 4: Average packet latency of a 4×4 2D-mesh network under different PIR and FIR on the links

B. ReUSE Concept

Since *REG B* in Fig. 3 can be thought of as an extension to *Router 2*'s FIFO, the design can be optimized by using data written to the FIFO one clock cycle earlier as input to the parity checker, thus removing the need for *REG B* as a separate register. This can be easily implemented, assuming a circular buffer implementation of the FIFO, as shown in Fig. 1. The buffer is accessed in such a FIFO implementation by using two pointers; one for reading ($Rd\_pointer$), other for writing ($Wr\_pointer$). If the buffer is empty, the read and write pointers will both point at the same memory slot, as seen in Fig. 1 (a). After each writing operation, the write pointer is shifted by one slot to the left. It can be seen that *REG B* can be implemented in such a way that parity checker is connected to the FIFO slot referred to by $Wr\_pointer - 1$.

The data is read from the memory slot referred by the read pointer. However, as it can be seen in Fig. 1 (b), the condition for the FIFO being full is when the read buffer is ahead of the write buffer by one memory slot ($Wr\_pointer = Rd\_pointer - 1$). Since writing would cause the write pointer to move to the same memory slot where the read pointer is (creating the "empty" configuration), the memory slot $Rd\_pointer - 1$ can never be overwritten in such an implementation. This definition helps to further optimize the mechanism, since *REG A*, as mentioned in Subsection III-A, stores the flit read from the FIFO during the previous clock cycle. This means that the flit in *REG A* is the same as the one stored in FIFO memory slot $Rd\_pointer - 1$.

The ReUSE mechanism takes advantage of the aforementioned optimizations in order to provide fault tolerance for NoC links. The implementation of the ReUSE FIFO can be seen in Fig. 2, data is normally read from the FIFO slot referred by $Rd\_pointer$ and written into the slot referred by $Wr\_pointer$. However, *REG B* has been implemented by connecting the parity checker to an additional MUX, which uses the previous write pointer as select line. If a fault is detected by the parity checker, the fault output is activated. This will cause the sending router's operation to be paused until the fault has disappeared, as discussed earlier, and reading of the memory slot referred by $Rd\_pointer - 1$, thus also removing the need for *REG A*.

TABLE I: Area overhead of the ReUSE mechanism

	Area (μm^2)	Overhead
Baseline router	8276.45	–
ReUSE router	9777.26	18%

TABLE II: Critical path delay overhead of the ReUSE mechanism

	Critical path Delay (ns)	Overhead
Baseline router	2.28	–
ReUSE router	2.46	7.8%

IV. Experimental Results

This sections presents results of the experiments used to validate the proposed ReUSE mechanism. The experimental results will also demonstrate the SET mitigation efficiency of the proposed mechanism. Additionally, in this section the overhead of the proposed ReUSE mechanism will be investigated in terms chip area, power and critical path delay.

For carrying out the experiments, the ReUSE mechanism was implemented for the links of a 4×4, 2D-mesh NoC utilizing the open source Bonfire NoC router [3]. The Bonfire router uses wormhole switching with 32-bit flit width. In order to reduce the area overhead, it uses Logic Based Distributed Routing (LBDR) [17] and credit-based flow control, which has the capability of transferring up to one flit per clock cycle, assuming there are free slots in the receiver's input buffer. The version of the Bonfire router used in this work does not use virtual channels. For calculating the overhead of the proposed mechanism, all experimental results were compared to a baseline Bonfire router which does not include any fault tolerance mechanisms. The source code for both the baseline router and for the proposed ReUSE router design along with all the scripts used for calculating the experimental results are maintained as an open-source project available at [18].

In order to measure the applicability of the ReUSE mechanism in harsh environments (like space), a set of random uniform fault injection experiments were run for SETs lasting 10%, 100% and 200% of the clock period. The experiments were run with different network loads and for fault rates up to 80 million faults per second. The faults were injected by forcing the signals in Modelsim simulation with faulty values.

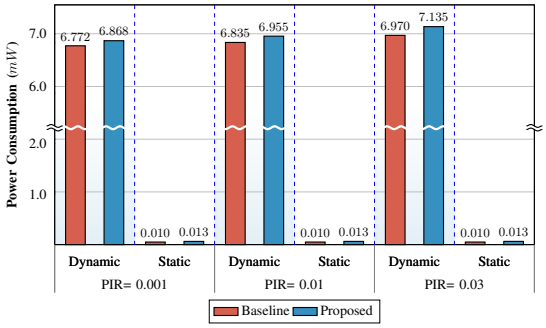


Fig. 5: Comparison of dynamic and static power consumption (in mW) of the proposed and baseline 4×4 network under different packet injection rates

The results of the experiments in Fig. 4 show that when compared to the fault-free run (shown with an orange line in the figure), the latency did not change much when faults were injected, even on under extreme fault injection rates.

The area overhead the ReUSE mechanism imposes on the baseline Bonfire router can be seen in Table I. Table II shows the critical path delay overhead. In order to obtain those results, both designs were implemented in Register Transfer Level (RTL) in VHDL and synthesized using TSMC 40 nm CMOS technology standard cell library in Synopsys Design Compiler, for 400 MHz clock frequency.

Finally, Fig. 5 shows the power usage of dynamic and static power of the ReUSE mechanism, when compared to the baseline. As it can be seen in the figure, the increase in power usage implied by the ReUSE mechanism is minimal.

V. Conclusion

Due to the trend of shrinking transistors' feature size, Network-on-Chip links have become more susceptible to run-time Single Event Transients (SETs), thus, affecting the operation of the entire system. This paper presented the ReUSE mechanism for handling SETs on inter-router links. To this end, the inherent redundancy in the input buffers is utilized, avoiding the usage of any additional re-transmission buffers when mitigating SETs at run-time. The proposed mechanism only adds 18% area overhead and 7.8% critical path delay overhead to the baseline router (router without any fault tolerance mechanisms). Experimental results show the effectiveness of the ReUSE mechanism even in extreme fault environments. As future work, the authors plan to extend this work to cover permanent faults on the router links as well. Moreover, the mechanism would be extended to cover faults occurring in the control part of NoC routers.

Acknowledgment

The work has been supported by EU's Twinning Action TUTORIAL, Estonian Institutional research grant IUT 19-1, Estonian center of excellence in IT EXCITE, and Estonian IT Academy programme.

References

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[2] N. Ni, M. Pirvu, and L. Bhuyan, "Circular buffered switch design with wormhole routing and virtual channels," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*, Oct 1998, pp. 466–473.

[3] S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in Network-on-Chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, April 2017, pp. 48–53.

[4] E. A. Rambo, C. Seitz, S. Saidi, and R. Ernst, "Designing Networks-on-Chip for high assurance real-time systems," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, Jan 2017, pp. 185–194.

[5] E. Wachter, V. Fochi, F. Barreto, A. Amory, and F. Moraes, "A hierarchical and distributed fault tolerant proposal for NoC-based MPSoCs," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2017.

[6] X. Chen, Z. Lu, Y. Lei, Y. Wang, and S. Chen, "Multi-bit transient fault control for NoC links using 2D fault coding method," in *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, Aug 2016, pp. 1–8.

[7] C. Feng, Z. Lu, A. Jantsch, M. Zhang, and Z. Xing, "Addressing transient and permanent faults in NoC with efficient fault-tolerant deflection router," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 6, pp. 1053–1066, June 2013.

[8] Q. Yu and P. Ampadu, "A dual-layer method for transient and permanent error co-management in NoC links," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 58, no. 1, pp. 36–40, Jan 2011.

[9] S. Ogg, B. Al-Hashimi, and A. Yakovlev, "Asynchronous transient resilient links for NoC," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2008, pp. 209–214.

[10] S. Shamshiri, A. Ghofrani, and K. T. Cheng, "End-to-end error correction and online diagnosis for on-chip networks," in *2011 IEEE International Test Conference*, Sept 2011, pp. 1–10.

[11] K. Janson, R. Pihlak, S. P. Azad, B. Niazmand, G. Jervan, and J. Raik, "Await: An ultra-lightweight soft-error mitigation mechanism for network-on-chip links," in *2018 13th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC) (Accepted for publication)*, July 2018.

[12] A. Dutta and N. A. Touba, "Reliable network-on-chip using a low cost unequal error protection code," in *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, Sept 2007, pp. 3–11.

[13] S. R. Naqvi, V. S. Veeravalli, and A. Steininger, "Protecting an asynchronous NoC against transient channel faults," in *2012 15th Euromicro Conference on Digital System Design*, Sept 2012, pp. 264–271.

[14] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das, "Exploring fault-tolerant Network-on-Chip architectures," in *International Conference on Dependable Systems and Networks (DSN'06)*, June 2006, pp. 93–104.

[15] R. R. Tamhankar, S. Murali, and G. De Micheli, "Performance driven reliable link design for networks on chips," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '05. New York, NY, USA: ACM, 2005, pp. 749–754. [Online]. Available: http://doi.acm.org/10.1145/1120725.1121009

[16] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, Dec 2003, pp. 7–18.

[17] J. Flich and J. Duato, "Logic-based distributed routing for NoCs," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 13–16, Jan 2008.

[18] "ReUSE reposity," https://github.com/Project-Bonfire/ReUSE, 2018.

Appendix 3

III
S. P. Azad, B. Niazmand, K. Janson, T. Kogge, J. Raik, G. Jervan, and T. Hollstein, "Comprehensive performance and robustness analysis of 2D turn models for Network-on-Chips," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May 2017

Comprehensive Performance and Robustness Analysis of 2D Turn Models for Network-on-Chips

Siavoosh Payandeh Azad *, Behrad Niazmand*, Karl Janson*, Thilo Kogge‡,
Jaan Raik*, Gert Jervan*, Thomas Hollstein*†
Department of Computer Engineering
*Tallinn University of Technology, † Frankfurt University of Applied Sciences, ‡Darmstadt University of Technology
email: {siavoosh, bniazmand, karl.janson , thilo, jaan.raik, gert.jervan, thomas}@ati.ttu.ee

***Abstract*—Routing algorithms play an important role in Network-on-Chip (NoC) based System-on-Chips. Turn model based routing disallows some of the turns in order to avoid deadlock, while providing partial adaptivity. In this paper, all 2D uniform turn models are examined for deadlock freeness and connectivity; 50 deadlock free turn models are extracted that provide full connectivity in the network. An extended adaptivity metric is introduced to classify the turn models; all extracted turn models are compared in terms of adaptivity, robustness and latency. Experimental results identify the most robust turn models and the most efficient ones in terms of latency.**

***Keywords*—Turn Model, Routing Algorithm, Robustness, Minimal Path, Network-on-Chip.**

I. Introduction

Network-on-Chip (NoC) has emerged as a paradigm to overcome some of the limitations existing in the conventional shared medium bus-based architectures [1], such as performance and scalability issues. In a NoC, the communication between cores is administered by on-chip routers based on a routing algorithm. Routing algorithms can be either classified as deterministic or adaptive [2]. Deterministic routing algorithms, use a single path for each source-destination pair, whereas adaptive routing provides more path diversity, taking into account criteria such as traffic load on links, etc. One of the important factors when choosing a routing algorithm is deadlock freeness. Deadlock occurs when a cyclic dependency is created between the packets in a NoC, waiting for resources held by other packets in the cycle [2].

