MASTER'S THESIS

Evaluation of Fault Tolerance in Networks on Chip

Fault Injection and Performance Analysis between different Fault Protection Schemes

submitted at the Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Science)

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Dr. Junshi Wang
Dr. Nima Taherinejad

at

Institute of Computer Technology (E384)
Vienna University of Technology

by

Lukas Temmel, B.Sc.
Matr.Nr. 1328331
Templergasse 28, 2340 Mödling, Austria

10.02.2019 _____

Abstract

Network on Chip is a communication system used in integrated circuits as an alternative to bus systems. Instead of all chip elements exchanging their data directly or via a bus, packets containing the information are sent over different topologies between source and destination, which can be for example a storage element or a processing unit. Fault tolerance of such systems is of critical importance due to its role as communication backbone. Additionally, the increasing number of elements on a chip and its complexity introduce more possibilities for faults in the circuits. A common way to improve the fault tolerance is to implement additional mechanisms like error correction code, adaptive routing algorithms, or re-transmission of packets.
However, how do these measures perform if they themselves become faulty? Furthermore, how does this affect the performance of the Network on Chip? Here it is shown that the impact of faults on fault tolerance in Network on Chips needs to be evaluated during the design process.
The network was evaluated by adding fault modules into the source code and triggering them to represent certain fault patterns. The results show that the fault tolerance measurements affect the reliability differently for certain faults and partly let the network perform worse than without any tolerance measurements activated. This emphasizes the importance of this evaluation and that it should be considered and applied for designing such systems.
This work is seen as a starting point to investigate if the fault injection process can be used to improve the sturdiness of such systems. Furthermore, it is seen as a convenient way, also for other research groups, to evaluate the behavior of a system under the effects of faults. The fault injection methods are not bound to this project and can be used for other Verilog Code based systems as well.

Kurzfassung

Network on Chip ist neben klassischen Bussystemen ein alternatives Kommunikationssystem für integrierte Schaltungen. Anstatt dass alle Chip-Elemente ihre Daten direkt oder über einen Bus austauschen, werden Pakete mit den Informationen über verschiedene Netzwerktopologien versendet. Fehlertoleranz ist für solche Systeme von besonderer Relevanz, da sie das zentrale Kommunikationselement schützt und es durch die steigende Dichte an Transistoren und deren Komplexität mehr Möglichkeiten für fehlerhafte Elemente gibt. Ein gängiger Weg um die Fehlertoleranz zu erhöhen ist zusätzliche Mechanismen zu installieren wie Error Correction Codes, adaptive Routing Algorithmen oder das erneute Senden von Paketen.
Jedoch stellt sich die Frage wie sehr das System in Mitleidenschaft gerissen wird, wenn auch diese Maßnahmen fehlerhaft werden? In dieser Arbeit wird gezeigt wie wichtig die Wirkung von Fehlern auf Fehlertoleranzmechanismen in Network on Chip und die Evaluierung dieser während des Designprozesses ist. Ein Testnetzwerk wird durch das Einfügen von Fehlermodulen evaluiert, welche nach bestimmten Fehlermustern Störungen erzeugen. Es zeigt sich, dass die Fehlertoleranzmechanismen auf unterschiedliche Art die Zuverlässigkeit des Systems beeinflussen und in manchen Kombinationen sogar schlechter als ein System ohne jegliche Fehlertoleranzmechanismen arbeiten.
Diese Arbeit wird als Ausgangspunkt gesehen, um zu Evaluieren ob das Einfügen von Fehlermodulen es erlaubt die Robustheit von solchen Systemen zu verbessern. Weiters ist der vorgestellte Prozess der Fehlerinjizierung nicht auf dieses Projekt limitiert und daher eine gute Methode andere Systeme unter dem Effekt von Fehlern zu testen, wenn diese auf Verilog Gate Level Code synthetisiert werden können.

Table of Contents

Acronyms

ACK Acknowledgement.

ALU Arithmetic Logic Unit.

ASIC Application Specific Integrated Circuit.

AST Abstract Syntax Tree.

BISD Built-In Self-Diagnosis.

BIST Built-In Self-Test.

CAD Computer Aided Design.

DUT Device Under Test.

E2E End to end.

ECC Error Correction Code.

FEC Forward Error Correction.

FIS Fault Injection Signal.

FIU Fault Injection Unit.

FLIT FLow control unIT.

FPGA Field Programmable Gate Array.

GUI Graphical User Interface.