Two main approaches exist for addressing deadlocks: deadlock avoidance where deadlocks are completely avoided and deadlock recovery where deadlock can happen, but is handled using a deadlock recovery mechanism. The focus of this paper is on deadlock avoidance. *Turn model* approach was first introduced in [3] for deadlock avoidance in 2D Mesh Network-on-Chips. A turn is defined as a change of direction in a packet's path. Directions are named based on cardinal directions: North (N), East (E), West (W) and South (S). In a 2D Mesh network, maximum of 8 *turns* exist: N2E, N2W, E2N, E2S, W2N, W2S, S2E and S2W. For instance S2E indicates a ↱ turn that, if allowed, it enables a packet coming from the South input port of the router be forwarded to the East output port. A total number of $2^8 = 256$ uniform 2D turn models can be derived from eight possible turns. In a uniform turn model, all network nodes have the same disallowed turns.

In [3], three deadlock free turn models are introduced, *i.e.* West-First, North-Last and Negative-First. Furthermore, [4] introduces the North-South First (NSF) turn model, by combining North-First and South-First. Moreover, the East-First turn model is addressed in [5].

Even though, some of the previous works such as [6] have covered performance comparison of some of the well-known turn models, to the best of our knowledge, exploration of the entire search space of all possible turn models for a 2D mesh NoC have not been thoroughly performed. In this paper, all 256 uniform turn models for routing in 2D Mesh NoCs are enumerated and the characteristics of the deadlock-free turn models are evaluated. The proposed approach uses metrics for connectivity and adaptivity, based on which turn models are classified. In addition, latency and robustness of all turn model groups are assessed.

The rest of this paper is organized as following: in section II a methodology for evaluation of deadlock freeness of turn models is discussed. In section III metrics for network connectivity and routing adaptiveness are introduced. In section IV robustness and latency of the chosen routing algorithms have been evaluated in minimal and non-minimal routing. Finally, section V concludes the paper.

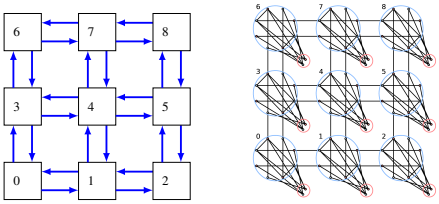
II. Evaluating Deadlock Freeness

In [7], the concept of Cyclic Dependency Graph (CDG) is used for deadlock detection, where nodes represent the network channels and edges denote the channels dependency. It is proven that for guaranteeing deadlock-freeness, CDG must be acyclic. In this work similar approach has been used, but instead of CDG, we use the concept of Routing Graphs (RGs) which was introduced in [8]. The rest of this section overviews the construction of routing graphs and describes the process of evaluating deadlock freeness.

A. Routing Graph

A Routing Graph, RG(V, E), is a directed graph, in which the set of vertices (V) denotes the set of all the input/output ports in the network (two nodes per port) and the set of edges (E) represent the set of (v_i, v_j) where v_i is a vertex (input or output port) that is depending on v_j port. For the sake of simplicity, a vertex v in RG is denoted as $node_{i,p,dir}$, which describes direction $dir \in \{in, out\}$ of port $p \in \{N, E, W, S, L\}$ of node i in the network. There are two different types of links represented as edges in the routing graph:

- *Inter-router edges*, representing connections between routers (from an output port of a router to an input port of an adjacent router).



(a) (b)

Fig. 1. a) Example of a 3 × 3 mesh network and b) the resulting routing graph for XY routing algorithm

- *Intra-router links*, representing allowed connections inside the router (from an input port of a router to an output port in the same router). An intra-router link can be: 1) From or to local port: representing dependency between the router's north, east, west and south ports and the local port connected to the processing element (PE). 2) Straight connections: describing dependency between ports involved in maintaining straight connections inside the router (e.g from west input to east output port of a router). 3) Turns: dependency of the ports in perpendicular direction (e.g. from east input to south output port).

As an example, Fig. 1b shows the RG for XY routing for a 3x3 2D Mesh network, corresponding to Fig. 1a. In order for a routing algorithm to be deadlock free, its corresponding RG must also be acyclic.

B. Proof of Deadlock Freeness

Theorem 1. *A deadlock in a turn model results in a cycle in the RG derived from the turn model.*

Proof. Let us assume that a deadlock in the turn model results in a RG with no cycles. Since the RG represents a sequence of all dependencies between the inputs and outputs of a routers under the applied turn model and there are no cycles in RG, there cannot be cyclic dependencies between the inputs and outputs of routers in the network. Hence, no deadlock can be formed. This is in contradiction with our initial assumption of having a deadlock for the turn model. This means that a deadlock in a turn model results in a cycle in the RG derived from the turn model. □

Using this method, it is possible to discard 35 turn models that have deadlocks, which leaves 221 deadlock free turn models.

III. Metrics for Connectivity and Adaptivity

Out of 221 deadlock free turn models, some provide partial connectivity. Examples of this case are: one turn model with zero turns and 8 turn models with one turn. To evaluate the connectivity of turn models, a simple metric has been used:

$$Connectivity_{RG} = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C_{i,j,RG} \tag{1}$$

where N is number of nodes in the network and:

$$C_{i,j,RG} = \begin{cases} 1 & \text{if exists a path from } node_{i,L,out} \text{ to } node_{j,L,in} \\ & \text{in } RG \text{ where } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

TABLE I
List of previously named turn models

#	Allowed turns	Conventional Name
0	E2N, E2S, W2N, W2S	XY [9]
13	S2W, S2E, N2W, N2E	YX [10]
33	E2S, S2W, S2E, N2W, N2E	Restricted North First [4]
39	E2N, E2S, W2N, W2S, S2W, N2W	East-First [5]
40	E2N, E2S, W2N, W2S, S2E, N2E	West-First [3]
41	E2N, E2S, W2N, W2S, N2W, N2E	North-Last [3]
42	E2N, E2S, W2N, S2E, N2W, N2E	Negative-First [3]
46	E2N, W2N, S2W, S2E, N2W, N2E	South-First [4]
48	E2S, W2S, S2W, S2E, N2W, N2E	North-First [4]

TABLE II
DoA and DoA_{Ex} for all 2D routing algorithms of Fig. 2

	4 turns			5 turns		6 turns	
Turn Model Num	0, 13	3, 5, 8, 10	1, 2, 4, 6, 7, 9, 11, 12	14, 15, 16, 17, 28, 33, 36, 37	18-27, 29-32, 34, 35	42, 43, 45, 47	38, 39, 40, 41, 44, 46, 48, 49
DoA	1			1.23		1.43	
DoA_{Ex}	1	1.41	1.63	2.11	2.41	3.83	4.33

and RG is representing the routing graph. In this case, the assumption is that *RG* represents a full mesh under a turn model based routing algorithm. Path search in RG can be either minimal or non-minimal. For a 3 × 3 mesh, the maximum connectivity is 72 which means that each node can communicate with eight other nodes. Using this simple metric, all the deadlock free turn models were evaluated and turn models that do not provide full connectivity have been excluded. There are only 50 uniform turn models that are deadlock free and also provide full connectivity. These turn models are visualized in Fig. 2. There are 14 four-turn turn models, 24 five-turn turn models and 12 six-turn turn models. Table I lists the turn models which are previously named and addressed in the literature.

In order to classify turn models, we can use the Degree of Adaptiveness (DoA) introduced in [3] which only considers the shortest paths from a source node to destination node. A general form of DoA metric can be formulated as:

$$DoA = \frac{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} NoSP_{i,j,rg}}{\text{number of pairs of nodes}} \tag{2}$$

Where N is the number of nodes in the network and $NoSP_{i,j}$ is defined as:

$$NoSP_{i,j,RG} = \begin{cases} \text{number of shortest paths in } RG \text{ from} & i \neq j \\ node_{i,L,out} \text{ to } node_{j,L,in} & \\ 0 & \text{otherwise} \end{cases}$$

The resulting DoA for the turn models are presented in Table II. However, it is no surprise that turn models with higher number of turns, also have higher DoA. Extending this metric (DoA_{Ex}) to include all the simple paths in the network (paths that do not have repeating nodes in them) provides a slightly different picture than the original DoA. DoA_{Ex} makes it possible to classify the turn models even further. The DoA_{Ex} metric can be described as follows:

$$DoA_{Ex} = \frac{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} NoSP'_{i,j,RG}}{\text{number of pairs of nodes}} \tag{3}$$

Where N is the number of nodes in the network and $NoSP'_{i,j}$ is defined as:

$$NoSP'_{i,j,RG} = \begin{cases} \text{number of simple paths in } RG \text{ from} & i \neq j \\ node_{i,L,out} \text{ to } node_{j,L,in} & \\ 0 & \text{otherwise} \end{cases}$$

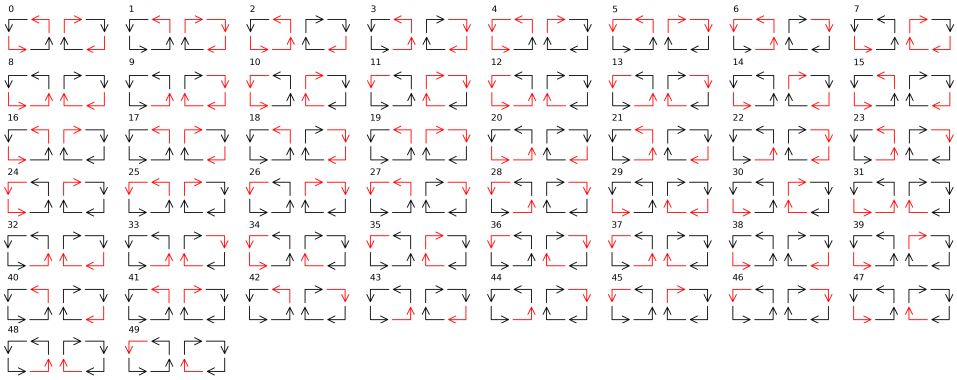


Fig. 2. Visualization of all deadlock free 2D turn models with full connectivity. The forbidden turns are drawn in red.

Table II presents DoA and DoA_{Ex} metrics for the turn models in Fig. 2. This table shows that inside each class of turn models (four, five, and six-turn turn models), there are sub-classes that have different characteristics. As an example, turn model no. 3 shares two turns with XY and two turns with YX which allows it to have non-minimal de-routes. Similarly, under non-minimal routing, turn model no. 1 and 2 have even further advantage in providing path diversity.

Similar investigation has been conducted for 3D routing algorithms, where there are 24 turns available. Out of over 16 million possible 3D turn models, over 95 thousand deadlock free turn models which provide full connectivity in the network are extracted. The full list of these turn models and their visualizations are accessible at [11].

IV. Experimental Results

In this section, all the 50 turn models extracted in the previous section, are compared in terms of robustness and latency.

A. Robustness Evaluation

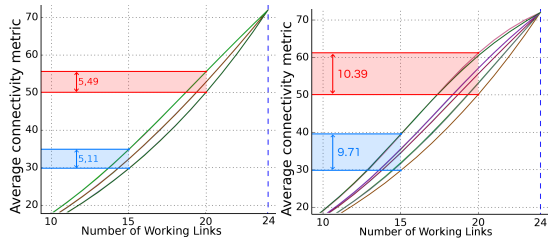
Robustness of a routing algorithm is defined here as the average connectivity metric of routing algorithm running on a 2D mesh NoC with specific number of faulty links. Experiments were conducted to evaluate the robustness of each routing algorithm. Using the connectivity metric introduced in section III, average connectivity of the network with n_{broken} permanently broken links (out of total number of n_{total} inter-router links), can be calculated using Algorithm 1. Where avg_{con} is a list of average connectivities and $avg_{con}[k]$ is average connectivity for a 3×3 network with k broken links.

For each turn model and each possible amount of broken links (n_{broken}) the connectivity metric is calculated and averaged over all possible configurations with n_{broken} links.

Fig. 3 illustrates the difference between average connectivity metrics for the turn models listed in Table II. Fig 3a shows the average connectivity metric of the turn models under minimal path routing for different number of working links. The three lines in the figure correspond to three classes of DoA in Table II, where turn models with higher DoA provide better connectivity. However, the gap between the curves is not substantial. Fig 3b depicts the average connectivity metric of turn models under non-minimal routing for different numbers of working links. In this case the curves diverge more, which corresponds to the seven classes, when considering DoA_{Ex}, and the gap between the curves is rather substantial where turn models with higher DoA_{Ex} provide better connectivity (Red and Blue marked regions in Fig 3).

Algorithm 1: average connectivity calculation algorithm

for $n_{broken} \in [0, n_{total}]$ **do**
  $list\_of\_configurations$ = list of all 3×3 2D mesh NoCs with n_{broken} broken links
  $sum_{con} = 0$
  forall $conf_{broken} \in list\_of\_configurations$ **do**
    Generate RG based on $conf_{broken}$
    sum_{con} += $Connectivity_{rg}$
  end
  $avg_{con}[n_{broken}] = sum_{con}$/len(list_of_configurations)
end
return avg_{con}



(a) (b)

Fig. 3. Comparison of avg. connectivity metric of turn models under a) minimal, b) non-minimal routing by number of available links.

B. Latency Evaluation

In this section, all the 50 deadlock free turn models shown in Fig. 2 are evaluated under synthetic traffic patterns using Noxim [12] NoC simulator. The experimentation setup parameters are as follows: A 4x4 2D mesh was considered. The system clock frequency is set to 1 GHz for all routers. As a synthetic traffic pattern, random uniform is considered in the simulations. Packets are generated using Poisson distribution.

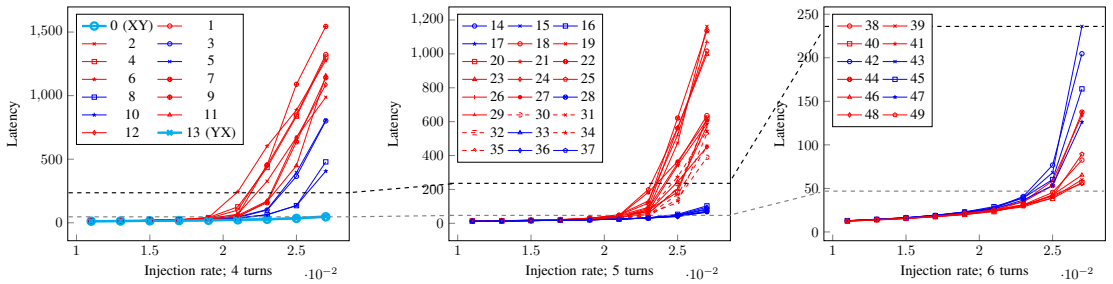


Fig. 4. Latency results under random uniform traffic for a) four, b) five, c) six-turn turn models.

The length of the packets is fixed and set to 8 flits, and FIFO depth of routers is 4 flits. For each simulation, the warm-up time is considered 1000 cycles in order to allow the transient effects to stabilize and subsequently, the simulation has been run up to 20000 cycles.

The average latency results are grouped based on the number of turns allowed, *i.e.* 4-turn, 5-turn and 6-turn turn models. Fig. 4a-c shows the average latency results for these turn models under random uniform traffic pattern (with packet injection rate ranging from 0.001 to 0.025). The curves are color coded in each figure to distinguish different classes of DoA_{Ex} (see Table II). The dotted lines in Fig. 4a-c indicate the corresponding highest value of the Fig. 4c and lowest value of Fig. 4a, since the range of axes is different between Fig. 4a-c. As it can be observed in Fig. 4a, two of the turn models (0 and 13 which are highlighted with thick cyan color), which correspond to XY and YX routing, outperform the other turn models in terms of average latency which conforms to the observations made in [13]. After those two turn models, both classes of 6-turn turn models and 5-turn turn models with lower DoA_{Ex} perform better than others. The performance (average latency and average throughput) and communication energy of all the 50 deadlock free turn models under different synthetic (Random uniform, Bit-reversal, Shuffle, Transpose and Butterfly) traffic patterns are available at [11].

V. Conclusion

In this paper, an in depth comparison of all 2D uniform turn models in terms of connectivity, adaptivity and robustness has been made. All possible 50 deadlock-free turn models that provide full connectivity have been extracted. An extended adaptivity metric have been introduced to classify the turn models even further. It became clear that one class of turn models with highest degree of adaptivity is much more robust under non-minimal path routing. The latency results shows that aside from XY and YX routing algorithms which outperform all other turn models under random uniform traffic pattern, 2 more classes fall very close to this class which are 5-turn turn models with lower adaptivity metric and 6-turn turn models with higher adaptivity metric.

Acknowledgments

The work has been supported by EU's H2020 RIA IMMORTAL, EU's Twinning Action TUTORIAL, Estonian Science Foundation grant ETF9429, Estonian institutional research grant IUT 19-1, Estonian IT Academy programme and funded by Excellence in IT in Estonia (EXCITE) project.

References

[1] S. Kumar, A. Jantsch, J. P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, 2002, pp. 105–112.

[2] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[3] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, 1992, pp. 278–287.

[4] Y. Miura, K. Shimozono, S. Watanabe, and K. Matoyama, "An Adaptive Routing of the 2-D Torus Network Based on Turn Model," in *Computing and Networking (CANDAR), 2013 First International Symposium on*, Dec 2013, pp. 587–591.

[5] S. Mubeen and S. Kumar, "Designing efficient source routing for mesh topology network on chip platforms," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, Sept 2010, pp. 181–188.

[6] A. Patooghy and H. Sarbazi-Azad, "Performance comparison of partially adaptive routing algorithms," in *20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, vol. 2, April 2006, pp. 5 pp.–.

[7] J. Duato, "A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 12, pp. 1320–1331, Dec. 1993. [Online]. Available: http://dx.doi.org/10.1109/71.250114

[8] S. P. Azad, B. Niazmand, P. Ellervee, J. Raik, G. Jervan, and T. Hollstein, "Socdep2: A framework for dependable task deployment on many-core systems under mixed-criticality constraints," in *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, June 2016, pp. 1–6.

[9] J. Duato, S. Yalamanchili, and N. Lionel, *Interconnection Networks: An Engineering Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[10] A. M. Shafiee, M. Montazeri, and M. Nikdast, "An innovational intermittent algorithm in networks-on-chip (noc)," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 2, no. 9, pp. 2907 – 2909, 2008. [Online]. Available: http://waset.org/Publications?p=21

[11] " Turn Model website ," http://turnmodel.pld.ttu.ee/, 2016.

[12] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 162–163.

[13] G. Ascia, V. Catania, M. Palesi, and D. Patti, "Neighbors-on-path: A new selection strategy for on-chip networks," in *2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, Oct 2006, pp. 79–84.

Appendix 4

IV
S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in Network-on-Chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 48–53, April 2017

From Online Fault Detection to Fault Management in Network-on-Chips: A Ground-Up Approach

Siavoosh Payandeh Azad*, Behrad Niazmand*, Karl Janson*, Nevin George*, Adeboye Stephen Oyeniran*, Tsotne Putkaradze*, Apneet Kaur*, Jaan Raik*, Gert Jervan*, Raimund Ubar*, Thomas Hollstein*†
Department of Computer Systems
*Tallinn University of Technology, †Frankfurt University of Applied Sciences
email: {siavoosh, bniazmand, karl.janson, nevin, steph, tsotne, akaur, jaan.raik, gert.jervan, raiub, thomas}@ati.ttu.ee

***Abstract*—Due to the ongoing miniaturization of silicon technology beyond the sub-micron domain and the trend of integrating ever more components on a single chip, the Network-on-Chip (NoC) paradigm has emerged to address the scalability and performance shortcomings of bus-based interconnects. As the feature size shrinks, the system gets much more susceptible to faults caused by wear-out and environmental effects. Thus, in order to increase the reliability, creates the need for having mechanisms embedded into such a system that could detect and manage the faults in run-time.**

In this paper, a ground-up approach from fault detection to fault management for such a NoC-based system on chip is proposed that utilizes both local fault management for fast reaction to faults and a global fault management mechanisms for triggering a large-scale reconfiguration of the NoC. Also, detailed description of strategies for fault detection, localization, classification and propagation to a global fault management unit are provided and methods for local fault management are elaborated.

***Keywords*—Fault Detection, Checkers, Fault Classification, Fault Localization, Fault management, Reconfiguration, Network-on-Chip.**

I. Introduction

Network-on-Chip (NoC) has emerged as a paradigm to address the scalability and performance shortcomings of traditional bus-based architectures [1], [2]. The trend of nano-scale electronics shrinking in size, makes them more susceptible to wear-out and environmental effects. This necessitates the detection and management of faults occurring at the run-time of the system, in order to provide higher reliability.

This work addresses a reliable NoC framework, which is maintained as an open-source project named Bonfire [3]. It provides support for fault detection and localization, local fault management, local fault classification, and fault information propagation to a global system health monitoring unit. In a NoC-based System-on-Chip, routers are responsible for transmitting data between the Processing Elements (PEs).

In Network-on-Chip, a router is composed of a data-path and a control part. The packets are transmitted via the data-path, while the control part directs the flow of data and the path the data should take when being transmitted between routers. Thus, both for the data-path and the control part, fault tolerance is of utmost importance for a reliable communication.

For the data-path, error detection and/or error correction techniques (such as single parity and Hamming encoding [4]) can be used. However, due to the area overhead of error correction techniques such as Hamming, the focus of this work is on single bit parity for the detection of faults in the data-path (inter-router links and data-path components of the routers).

On the other hand, faults in the control part of NoC routers should be handled. One way is to detect them via concurrent online checkers (for instance via the approaches proposed in [5], [6]) due to their low fault detection latency. There are also other methods such as Built-In-Self-Test (BIST) [7]. However, they interrupt the normal operation of the system for testing upon a fault occurrence. Thus, in the scope of this work, we focus on concurrent online detection of faults for the control part of routers. It is important to note that the checker outputs also facilitate fault localization [8], pinpointing the defective part in the circuit. Additionally, higher abstract deductions can be made based on them, such as existence of defect in turns in a router (a path from an input port to an output port). Such information can be used for reconfiguration of the routing algorithm or re-mapping of the tasks by units in charge of application mapping and scheduling. Works such as [9] have addressed multi-layer fault diagnosis and combining checkers at different levels of abstraction, however, they impose high latency. Furthermore, they have not addressed any mechanism for classification of faults and fault management, which are considered in our work.

In this work a ground-up approach from fault detection to management for NoC-based System-on-chips is proposed. Strategies for fault detection, localization, classification and propagation to a global fault management unit are described. Furthermore, in order to improve the reaction time to faults, methods for local fault management are elaborated.

The rest of this paper is organized as follows: in section II the basics of the Bonfire framework, including the NoC and router architecture are discussed. Section III describes the fault model used in this work and the method of fault injection on the links. In section IV different fault detection mechanisms for control and data path of the routers are discussed. Section V describes methods of fault localization. Sections VI describe the the process of fault classification and Section VII provides methods of handling faulty packets at the router level. Section VIII details fault information propagation to system health monitoring unit and finally, section X concludes the paper.