HBH Hop By Hop.

HDL Hardware Description Language.

IC Integrated Circuit.

IDE Integrated Development Environment.

IP Intellectual Property.

ITRS International Technology Roadmap for Semiconductors.

LT Link Transaction.

LUT Look Up Table.

MMU Memory Management Unit.

MPSoC Multi Processor System on Chip.

NI Network Interface.

NoC Network on Chip.

OSI Open Systems Interconnection.

PCB Printed Circuit Board.

PE Processing Element.

RAM Random Access Memory.

RC Routing Calculation.

RTL Register Transfer Level.

SA Switch Allocation.

SEU Single Event Upset.

SoC System on Chip.

SPICE Simulation Program with Integrated Circuit Emphasis.

SRAM Static Random Access Memory.

ST Switch Transaction.

TMR Triple Module Redundancy.

VA Virtual Channel Allocation.

VC Virtual Channel.

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language.

VM Virtual Machine.

1 Introduction and Motivation

In this section we start with outlining the development of System on Chips (SoCs) and Multi Processor System on Chips (MPSoCs), which led to an increased interest in Network on Chips (NoCs) over commonly used bus systems. Advantages and disadvantages will be pointed out before focusing on faults and their critical effect on such a system, followed by tolerance methods to counter them. At the end of this chapter, the further structure and tasks of this thesis will be laid out.

1.1 Introduction to Network on Chip

Processors have become an integral part of our world, and their influence in different application areas is still increasing[1]. Their development is continually evolving to improve different aspects of the chip like the number of I/O ports, lower power consumption or a higher frequency with which the processor could work.

A pivotal improvement which was beneficial to all these scopes was to reduce the size of the core elements of the processor, transistors (cf. Figure 1.1 [2]) and its connecting wires. The scientific progress can be followed in the documentation of the International Technology Roadmap for Semiconductors (ITRS). This allowed to reach higher densities of transistors on a wafer, which increased the yield of chips per wafer by reducing the space a micro-architectural feature (e.g. Arithmetic Logic Unit (ALU) or Memory Management Unit (MMU)) needs.

This allows that functionality which was before placed separately from the processor on the circuit board (e.g. interface logic, MMUs) can now all be integrated into one die, a so-called SoC. Benefits are time-saving development process, high specialization for the chips task, power saving, shorter data path within the chip components and due to the possibility of buying Intellectual Propertys (IPs) from different vendors a certain form of independence [NND09, p.2].

Furthermore integrating several processor cores into one die is used for performance improvements for so-called MPSoCs [WJ08] because linking several less complex cores is preferred over further expanding a single core due to its complexity diminishing the benefits [NND09, p. 1-2].

[1] E.g. Intel alone sold more than 10 million chips for media tablets in 2015 [1]

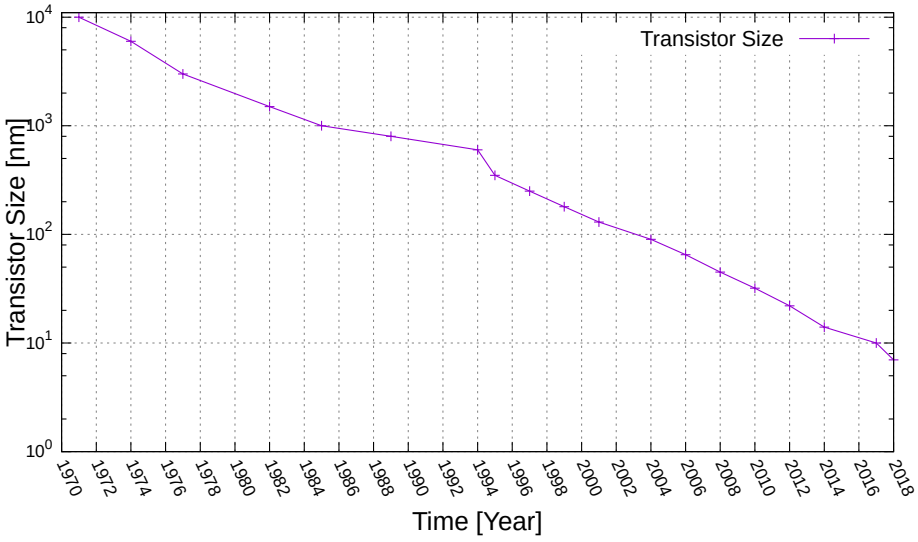


Figure 1.1: Shrinking transistor size from 1971 until 2018.