II. Bonfire framework

A. Bonfire NoC Architecture

The aim of the Bonfire project is to create a fault-tolerant framework for testing dependability mechanisms in a NoC-based System-on-Chip(SoC). The targeted NoC is using a 2D mesh topology where each tile of the network consists of a wormhole switching router equipped with fault tolerance

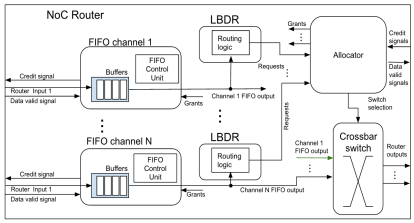


Fig. 1. Overview of the architecture of baseline credit-based flow control NoC router used in Bonfire network

mechanisms and a Processing Element (PE) connected to it via a Network Interface (NI). Each PE comprises a Plasma core [10], which is a 32-bit MIPS-I based open-source processor with three pipeline stages, along with 8 KB of RAM (as local memory). Details of the components of the framework are described in the following subsections. Bonfire is maintained as an open-source project, available at [3] .

B. Bonfire Router

The Bonfire network described in this paper utilizes 32 bit credit-based wormhole switching in the routers. Fig. 1 shows an overview of the baseline router used in the Bonfire network, without any fault-tolerance mechanism. The router comprises of an input buffer (implemented as First-In-First-Out (FIFO)), routing computation unit (implemented using Logic-Based Distributed Routing (LBDR) mechanism [11]), switch allocator (prioritizing multiple requests to the same output port based on Round-Robin policy) and crossbar switch.

We have opted for LBDR [11], since it is scalable compared to table-based routing in NoCs. Furthermore, LBDR describes the topology and the routing algorithm in a 2D NoC in terms of a fixed number of configuration bits, *i.e.* connectivity and routing bits. This makes it possible to use the connectivity bits for the indication of links in the 4 main directions as healthy or faulty, by setting the corresponding connectivity bit to zero (faulty) or one (healthy). Routing algorithm re-configuration (if necessary) can be done by changing the routing bits.

C. System Health Monitoring Unit (SHMU)

The Bonfire project targets a holistic system health monitoring and management solution. To implement this, a dedicated unit, called System Health Monitoring Unit (SHMU) [12], [13], is proposed which handles fault information collection and system-scale fault management and reconfiguration.

In Bonfire project SHMU runs as software on one of the Processing Elements (PEs) in the network. And if the processor fails, the SHMU tasks can be mapped on another node. Details about functionality and implementation of SHMU is beyond the scope of this paper.

III. Fault Model

In this work, we focus on single stuck-at fault model [14], which means in each router module only one fault can occur at a time. For data-path related modules, including the links, only one bit can get faulty at a time on the specific link. The same applies to the control part related modules. Thus, separate control part modules and data links from different ports can get faulty at the same time, but only one fault in each of them at a time. Transient faults are modeled as single stuck-at faults which last one clock cycle. Intermittent faults are modeled as bursts of transient faults in short periods. Permanent faults are modeled as a moving from transient fault to intermittent state and then finally with a permanent stuck-at fault.

In this work, fault injection is done using force command of ModelSim from Mentor Graphics [15]. The injection points are links between routers and also internal signals of the individual modules inside the router.

IV. Fault Detection

The Bonfire framework uses different methods for detection of faults in data-path and in the control part of the network.

A. Data-path Fault Detection

Since this work focuses on a single stuck-at-fault model, a simple parity checker module is used to cover all single-bit faults on the input ports of the router. Upon receiving a faulty flit, the router starts a fault classification process and also manages the fault locally in order to prevent network congestion (for more information, please refer to section VII).

B. Control Part Fault Detection

Concurrent Online checkers are utilized to detect faults in the control part of the NoC routers. A *checker* is a concurrent online fault detection module [5], [6]. It detects faults occurring at inputs and outputs of fan-out free regions [16] of the circuit with low latency. Since checkers provide fault information required for fault localization, this method is preferable to Double or Triple Modular Redundancy (DMR and TMR) schemes. The use of concurrent checkers for online fault detection in control part of NoC routers are described in more detail in [5], [6], [17], [18]. It is worth noting that the complete set of checkers for the control part of Bonfire NoC are available at [3], which covers the control part of FIFO, routing logic (LBDR) and allocator unit (allocator) shown in Fig. 1.

V. Fault Localization

As the number of checkers can grow very large (in the order of hundreds per router), it is not feasible to send the fault detection information from all these checkers to SHMU. Also, in case of a NoC router, for example, flipping of a bit in a register in one of the router's internal modules will not provide valuable information to the SHMU in the application layer. However, if the outputs of the checkers connected to this module are combined, it is possible to translate the output of the checkers into more meaningful abstracted information.

By combining the checkers for the control part of the router, it is possible to report faults at a more abstract level. For instance, in [8], a fault localization method is introduced which groups sets of checkers, making an assertion vector, facilitating finding fault location at different granularity levels in the control part of a NoC router. This can also be used when signaling higher levels in the architecture, such as the application level about the occurrence of faults.

Works such as [19], model faults in the control part as a complete node failure. In [20], illegal turns in the routers are detected, however, each router depends on the information

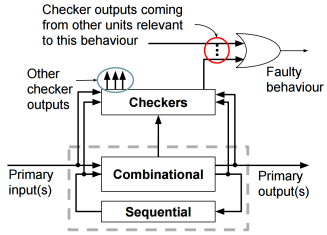


Fig. 2. General structure of the fault detection, grouping and classification mechanisms

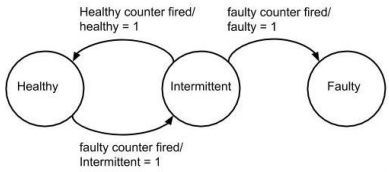


Fig. 3. Finite State Machine (FSM) for the fault classifier unit

from its neighbor routers for online fault detection. On the other hand, in our work, we combine checker outputs (as shown in Fig. 2) for the control part of a router. Further, this can be translated into detection of a *turn fault*. Unlike [20], we use the checker outputs in the current router to model turn faults, and there is no need for collecting information from neighbor routers. A turn fault is defined as a fault occurring in one of the components on the path from an input port to an output port of the router (*e.g.* a West to North turn fault or a straight path). This information can be passed to SHMU to the application layer. Later, if required, the SHMU can initiate re-configuration of the routing algorithm or re-mapping of the tasks on the nodes based on the fault information received from the lower (hardware) level.

VI. Fault Classification

With the growing number of transient faults, it would be impractical to send a separate notification to SHMU for each occurring fault. Not only would this impose additional latency by sending a notification from hardware to application layer, but it will also incur a significant power overhead.

To overcome this problem, faults are classified locally in the routers as *permanent*, *intermittent* or *transient*. The classified fault information is transmitted to SHMU if the fault is classified as intermittent or permanent. In [21], [22], an online fault classification mechanism is introduced as part of a cross-layer fault management framework, however, no details regarding the implementation of the fault classifier is provided. Whereas, in our work, a fault classification method based on [23], [24] is implemented; where a set of counters are used to count the healthy and faulty packets going through a network link. Each of the counters are compared with a threshold value. When a counter reaches its threshold, a signal is issued which is used by a control Finite State Machine (FSM) in charge of health making decision. Fig. 3 illustrates the FSM Diagram of the classifier unit. Every time the faulty packet counter

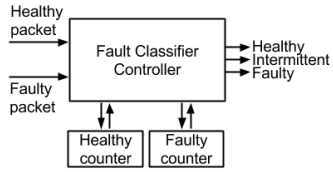


Fig. 4. Fault classifier block diagram

reaches its threshold, the FSM moves one step closer to the Faulty state. Every time the a counter reaches its threshold, both counters would be reset. It is noteworthy to mention that there could be different variations of state diagram models implemented for classification. The current state diagram as described in the Fig. 3 implements a scheme where there is no recycling of once faulty links. In contrast to [23], [24], since no error correction method has been used in this method, only two four-bit counters are utilized (see Fig. 4).

VII. Local Fault Management

Once a fault has been detected in the system, if it is classified as intermittent or permanent, the SHMU is notified. After obtaining the fault information, processing and making a decision, the SHMU can issue a command regarding that particular fault. But, during this time the effect of the fault has already propagated to other parts of the system and containment of the effects would be difficult if not impossible. So even though SHMU is responsible for fault management in the system, it can only manage the faults (in global scale) and some more detailed, distributed, mechanisms are needed for management of faults locally. This problem can be solved by implementing local fault management at each router. In this section two solutions for local management of the faults are provided.

A. Packet Dropping Mechanism

One of the important cases to be addressed is appearance of faulty flit at the input port of a router, where the following situations might happen:

- **Fault in the flit type:** in this case, it is usually not possible to identify the flit type and it (and also subsequent flits belonging to the same packet) cannot be routed. If this is not taken care of, eventually the input buffer (FIFO) of the router will get full, which can, in turn, leads to network congestion.
- **Fault in the payload data:** this type of fault does not have any effect on the network performance. However, since the packet data is corrupt, the fault will manifest itself in the application layer.
- **Fault in the destination address field:** the routing module might not be able to route the packet or the packet gets forwarded to a wrong destination. This might also result in network congestion if it is not properly taken care of.

One of the approaches to bypass the problem of having faulty flits is to use error-correction techniques, such as Hamming coding (single bit error correction, double bit error detection) for all flits. By comparing the overhead of these

TABLE I
AREA AND FLIT SIZE OVERHEAD COMPARISON OF SINGLE BIT PARITY AND HAMMING DECODER

Unit name	Unit area (μm^2)	Overhead in flit size (bits)
32-bit single bit parity checker	663	1
32-bit Hamming decoder	7050	8

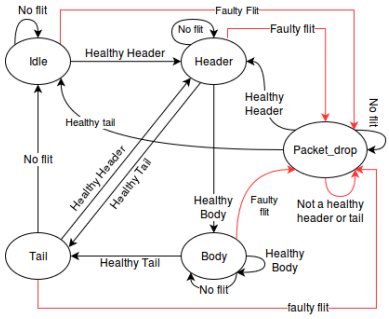


Fig. 5. Finite State Machine (FSM) diagram for the packet dropping mechanism

methods (shown in Table I) using AMS 0.18 μm CMOS technology library [25], it becomes clear that those methods impose additional area overhead to the correction circuit and also increase the flit (due to the additional bits needed for error correction). This, will affect the size of other network parameters, such as the physical link width which, in turn, also increases the size of the input buffer (FIFO) and crossbar switch.

In order to handle the above-mentioned situations, a *packet dropping* mechanism is incorporated in Bonfire framework. However, while using wormhole switching, in some cases, dropping the packet is not possible, for instance, when packet's header flit has already left the router. In such case, it is possible to cut the packet from the current position and attach a fake tail to it and forward it, while dropping the rest of the packet. This will not affect the network's operation. The results of such measures would manifest themselves in the application layer by comparing the packet length with the information in the header flit or as corrupt data. In our router architecture, the packet dropping mechanism is handled by a Finite State Machine (FSM), as is shown in Fig. 5. Additionally, the packet dropping mechanism has to generate fake credits for the upstream router in order not to interrupt the flow of the traffic.

In the Bonfire router, the packet dropping mechanism is improved even further by adding the flit saving functionality – a capability to detect position of the error in flits. In case the error is in the payload part, the packet will still be transmitted to its destination, thus making the application layer to handle the data errors. This will avoid re-transmissions in non-critical applications (for example many multimedia applications).

B. Routing Management

Once a link is classified as faulty, the router automatically sends this information to upstream router to update its LBDR connectivity bits (these bits can be over-written by SHMU later). If the change in connectivity bits happens when a packet is being processed, it might result in the packet being divided or mis-routed. In order to avoid this problem, the new connectivity bits should be stored in a register and routing module should wait until a new header flit arrives. The same approach is applied to routing bits of LBDR reconfiguration. The reconfiguration command is issued by the processing element at each node (once the reconfiguration message is issued by SHMU).

Another important point is to take care of faults occurring in the FIFOs which will be propagated to LBDR modules. In this case, to prevent congestion and network failure, the routing module (LBDR) should manipulate the FIFO modules in order to drop the packet. This is performed with a secondary and much simpler packet dropping mechanism, which generates fake grant signal to the FIFO when a faulty header is detected using a simple parity unit. Since LBDR is only sensitive to the header flit when making routing decisions, there is no need for support for cutting the packet and attaching fake tail. It is assumed that the dropped packets would be handled at the application layer.

VIII. FAULT INFORMATION PROPAGATION

The process of information transmission to the SHMU is also crucial. This can be done either (1) via reusing the existing network, or (2) by using an additional infrastructure working in parallel with the main NoC. There have been many proposals for fault information propagation to a global fault management unit. Some of the proposals, such as iJTAG [21], [22], [26], use scan chains. However, using an infrastructure like iJTAG requires single (or very limited) number of access points which limits the mapping possibilities for the SHMU on nodes since SHMU must have direct access to iJTAG access point. In addition, in approaches such as [21], [22], [26], the Instrument Manager (IM), which works as the iJTAG network controller and knows the structure of the instrumentation network, can become a single point of failure.

Some of the previous works in the literature have taken advantage of dual NoC architectures, such as [27]–[34]. In [34], in addition to the main network, a checker network is used (which is assumed to be 100% reliable) in order to deliver data to its destination in case of a fault occurrence in the main network. In [33], in parallel to the main NoC (which is used for transmitting the data), an additional control network is considered which is used for sending reconfiguration data for updating the connectivity and routing bits of LBDR in the network routers. The control network is used to inform a global manager node regarding faults occurring in different nodes. Despite the advantages these works might bring, they all incur additional area and power overhead. Moreover, if the area of the augmented circuitry for transmitting the fault information is not negligible, it can increase the probability of faults occurring in the additional network itself as well.

In this work, the classified fault information is propagated to SHMU via the NoC itself. The information would be bypassed to the Network Interface (NI). The NI will check the address of the SHMU and will pass the info to the node if SHMU is mapped on the same node (self diagnostic) or will generate

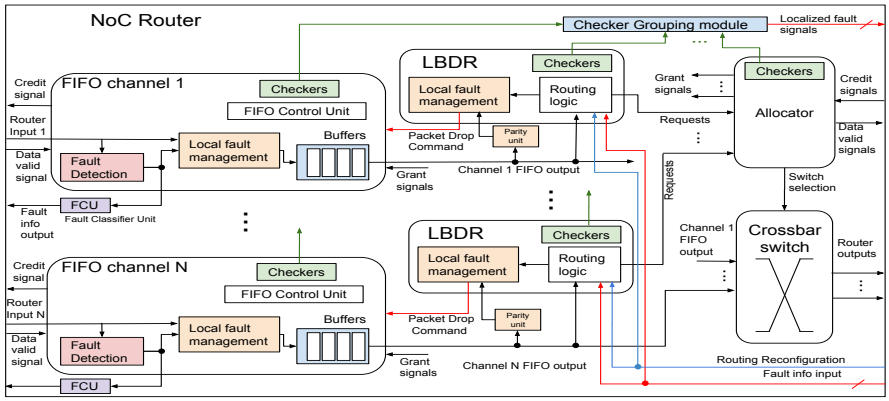


Fig. 6. Overview of the architecture of baseline credit-based equipped with fault tolerance mechanisms

TABLE II
AREA AND AVERAGE PACKET AND FLIT DROP FOR DIFFERENT PACKET DROPPING MECHANISMS.

Unit name	Unit area (μm^2)	Area overhead%	Average packet drop	Average flit drop
Original FIFO	14357	–	–	–
FIFO with packet dropping	16045	11.7%	≈1%	3.3%
FIFO with flit saving	16042	11.7%	≈1%	1.14%

TABLE III
AREA OVERHEAD RESULTS OF SELF-UPDATING LBDR OVER BASELINE LBDR

Unit name	Unit area (μm^2)	Increase in LBDR size	Increase in baseline router size
LBDR	1744	–	–
Self updating LBDR	2940	68.5%	5.9%

and send packets through the network to SHMU as soon as it finds idle time.

As mentioned in the previous section, after the local classification of the faults the information is sent to SHMU, which updates the system health map and can also trigger global re-configuration of the system in order to compensate for the faults. The reconfiguration packets will be sent to each node from SHMU and the node will send the reconfiguration information through the NI to the router. However, if the main NoC is used for transmission of the fault information and reconfiguration packets, under the running routing algorithm, the faults that should be reported, might also themselves prevent the messages to be correctly transmitted to the SHMU.

IX. RESULTS

Table II shows the area overhead of solutions for FIFO described in section VII (obtained using AMS 0.18 μm CMOS technology library [25] and synthesized via Synopsys Design Compiler [35]) along with the average flit and packet dropping ratio with random single stuck-at-fault injection on the network links with average rate of 5×10^6 faults per second. As it can be seen, when comparing the packet dropping approach to flit saving, the average full packet drop rate is not changing. This is due to the faults occurring in the header flit. However, the amount of flit drops is reduced by half, since the flits with the faulty payload will not be dropped in case of flit saving.

Table III shows the area overhead of the self updating LBDR unit. Both the area overhead of the self updating LBDR over the original LBDR (around 68%) and also its area overhead with respect to the baseline router without any fault tolerant mechanism (around 6%) are assessed.

By putting together all the mechanisms described in previous sections (fault detection, localization, classification and local management as shown in Fig. 6), the router grows 60.7% in area which is still lower than duplicate/triplicate-based methods such as DMR and TMR, while it also provides fault localization, management and system reconfiguration support at the same time. Moreover, the instantaneous detection of faults in the control part via the concurrent online checkers and combining them facilitates inferring more abstract and high level fault information (such as turn faults). Two main reasons for using such abstraction of the information are: (1) there is no advantage in transmitting very detailed fault information to the SHMU, since in order to make high-level decisions, SHMU has to abstract the information into turn faults. (2) Additionally, it reduces the amount of information to be transferred to SHMU through the NoC, thus, reducing the network latency and power consumption.

X. CONCLUSION

In this paper, a ground-up approach from fault detection to fault management for a Network-on-Chip based system is proposed. Concurrent online checkers are utilized to detect the faults in the control path and single parity check is used for the data-path. Fault localization and abstraction (into turn faults) are achieved by grouping information gathered from the control part checkers. Moreover, methods of local fault management at the hardware level using different packet dropping mechanisms are introduced and compared. To reduce the overhead of fault information propagation to application layer and its additional processing load, local fault classification mechanism is implemented which generates minimal, classified fault information for propagation.

Additionally, the necessity of having a relocatable System Health Monitoring Unit (SHMU) at the software layer is elaborated. SHMU utilizes the NoC itself for transmitting fault information after classification, thus avoiding a dual-NoC architecture and results in lower area overhead. The experimental results show the overall cost of applying such mechanisms, having 60.7% area overhead, which still makes it a better option than DMR/TMR-based approaches.

ACKNOWLEDGMENTS

The work has been supported by EU's FP7 STREP BASTION, H2020 RIA IMMORTAL, H2020 Twinning TUTORIAL, Estonian institutional research grant IUT 19-1, by the Estonian Center of Excellence in IT EXCITE funded by the European Regional Development Fund, and supported by Estonian IT Academy programme.

REFERENCES

[1] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 684–689.

[2] L. Benini and G. D. Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan 2002.

[3] "Project bonfire network-on-chip," https://github.com/Project-Bonfire, 2015.

[4] S. Ghosh, N. A. Touba, and S. Basu, "Synthesis of low power ced circuits based on parity codes," in *23rd IEEE VLSI Test Symposium (VTS'05)*, May 2005, pp. 315–320.

[5] 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 *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. IEEE Computer Society, 2012, pp. 60–71.

[6] P. Saltarelli, B. Niazmand, J. Raik, V. Govind, T. Hollstein, G. Jervan, and R. Hariharan, "A framework for combining concurrent checking and on-line embedded test for low-latency fault detection in noc routers," in *Proceedings of the 9th International Symposium on Networks-on-Chip*, ser. NOCS '15. New York, NY, USA: ACM, 2015, pp. 6:1–6:8.

[7] R. Sharma and K. K. Saluja, "An implementation and analysis of a concurrent built-in self-test technique," in *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, June 1988, pp. 164–169.

[8] K. Chrysanthou, P. Englezakis, A. Prodromou, A. Panteli, C. Nicopoulos, Y. Sazeides, and G. Dimitrakopoulos, "An online and real-time fault detection and localization mechanism for network-on-chip architectures," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, pp. 22:1–22:26, Jun. 2016.

[9] G. Schley, A. Dalirsani, M. Eggenberger, N. Hatami, H. J. Wunderlich, and M. Radetzki, "Multi-layer diagnosis for fault-tolerant networks-on-chip," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2016.

[10] Plasma CPU. http://plasmacpu.no-ip.org.

[11] J. Flich and J. Duato, "Logic-based distributed routing for nocs," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 13–16, Jan 2008.

[12] S. P. Azad, B. Niazmand, P. Ellervee, J. Raik, G. Jervan, and T. Hollstein, "Socdep2: A framework for dependable task deployment on many-core systems under mixed-criticality constraints," in *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, June 2016, pp. 1–6.

[13] S. P. Azad, B. Niazmand, J. Raik, G. Jervan, and T. Hollstein, "Holistic approach for fault-tolerant network-on-chip based many-core systems," *CoRR*, vol. abs/1601.07089, 2016. [Online]. Available: http://arxiv.org/abs/1601.07089

[14] A. Dalirsani, "Self-diagnosis in network-on-chips," PhD Thesis, Institut fr Technische Informatik der Universitt Stuttgart, July 2015.

[15] "ModelSim, Mentor Graphics," https://www.mentor.com/products/fv/modelsim/, 2017.

[16] R. Ubar, J. Raik, and H. Vierhaus, *Design and Test Technology for Dependable Systems-on-chip*, ser. Premier reference source. Information Science Reference, 2010. [Online]. Available: https://books.google.ee/books?id=_1zzPfTZND8C

[17] P. Saltarelli, B. Niazmand, R. Hariharan, J. Raik, G. Jervan, and T. Hollstein, "Automated minimization of concurrent online checkers for network-on-chips," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2015 10th International Symposium on*, June 2015, pp. 1–8.

[18] P. Saltarelli, B. Niazmand, J. Raik, R. Hariharan, G. Jervan, and T. Hollstein, "A framework for comprehensive automated evaluation of concurrent online checkers," in *Digital System Design (DSD), 2015 Euromicro Conference on*, Aug 2015, pp. 288–292.

[19] Y. Jojima and M. Fukushi, "A fault-tolerant routing method for 2d-mesh network-on-chips based on components of a router," in *2016 IEEE 5th Global Conference on Consumer Electronics*, Oct 2016, pp. 1–2.

[20] L. Huang, X. Zhang, M. Ebrahimi, and G. Li, "Tolerating transient illegal turn faults in nocs," *Microprocessors and Microsystems*, vol. 43, pp. 104 – 115, 2016, many-Core System-on-Chip Architectures and Applications (PDP 15). [Online]. Available: //www.sciencedirect.com/science/article/pii/S0141933116000284

[21] K. Shibin, S. Devadze, and A. Jutman, "On-line fault classification and handling in ieee1687 based fault management system for complex socs," in *2016 17th Latin-American Test Symposium (LATS)*, April 2016, pp. 69–74.