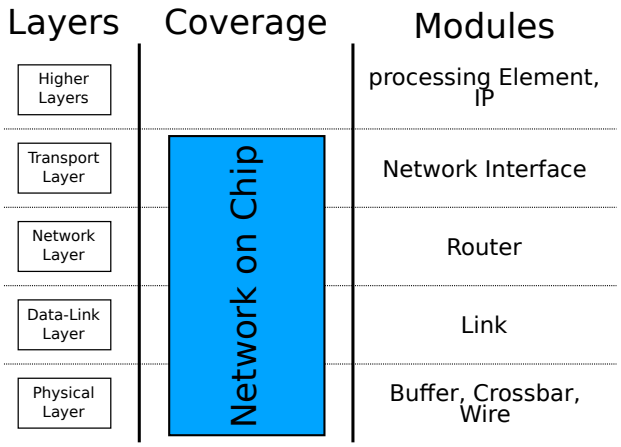


Figure 1.2: Open Systems Interconnection (OSI) layers covered by the NoC and the resembling modules (cf. [RFZJ13, Figured 1]).

Both techniques provide a high density of computation resources so that the limiting part of the chip shifts towards the on-chip communication structure between these elements. Traditional bus systems are not able to unlock the full potential of such a system [KPN+05, RFZJ13].
For instance, their structure allows only one element as a master for communication (cf. Figure 1.3 left side), which is counterproductive for a system which excels by its number of elements. Therefore, a form of parallel communication is needed. Additionally, every added core or IP extends the global wire length and increases by that the latency of the signals which aggravates if the chip size further grows or if the clock frequency increases [NND09, p.2ff]. As a result, the bandwidth will not scale with an increasing number of connected elements. Furthermore, a bus has no form of redundancy in its base concept. If it fails the entire system will halt, making it to a single point of failure for the communication of the chip [Ebr13].

A solution for these drawbacks is a network-like architecture, a so-called NoC, operating in the four bottom layers of the OSI-model (cf. Figure 1.2). By design, parallel communication is

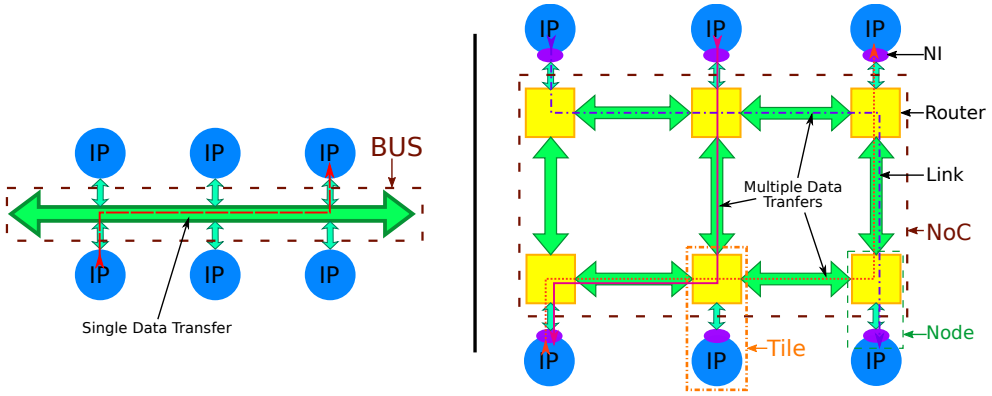


Figure 1.3: Overview over Bus and NoC.

possible where the nodes to which the elements are connected can communicate independently with each other (cf. Figure 1.3 right side). They scale well with additional elements by merely increasing the network size. This does not diminish the bandwidth nor extend the wire length, which stays the same between each node and will not grow as the network grows in size. The redundancy within its structure provides the possibility to implement fault tolerant techniques avoiding a single point of failure. Regardless of all these benefits, an NoC consumes a significant part of the chip's resources due to the additional logic. Compared to bus systems researchers expect that it can easily become a dominant part of the power budget of the chip [NND09, MO09] and this needs to be considered during the development [PM05]. Still, the benefits outweigh the disadvantages and make NoCs an integral part for SoCs and other many core systems as it is presented in Table 1.1 [FL10, BM02].

Table 1.1: Bus vs. Network Arguments (cf. Table 1 by [BM06]).