[22] A. Jutman, K. Shibin, and S. Devadze, "Reliable health monitoring and fault management infrastructure based on embedded instrumentation and ieee 1687," in *2016 IEEE AUTOTESTCON*, Sept 2016, pp. 1–10.

[23] J. Silveira, M. Bodin, J. M. Ferreira, A. C. Pinheiro, T. Webber, and C. Marcon, "A fault prediction module for a fault tolerant noc operation," in *Sixteenth International Symposium on Quality Electronic Design*, March 2015, pp. 284–288.

[24] J. Silveira, C. Marcon, P. Cortez, G. Barroso, J. a. M. Ferreira, and R. Mota, "Scenario preprocessing approach for the reconfiguration of fault-tolerant noc-based mpsocs," *Microprocess. Microsyst.*, vol. 40, no. C, pp. 137–153, Feb. 2016. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2015.08.005

[25] (2016) AMS 0.18um CMOS process. http://ams.com/eng/Products/Full-Service-Foundry/Process-Technology/CMOS/0.18-m-CMOS-process/.

[26] "Ieee approved draft standard for access and control of instrumentation embedded within a semiconductor device," *IEEE P1687/D1.71, March 2014*, pp. 1–347, Nov 2014.

[27] A. K. Abousamra, R. G. Melhem, and A. K. Jones, "Deja vu switching for multiplane nocs," in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, May 2012, pp. 11–18.

[28] R. Das, S. Narayanasamy, S. K. Satpathy, and R. G. Dreslinski, "Catnap: Energy proportional multiple network-on-chip," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 320–331. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485950

[29] A. Flores, J. L. Aragon, and M. E. Acacio, "Heterogeneous interconnects for energy-efficient message management in cmps," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 16–28, Jan 2010.

[30] S. Volos, C. Seiculescu, B. Grot, N. K. Pour, B. Falsafi, and G. D. Micheli, "Ccnoc: Specializing on-chip interconnects for energy efficiency in cache-coherent servers," in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, May 2012, pp. 67–74.

[31] N. E. Jerger, A. Kannan, Z. Li, and G. H. Loh, "Noc architectures for silicon interposer systems: Why pay for more wires when you can get them (from your interposer) for free?" in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 458–470.

[32] J. Balfour and W. J. Dally, "Design tradeoffs for tiled cmp on-chip networks," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 187–198. [Online]. Available: http://doi.acm.org/10.1145/1183401.1183430

[33] A. Strano, D. Bertozzi, F. Trivino, J. Sanchez, F. Alfaro, and J. Flich, "Osr-lite: Fast and deadlock-free noc reconfiguration framework," in *Embedded Computer Systems (SAMOS), 2012 International Conference on*, July 2012, pp. 86–95.

[34] R. Parikh and V. Bertacco, "Formally enhanced runtime verification to ensure noc functional correctness," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. ACM, 2011, pp. 410–419.

[35] (1994) Synopsys design compiler. http://www.synopsys.com.

Appendix 5

V

K. Janson, C. J. Treudler, T. Hollstein, J. Raik, M. Jenihhin, and G. Fey, "Software-level TMR approach for on-board data processing in space applications," in *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 147–152, April 2018

2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits and Systems

Software-Level TMR Approach for On-Board Data Processing in Space Applications

Karl Janson*, Carl Johann Treudler†, Thomas Hollstein*§, Jaan Raik*, Maksim Jenihhin*, Goerschwin Fey‡
*Tallinn University of Technology, †German Aerospace Center (DLR),
‡Hamburg University of Technology, §Frankfurt University of Applied Sciences
email: karl.janson@ttu.ee, carl.treudler@dlr.de, thomas.hollstein@ttu.ee,
jaan.raik@ttu.ee, maksim.jenihhin@ttu.ee, goerschwin.fey@tuhh.de

***Abstract*—Handling faults in computing systems is often expensive in terms of power, area and financial costs. In domains requiring high reliability in harsh environments, like the space domain, special highly reliable components are used, which may adversely impact the processing performance.**

In this paper, we propose the STROBES algorithm for fault handling in a multi-node embedded system which can be composed of standard commercial off-the-shelf components. In particular, it does not require underlying synchronization, but relies on embedded system's properties to derive bounds for communication and processing times. The algorithm can handle asynchronous behavior between the nodes up to user-defined bounds, in addition to a fault in the state or fail-stop failure of a single node. Theoretical analysis shows that this is sufficient for extended operating times. Experimental data show the efficient behavior of the STROBES algorithm for practical application with different state and time bounds.

***Keywords*-software level TMR; space applications; single event upsets; fault handling**

I. Introduction

Reliable computer systems in space need the ability to handle *Single Event Effects* (SEEs), particularly *Single Event Upsets* (SEUs) caused by radiation. The usual approach for testing such components includes up-screening and subjecting them to a simulated radiation environment while observing the effects [1]. For increased fault-tolerance, redundancy can be applied on hardware, software, time, and information[2]. Usually specially-built hardware, along with highly reliable (hi-rel) components is used in space applications to mitigate SEUs (e.g., by redundancy at lower levels [3]), while standard commercial off-the-shelf (COTS) components do not have such protections built in. However, the hi-rel components result in an increase in cost and they severely limit the speed and integration density of the components when compared to terrestrial applications [4]. In order to reduce recurrent development costs, the design could be made modular and scalable while being able to adopt varying dependability requirements [5]. However this does not solve all the problems hi-rel components introduce. This drives the use of commercial off-the-shelf (COTS) components in space applications.

In [6], a software-based Triple Modular Redundancy (TMR) approach for critical tasks is proposed for targeting multi-core Network-on-Chip (NoC) environments. However, fault-isolation and recovery are expected from the system it is run on. A repeated failure in a single core will result in the core being removed from the system. Additionally, [7] proposes a software redundancy based passive and active task replication for NoC based multi-core platforms. In the approach a duplicated version of critical tasks are either executed along with the original tasks or will be invoked in case of a failure in the processing element executing the original task. A repairing algorithm by remapping of faulty processing element's task on spare processing elements in a NoC-based System-on-Chip was proposed in [8] and [9]. However, those approaches assume running on a specific hardware. Additionally, they do not do not provide a real-time protection for application tasks. However, SHiFA [10] proposes dynamic mapping of applications to healthy processing elements in multi-core systems, but this is done only for small computational kernels which are dispatched and terminated. This approach is similar to the one in [11] which also considers anti-aging as an additional parameter during task distribution. All of the works listed above consider multi-core chips where losing a single core causes a system-wide failure.

The STROBES algorithm, proposed in this paper, is intended for reducing cost of space applications by allowing implementation of fault-tolerant systems using standard, unmodified, COTS components. However, since the proposed algorithm does not rely on special hardware, it is generic enough to also be used in other domains as well. Similar to the theoretical framework of [12], we make no assumption of underlying synchronization mechanisms, but only rely on worst-case execution and communication times, making our implementation more efficient.

The main highlights of the STROBES algorithm introduced in this paper are: *1)* software-level mitigation of hardware faults in a multi-node embedded system; *2)* low-cost fault detection using Cyclic Redundancy Checks (CRC) (i.e. no need for forward error correction); *3)* it relies on un-modified COTS components; *4)* detailed algorithmic description of the approach for fault mitigation is presented.

The paper is structured as follows: Section II introduces the system and fault model in more detail. Section III provides details about the STROBES algorithm. Section IV presents a theoretical analysis of STROBES' SEU handling capabilities. Section V provides empirical data on a simulation model and finally, Section VI concludes the paper.

II. System Assumptions and Fault Model

A. Processing System

The main assumption for using the STROBES method is that the application state (memory used by the application)

2473-2117/18/$31.00 ©2018 IEEE
DOI 10.1109/DDECS.2018.00033

on each node can be isolated. Applications which can be protected by STROBES need to include synchronization barriers for safe interruption for fault detection and correction, during which a checkpoint (CP) of the application state is taken and compared between the nodes (see Section III).

An additional assumption is that the processing elements are identical and connected to each other using a fully connected mesh network with broadcast support. However, the network does not need to guarantee deterministic arrival times for data.

B. Fault Model

In order to model various defects in the system, we have defined three fault classes, which our method has been designed to mitigate. The faults considered are the following:

1) $S = S_1, \ldots, S_n$ – *Corruption of the application state*: Fault S_i denotes that the application state of node P_i is corrupted. This can happen either due to radiation, faults in memory modules.
2) $J = J_1, \ldots, J_n$ – *Execution delays*: Fault J_i in node P_i means that at a time T the execution status of node P_i is different when compared to other nodes.
3) $N = N_1, \ldots, N_n$ – *Node stops processing*: Fault N_i denotes that node P_i is silent. This might be caused by the node P_i stopping sending out data or by a network problem. In a system with an active fault with type N_i only $n-1$ (where n is total number of nodes in the system) nodes can communicate with each other.

We denote the universe of all possible faults in the processing system as $U = S \cup N \cup J$.

Corruption of the checkpoint manifests as a Type-S fault. This fault can happen for different reasons (e.g. bit-flip in the application memory due to radiation, the application misbehaving and generating faults). However, the source of the fault is not important for fault handling using the STROBES algorithm.

Typical execution delays include shifting of start-up times of nodes, clock drift during operation, different application run times on nodes due to different loads, etc.

Since the STROBES algorithm has a special mode for initial synchronization (as shown in Section III), during initial startup of the system, there is no limit to the extent of execution delays J it can tolerate. However it cannot handle a stopped node N_i during this phase.

During runtime the STROBES algorithm can tolerate a single state corruption S_i or stopped node N_i at a time combined with any amount of execution delays J as long as the delay stays in the boundaries defined by timeouts in the algorithm (Section III). The timeouts are used for tolerating smaller variations in program execution and communication time caused by runtime differences happening during normal runtime, like the ones caused by network delays, different cache behaviors of the processors etc.

The two fault domains $S \cup N$ and J typically, in practice, have independent root causes. Type-J faults are present in any distributed system as components are never totally homogeneous. Type-S faults model SEUs and Type-N faults model more severe failures.

III. STROBES Algorithm for Fault Handling

The proposed software-level TMR approach for fault handling in on-board data processing in space applications (STROBES) can be summarized by the finite state machine visualized in Figure 1. It is designed for simplicity and makes use of a set of assumptions about the system and its behavior discussed earlier. Notably, different from the asynchronous approaches for fault handling [13], [14], timing-assumptions are made throughout the implementation.

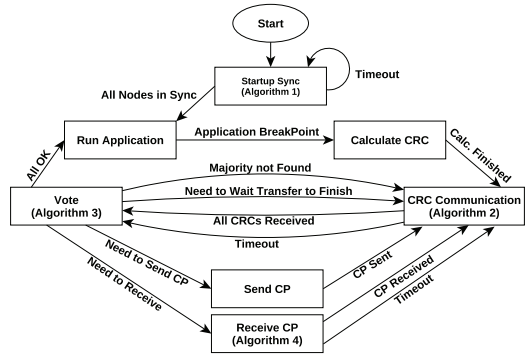


Figure 1. State diagram of the STROBES algorithm. Each node runs an instance of this algorithm

Each processing node $P_1, \ldots, P_n$ runs an instance of the STROBES algorithm. The system can handle fault types as described in the previous section. In this paper, we focus on the case study of TMR (i.e. $n = 3$), targeting data-processing system with three nodes. A larger number of simultaneous faults could be handled with more processing nodes ($n > 3$).

The STROBES algorithm works in cycles. The main steps of each cycle are as follows:

Algorithm 1 Initial synchronization

```
1:  typedef nodecrc = (cycle, crc)
2:  crcMap = map(node_id, nodecrc)
3:  while True do
4:      broadcast CRC
5:      while !CRC_timeout do
6:          if !buffer.empty() then
7:              (type,cycle,node_id,data)=buffer.pop()
8:              if type == crc then
9:                  crcMap[node_id]= (cycle, data)
10:         else if crcMap.size() == n then
11:             broadcast CRC
12:             next_state= Run Application
13:             return
```

First, an initial synchronization (*Startup Sync.*) is performed using Algorithm 1. This step is for mitigating synchronization issues when nodes are being initialized at slightly different times. Loosely, the broadcasts its dummy

CRC *e.g.*, all zeros (line 4) and then waits for dummy CRCs from 2 other nodes (while loop at line 5). The content of these is ignored as they are used only for synchronization. If all three CRCs have been received (line 10), it will send out an additional dummy CRC. This is needed for the node which was initialized the latest, since otherwise it does not receive any CRCs from other nodes. Afterwards, the system is ready for actual data processing.

Run Application in Figure 1 indicates parallel independent data processing by all nodes. The duration of this step is task specific. The running application is paused when it hits a breakpoint to calculate a checksum for the current checkpoint (CP) of the application's memory (*Calculate CRC*). The areas of the application's memory covered by the CP are application specific and not defined by the STROBES algorithm. The CP includes the application's state, and optionally executable code and other constant data. Any checkpoint corruption is assumed to result in an erroneous checksum. The types of checkpoint corruption which can be detected depends on the type of checksum used, which must be appropriate for the expected checkpoint corruption suffered by the system. We use a CRC code.

The CRC code is sent to all other processing nodes and the other node's checksums are received (*CRC Communication*). Once all checksums have been received or a timeout is met, voting on the checksums is performed seperatly in every node to identify failures (*Vote*). If no failure is detected, the application resumes (*Run application*). Otherwise, the faulty node receives the correct checkpoint (*Receive CP*). One non-faulty node sends the checkpoint (*Send CP*) and the other non-faulty node(s) wait for the checkpoint transfer to finish by going into *CRC Communication* state and waiting for CRCs from other nodes. Once the CP transmission is completed, also the sender and the receiver will go into *CRC Communication* state. This allows re-synchronization with the waiting node(s).

Algorithm 2 CRC communication

```
1: broadcast CRC
2: while (crcMap.size() < n and !CRC_timeout) do
3:     while !buffer.empty() do
4:         (type, cycle, node_id, data) = buffer.pop()
5:         if type == crc then
6:             crcMap[node_id]= (cycle, data)
7: next_state= Vote
8: return
```

Algorithm 2 details *CRC Communication* state. After broadcasting the CRC, the algorithm reads the receiving buffer (line 4) and stores any received CRC (line 6) until the receiving buffer is empty (line 3). This process continues until CRCs from all n nodes have been received or the timeout is reached (line 2). Afterwards, the STROBES algorithm proceeds to *Vote* state.

The details for *Vote* state under a single fault assumption are shown in Algorithm 3. Note, that under a single fault assumption at most one node which has an erroneous cycle

Algorithm 3 Voting at the processing node

```
1: if crcMap.size() == 0 then
2:     next_state= CRC Communication; return
3: cycle_maj = vote_cycle_number(crcMap)
4: if all cycle numbers are equal then
5:     next_state= CRC Communication; return
6: if current_node_cycle != cycle_maj then
7:     next_state= Recv. checkpoint; return
8: crc_maj = vote_crc(crcMap)
9: if all CRCs are different then
10:    next_state= CRC Communication; return
11: if all CRCs equal and crcMap.size() == n then
12:    next_state= Run application; return
13: if current_node_crc != crc_maj then
14:    next_state= Recv. checkpoint; return
15: min_addr = minimal address in correct_nodes
16: if current node's address == min_addr then
17:    next_state= Send checkpoint
18: else
19:    next_state= CRC Communication
20: return
```

number due to a synchronization failure or a faulty CRC checksum may enter the voting state. Algorithm 3 starts with voting on the current cycle number. A node with an erroneous cycle number will receive the correct application checkpoint (line 7). Next, checksums are considered. If all checksums are different, then the algorithm proceeds to the *CRC Communication* step (line 10). If all checksums are the same, the application is resumed (line 12) – note, that we assume that a node with faulty cycle ID will have a faulty CRC, too. Next, majority voting determines the correct checksum. A faulty node will wait for receiving the correct application checkpoint (line 14); one node with correct checksum will send the application checkpoint (line 17) and all other nodes with correct checksum will wait for this update to finish (line 19).

Algorithm 4 Receiving the correct checkpoint

```
1: CP = Pointer to application's checkpoint memory
2: while !buffer.empty() and !CP_recv_timeout do
3:     (type, cycle, data) = buffer.pop()
4:     if type == checkpoint then
5:         if cycle >= cycle_maj then      ▷ as determined in Algorithm 3
6:             *CP = data
7:             next_state= CRC Communication
8:             return
9: next_state= CRC communication
10: return
```

Finally, Algorithm 4 shows how the erroneous checkpoint is updated. The buffer for receiving data over the network is read until the correct checkpoint has been received timeout is met. Using a timeout is important if the time shift was too large for the faulty node to still receive the checkpoint from a correct node.

If one node permanently stops processing, *i.e.*, it suffers a fault of Type-N, the processing performance of the overall system degrades. All nodes will always wait first for the timeout to receive checksums and then wait for the timeout for the checkpoint transfer. However, overall correct operation will continue until at least two nodes are functional.

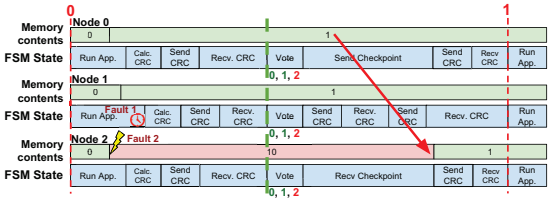


Figure 2. Fault handling of the STROBES algorithm

Figure 2 shows the behavior of the STROBES algorithm in presence of faults. The figure shows fixing of two types of faults: "Fault 1" in the figure represents an execution delay, where the application on Node 1 runs slightly longer when compared to other nodes (Type-J fault). This fault is handled by other nodes by waiting until they receive the CRC from Node 1. The green dashed line in the figure represents the time when the system is re-synchronized. "Fault 2" in the figure represents a faulty checkpoint for Node 2. During the voting phase each node independently understands, by the process of majority voting, that Node 2 is faulty. The fault is fixed by Node 0 (since it has the lowest address) sending its checkpoint to Node 2. Afterwards the system is re-synchronized by the nodes waiting for each other's CRCs.

The algorithm guarantees successful synchronization of the system under the proposed fault model if the chosen time-out value CRC_timeout is at least twice less than the time needed for the transmission of the checkpoint $t_{transfCP}$, and the CRC_timeout is larger than the maximum variation of *Run App* phases between different nodes Δt_{RunApp}, taking also into account the worst case communication delays between the nodes MAX(t_{comm}), as illustrated by Equation 1:

$$t_{transfCP}/2 > CRC_{timeout} > \Delta t_{RunApp} + MAX(t_{comm}) \quad (1)$$

IV. SEU Correction Capability

This section provides theoretical information on the STROBES algorithm's capability of handling bit flips in the application state, caused by a transient fault (e.g. an SEU). Since the STROBES algorithm is designed to protect mostly against SEUs in the application memory storing the checkpoint, the model introduced in this section does not include faults in other parts of the system.

A. Detection and Recovery Performance Prediction

This subsection compares two simple systems: a single-node system without any fault-tolerance measures, and the STROBES algorithm described in Section III running on three identical nodes. See Table I for the used symbols and parameters.

1) Single Node: For the single-node system the expected number of SEUs (λ_{SEU}) during process execution time can be calculated using Equation 2

$$\lambda_{SEU} = BER \cdot s_s \cdot t_p \quad (2)$$

The bit error rate (BER) in Equation 2 is a value specific to the sensitivity of the memory component and ion spectrum of the environment the system operates in.

Table I: Symbol Definitions

Symbol	Explanation	Symbol	Explanation
BER	Bit error rate	t_p	Process execution time, in seconds
BW	Bandwidth between two nodes, in $^{bit}/_s$	t_{pc}	Time for processing and checking
r_c	CRC calculation rate, in $^{bit}/_s$	t_t	Duration of the checkpoint transfer
s_s	Checkpoint size in bits	λ_{SEU}	Average SEU count during t_p
s_o	Size of communication overhead, in bits	λ_{pc}	Average SEU count during t_{pc}
c	Reduction factor of the CRC	λ_t	Average SEU count during t_t
o	Overhead factor for transfer	$P_\lambda(k)$	Poisson distribution, $\frac{\lambda^k}{k!}e^{-\lambda}$

2) System Using STROBES: For a STROBES system, additional steps are necessary in order to model the SEU rate. Also, conditional execution of checkpoint transfer leads to a more complex model, such as shown in Equation 3:

$$\lambda_{pc} = BER \cdot s_s \cdot \left(t_p + \frac{s_s}{r_c} + \frac{s_s \cdot c + s_o}{BW}\right) \quad (3)$$

The first terms (BER and s_s) of Equations 2 and 3 are identical. However, the time required to complete a cycle is different. This is due to the fact that the STROBES approach has some additional timing overhead due to the need for calculation and transfer of the CRC (it is assumed that all nodes exchange their CRCs in parallel).

The case in which a mismatch is detected while comparing the CRCs, and the checkpoint needs to be transfered from a correct node to the faulty node is modeled by Equation 4.

$$\lambda_t = BER \cdot s_s \cdot \left(\frac{(1+o)s_s + s_o}{BW}\right) \quad (4)$$

Duration of the CP transfer period (right term in Equation 4) is determined by the time it takes to transfer the data from the correct node to the faulty node. Error rates for both phases contain a term which is proportional to ${s_s}^2$. For parameters chosen similar to those in example shown in Subsection IV-B, size of the checkpoint influences the error rate the most, for any given BER.

$$p_{x,ok} = P_o(\lambda_x) \quad (5)$$
$$p_{x,err} = 1 - P_o(\lambda_x) \quad (6)$$
$$p_{sys,ok} = {p_{pc,ok}}^3 + 3 \cdot ({p_{pc,ok}}^2 \cdot p_{pc,err}) \cdot {p_{t,ok}}^3 \quad (7)$$

The probability of the system completing one cycle of the STROBES algorithm (Equation 7) is the cumulative probability that all three nodes are error-free (${p_{pc,ok}}^3$) and the three possible cases in which one error occurs ($3 \cdot ({p_{pc,ok}}^2 \cdot p_{pc,err})$) and is followed by an error-free transfer phase (${p_{t,ok}}^3$).

B. Example

The example below illustrates the STROBES' SEU handling capability in regard to CP size. It represents a small embedded system consisting of processing nodes connected together with in a full-mesh network with the following performance parameters: $t_p = 200ms$, $s_s = 0.5 - 256MiB$, $BW = 39^{Mb}/_s$, $BER = 10^{-7}$, $r_c = 80^{Mb}/_s$.

Figure 3a shows the expected Mean Time To Failure (MTTF) dependence on the checkpoint size for a STROBES system and a single, unprotected node with the same parameters. To illustrate the limits of the approach, only larger checkpoint sizes are shown in the figure. For a system with size of 0.5 MiB an increase in MTTF of over $118 \cdot 10^3$ can

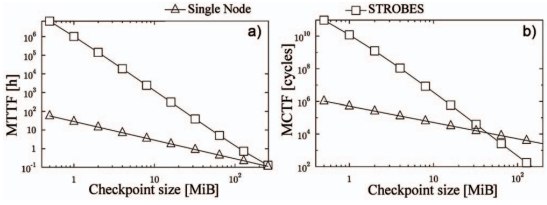


Figure 3. a) MTTF over CP size; b) MCTF over CP Size

be seen, where as for 128 MiB it is only about three times. Additionally, as size of the CP grows, there is also a higher chance of an SEU happening.