Bus Pros & Cons	**Network Pros & Cons**
Con: Every unit attached adds parasitic capacitance, therefore electrical performance degrades.	Pro: Only point-to-point one-way wires are used for all network sizes, thus local performance is not degraded when scaling.
Con: Bus timing is difficult in a deep submicron process.	Pro: Network wires can be pipelined because links are point-to-point.
Con: Bus arbitration can become a bottleneck. The arbitration delay grows with the number of masters.	Pro: Routing decisions are distributed, if the network protocol is made non-central.
Con: The bus arbiter is instance-specific.	Pro: The same router may be reinstantiated, for all network sizes.
Con: Bus testability is problematic and slow.	Pro: Locally placed dedicated BIST is fast and offers good test coverage.
Con: Bandwidth is limited and shared by all units attached.	Pro: Aggregated bandwidth scales with the network size.
Pro: Bus latency is wire-speed once arbiter has granted control.	Con: Internal network contention may cause a latency.
Pro: Any bus is almost directly compatible with most available IPs, including software running on CPUs.	Con: Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems.
Pro: The concepts are simple and well understood.	Con: System designers need reeducation for new concepts.

1.2 Faults in Network on Chip

Faults can appear in any integrated circuits under ordinary conditions due to production issues or environmental influences. This changes logic values in the circuits and influences the behavior of the NoC. The results can manifest themselves differently for example as data loss, the transfer of corrupted information, slowdowns (due to congestion, misrouting or retransmissions) or even connection loss between network parts. Especially deadlocks, where packets cannot be further propagated because they depend on another router's resources, which this one will not release because itself is waiting to transmit its packet to free up its resources for the next task. These dependencies can form a circle (cf. Figure 1.4), handicap the usage of these affected routers. It is a severe threat for the interconnection network due to the capability of blocking large parts or even the whole network [DT03, RFZJ13].

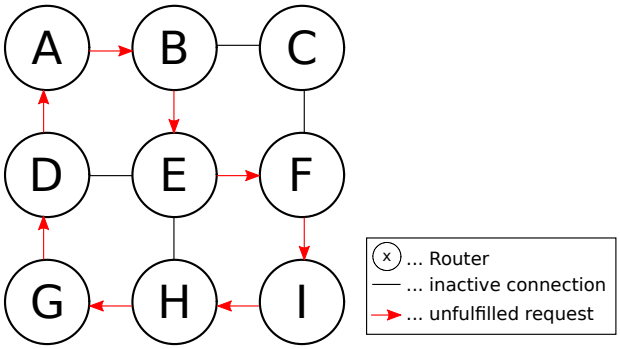


Figure 1.4: Schematic representation of routers depending on another one's resources forming a circle.

Besides the steady technology improvements, which lead to a higher integrated density, increase the systems fault probability and shorten the lifetime [Whi10]. This particular failure behavior of such a system can be seen through the classical bathtub diagram which is well known in the semiconductor industry (cf. Figure 1.5). The drop at the beginning is due to production errors, which appear as permanent faults from which the system can not recover and the affected chips are dropped out. The middle part represents the average drop out of a well running system, so the permanent faults are mostly filtered out and irregular occurring faults are responsible for this constant fault rate. The rise on the end of the graph is caused by aging effects and wear-out, which are starting to take their toll [DT03, p. 413].

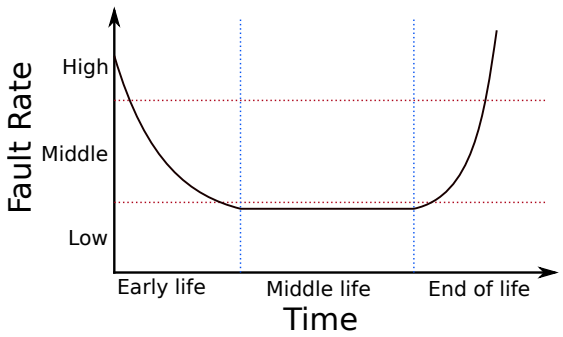


Figure 1.5: Common fault rate in a lifecycle of an Integrated Circuit (IC) product (cf. [DT03, p. 414]).

The type of impact that faults take on NoC can be abstracted into three groups: physical failure,

logic errors and, specifically for NoCs due to the added network theory, network misbehaviors (cf. Figure 1.2). Physical failures represent variations in the voltage levels of the transistors and are commonly triggered by radiation, crosstalk or aging. Logic errors are a continuation and describe the results of physical failures to the logical layer, in the form of incorrect logic values e.g. an AND-Gate signal having a logical high output even if one input is low. Network misbehavior covers the resulting errors in the router behavior and packet transfer [WEH+], examples can be found in [PPNS12].