As the CP size increases, the system spends more time calculating the CRCs, and additionally, it takes also more time to transmit the checkpoint in case of a fault. With an assumption of only single SEU during a cycle, MTTF can be normalized to an average length of the cycle, which introduces another metric that fits better for STROBES – Mean Cycles To Failure (MCTF). The dependence of MCTF on MTTF can be seen in Figure 3b.

V. Experimental Results

A. Simulation Tool

For testing the feasibility of the STROBES algorithm, its behavior was simulated using a custom high-level event-based simulation tool. The simulator runs an implementation of the proposed method concurrently on three simulated nodes. In order to profile the algorithm in an application-independent setting, the application is replaced by a time delay, combined with a fault status information of its memory state. Due to the abstract nature, simulation measures time in discrete Time Units (TU).

The tool simulates the nodes running the STROBES algorithm in a fully connected mesh network with buffered interfaces and broadcast support. The network is not simulated down to protocol level, however a network delay and maximum bandwidth is taken into account while transferring data, calculation of which is based on Equation 8 using values provided in Subsection IV-B.

$$T_t = \frac{s_s + s_o}{BW} \tag{8}$$

B. Results

To evaluate the fault handling efficiency of the STROBES algorithm, it was simulated under different fault conditions for different CRC timeout values and the worst case fault recovery time was recorded.

In total, four types of experiments were performed: *1)* one or two nodes being late during initial synchronization; *2)* one or two nodes being late during normal runtime (differences caused by network delays, different CPU load on nodes, etc.); *3)* one or two nodes being late during runtime, with an additional faulty checkpoint in one of the nodes. Moreover, separate experiments were run for the late nodes and for the synchronized nodes having the faulty checkpoint; *4)* re-synchronization of a node previously silent node (Type-N fault).

Below are the results of fault recovery experiments which were run. Each set of the experiments (except the re-integration of the previously silent node) were run for both one and two late nodes:

a) Profiling of initial synchronization – The effects of execution delays (Type-J fault) caused by different booting times of nodes.

b) Run-time synchronization – Effects of execution delays (Type-J fault) caused by the characteristics of the distributed nature of the STROBES algorithm (network delays, uneven processor load on nodes, etc.).

c) Run-time execution delays with a faulty checkpoint – In addition to execution delays, the CP of one node is faulty (Type-J AND Type-S). The experiment was repeated for both the faults being in a late node or in a node which was synchronized with another node at the point of the CP fault happening.

d) Re-integration of a previously silent node – Initially one of the nodes is silent (Type-N fault). After some delay, the node comes back "alive" (e.g. node is rebooted by a watchdog). However, there is still a Type-J fault, together with Type-S fault since the node will be multiple cycles late. Additionally, two of the nodes will be running normally, while the formerly silent node will be in initial synchronization mode. The experiment measures the time required for the silent node to become synchronized into the system from the point it resumed communication with the rest of the nodes.

All experiments on runtime execution delays were run for both cases where one node was late (e.g. J_0 AND $!J_1$ AND $!J_2$) and cases where two nodes were late (e.g. J_0 AND J_1 AND $!J_2$). The second case covers also the situation where all nodes are out of synchronization (most realistic case).

Each experiment consists of a range of sub-experiments (in the order of thousands). During the experiments, a range of CRC timeout values (according to Equation 1) were tested with full range of possible execution delay values (that is, only cases where the inaccuracies were smaller than the CRC timeout). The worst case result of all sub-experiments was recorded as the final result of the an experiment.

All experiments were repeated for different parameter sets (representing different system configurations) as shown in Table II. The parameter sets were chosen so that there would be both, a smaller and a larger value, represented for application runtime and for maximum CP size. Additionally, it was intended that both cases, where CP transmission time is longer than application runtime, and also, where it is shorter, would be represented. Maximum value of CP size was selected to be 10 MiB, since according to results presented in Section IV, this value is near the upper end, where the STROBES algorithm still behaves better in terms of tolerance against SEUs, than the single-node algorithm.

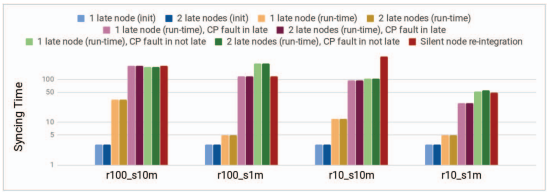


Figure 4. Fault handling of STROBES algorithm for different fault classes.

Figure 4 shows the worst case synchronization time for different fault scenarios across multiple experiment sets. Since initial synchronization does not depend on the protected application, it stays constant over all experiment sets. Additionally, according to Figure 4, the differences in worst case synchronization times in presence of a faulty CP together with run-time execution delays do not depend on the number of late nodes.

However, there seems to be a difference in worst case synchronization in time based on the faulty node and the CP size – in case of smaller (1 MiB) CP there is a noticeable difference if the fault is in the late node or not (in case the fault is in the late node, the system fixes the fault much faster). However, there is not much difference in case of larger, 10 MiB checkpoint size.

The worst case synchronization time for re-integration of a silent node is generally faster for smaller application runtimes and CP sizes. On average it is equal to, or even smaller than the time required to fix execution delays in presence of a faulty checkpoint. This is due to the fact that such a node is, according to the simulation model, behind the rest of the nodes by multiple application cycles and thus out of synchronization with other nodes and has a checkpoint which is different than the nodes.

The peak in the worst case time of re-integration of a silent node in case of application runtime of 10 TU and CP size of 10 MiB can be explained by the the fact that there exists a set of parameters (value of CRC timeout), where it takes for a silent node multiple synchronization tries to succeed. Since the maximum CRC waiting delay is adjustable in a relatively large range according to Equation 1, it is possible to find a more optimal set of parameters by proper modeling and simulation of the behavior of the STROBES algorithm before deployment.

VI. Conclusion

We proposed a software-level approach for handling synchronization problems together with a corrupted state or even full failure of one processing node. The proposed STROBES approach puts no further requirements on the processing hardware besides being able to derive upper bounds for communication times and processing times, which is standard for such systems. A large application state is efficiently handled by resorting to error detection instead of forward error correction. Theoretical analysis and empirical results show the capabilities of our algorithm for fault handling.

Table II: Experiment Sets

Notation	Application runtime (in TU)	CP size (in MiB)	CP transfer time (in TU)
r100_s10m	100	10	65
r100_s1m	100	1	7
r10_s10m	10	10	65
r10_s1m	10	1	7

While most of our discussions referred to triplication, the algorithm itself is not restricted. In future work we will study how to extend the scope to more processing nodes to handle more than a single state corruption or fail stop of a node.

Acnowledgements

This work has been supported by H2020 RIA IMMORTAL, H2020 Twinning TUTORIAL, Estonian institutional research grant IUT 19-1, the Estonian Center of Excellence in IT EXCITE, the European Regional Development Fund, and Estonian IT Academy programme.

References

[1] European Cooperation for Space Standardization, *ECSS-Q-ST-60-13C- Space product assurance- Commercial electrical, electronic and electromechanical (EEE) components*. ESA Requirements and Standards Division, 2013.

[2] E. Dubrova, *Fault-Tolerant Design*, 1st ed. 978-1-4614-2113-9: Springer-Verlag New York, 2013.

[3] European Cooperation for Space Standardization, *ECSS-Q-HB-60-02A – Techniques for radiation effects mitigation in ASICs and FPGAs handbook*. ESA Requirements and Standards Division, 2016.

[4] S. Habinc, A. Sakthivel, J. Ekergarn, A. Bjorkengren, R. Pender, S. Landstrom, F. Cordero, J. Mendes, T.-M. Ho, and K. Stohlmann, "Mascot on-board computer based on GR712RC," in *Data Systems in Aerospace (DASIA)*, 2013.

[5] C. J. Treudler, J.-C. Schröder, F. Greif, K. Stohlmann, G. Aydos, and G. Fey, "Scalability of a base level design for an on-board-computer for scientific missions," in *Data Systems In Aerospace (DASIA)*, 2014.

[6] J. W. Alexander, B. J. Clement, K. P. Gostelow, and J. Y. Lai, "Fault mitigation schemes for future spaceflight multicore processors," in *Design, Automation and Test in Europe (DATE)*, 2012, pp. 358–367.

[7] N. Chatterjee, S. Paul, and S. Chattopadhyay, "Fault-tolerant dynamic task mapping and scheduling for network-on-chip-based multicore platform," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 4, pp. 108:1–108:24, May 2017. [Online]. Available: http://doi.acm.org/10.1145/3055512

[8] C. L. Chen, Y. H. Chen, and T. Hwang, "Communication driven remapping of processing element (PE) in fault-tolerant NoC-based MPSoCs," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 666–671.

[9] V. Fochi, L. L. Caimi, M. Ruaro, E. Wchter, and F. G. Moraes, "System management recovery protocol for MPSoCs," in *IEEE International System-on-Chip Conference (SOCC)*, 2017, pp. 367–374.

[10] M. Fattah, M. Palesi, P. Liljeberg, J. Plosila, and H. Tenhunen, "SHiFA: System-level hierarchy in run-time fault-aware management of many-core systems," in *Design Automation Conf.*, 2014, pp. 1–6.

[11] C. Bolchini, A. Miele, and D. Sciuto, "An adaptive approach for online fault management in many-core architectures," in *Design, Automation and Test in Europe (DATE)*, 2012, pp. 1429–1432.

[12] J. F. Hermant and G. L. Lann, "Fast asynchronous uniform consensus in real-time distributed systems," *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 931–944, 2002.

[13] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[14] C. Honvault, M. Le Roy, P. Gula, J. C. Fabre, G. Le Lann, and E. Bornschlegl, "Novel generic middleware building blocks for dependable modular avionics systems," in *European Dependable Computing Conference*. Springer, 2005, pp. 140–153.

Curriculum Vitae

1 Personal data

Name	Karl Janson
Date and place of birth	8 October 1988 Pärnu, Estonia
Nationality	Estonian

2 Contact information

Address	ICT–509, Akadeemia Tee 15A, 12618, Tallinn, Estonia
Phone	+372 620 2267
E-mail	karl.janson@taltech.ee

3 Education

2015 – ...	Tallinn University of Technology	PhD studies
2013 – 2015	Tallinn University of Technology	MSc
2008 – 2013	Tallinn University of Technology	BSc

4 Language competence

Estonian	native
English	fluent

5 Professional employment

2015 – ...	Tallinn University of Technology, Early Stage Researcher

Elulookirjeldus

1 Isikuandmed

Nimi	Karl Janson
Sünniaeg ja -koht	8. oktoober 1988 Pärnu, Eesti
Kodakondsus	Estonian

2 Kontaktandmed

Aadress	ICT–509, Akateemia Tee 15A, 12618, Tallinn, Eesti
Telefon	+372 620 2267
E-post	karl.janson@taltech.ee

3 Haridus

2015 – . . .	Tallinna Tehnikaülikool	Doktoriõpe
2013 – 2015	Tallinna Tehnikaülikool	MSc
2008 – 2013	Tallinna Tehnikaülikool	BSc

4 Keelteoskus

eesti keel	emakeel
inglise keel	kõrgtase

5 Teenistuskäik

2015 – . . .	Tallinna Tehnikaülikool, nooremteadur

ISSN 2585-6901 (PDF)
ISBN 978-9949-83-654-3 (PDF)