There are several causes for physical failures in MOSFETs, which are today's main technology for chip transistors. The following list should provide an overview [RFZJ13, p. 6].

- **Radiation**: While radioactive decay in the chip's surrounding is one possibility, in ordinary situations cosmic radiation is the primary source for flipping a bit in memory cells. Additionally, the reduced size of the PN junction of the transistors leads to a higher sensitivity against radiation and the increased density of these junctions on an area means that more transistors could be affected.
- **Electromagnetic Interference**: Signal and impulses of nearby wires influence each other, which can cause delays and signal-blurs due to so-called crosstalk. Design improvements (thinning the wire) were made to reduce this failure but introduced capacitive and inductive issues. Nowadays the wires get so thin that in combination with high frequencies the *Skin-Effect* starts to become a serious issue due to increasing the wire impedance.
- **Electrostatic Discharge**: The current is without any doubt able to damage circuits. Usually, external sources produce an effect via pins of the chip but also breakdowns within the chip can lead to short-circuits.
- **Aging**: This term summarizes different effects which can appear during the lifetime of a chip. Typically the source for these issues is active from day one, but it will not become threatening for the performance until a certain level of damage is aggregated.
  - **Electromigration**: The current moving through the wires has an abrasive effect on them. This reduces material, especially in areas with high current density. It can lead to delays or worse to malfunction and even new connections between separated parts (wires) of the circuit.
  - **Negative Bias Temperature Instability**: Temperature related behavior, where the threshold voltage of a *PMOS* increases due to trapped charges in the gate oxide layer.
  - **Hot Carrier Injection**: The transistor's switching behavior gets altered (without chances of recovery), due to electrons (or holes). These electrons are accelerated by the electric fields, penetrate the gate-dielectric and are staying there. This dulls the reactivity of the transistor.
  - **Time Dependent Dielectric Breakdown**: Imperfections in the dielectric material combined with electric fields can lead to conducting paths, which break the transistor.
- **Process Variability**: Foundries strive for the highest quality of their ICs, but still their output suffers from variation within their production cycle. Imperfections can be caused by

not 100% pure used materials (wafers), problems during the doping process or the lithography procedure (the masks wear out during the usage), which in combination with the etching process cause size variation in the structures. While impurities lead to different threshold levels in parts of the chip and are accelerating the dielectric breakdown, geometrical variation adds to the problematic of different delays, *Elecotromagnetic Interference* and *Electromigration*.

- **Dynamic Temperature Variation**: Heat is produced in the IC but its intensity is depending on what operation is executed, which technology is used and which part of the chip is active. Furthermore, if it exceeds certain temperature levels the power consumption increases with the heat of the system which causes performance drops and accelerates the occurrence of aging issues and other wear-out connected issues.

Additionally, the appearance pattern of faults is an important classification parameter and can be separated into three classes [RFZJ13, p. 4].

- **Transient Fault**: Appears spontaneously for a short duration.
- **Intermittent Fault**: Appears spontaneously as well but with a longer and fluctuating appearance.
- **Permanent Fault**: As the name implies, a fault appears and will not recover.

While *Transient Faults* are commonly caused by charged particles (e.g. due to some form of radiation), *Intermittent Faults* appear due to electromagnetic interference or effects of aging and often become worse, therefore, transforming themselves into *Permanent Faults*, which can solidify themselves as broken transistors or wires [RFZJ13].

To conclude, the ability to tolerate faults is important for NoCs. The general goal is to improve its reliability, which benefits the life length of the system and its robustness. Therefore, so-called fault tolerance methods have been developed and used.

1.3 Introduction to Fault Tolerance Methods

As shown above, it is very difficult, practically impossible, to eliminate all fault sources during the production and even if this could be achieved, wear-out and age-induced errors can still appear over time. Therefore, this highlights the importance of how severely a fault affects the circuit. Will it still be able to function in a usable way or will the fault break the whole system? A part of the errors will not impact the system because so-called masking hides them, e.g. one input of an AND-Gate is erroneously low but paired with another low as input this leads to the correct result of a low output signal. Consequently, techniques to improve the sturdiness against these faults have been developed. These are called fault tolerance methods and commonly are a combination of detection (e.g. Built-In Self-Test (BIST)) and counter-measures [WEH+].
While all of them try to improve the overall reliability of the system, their techniques can be divided into three groups according to [RFZJ13, Chapter 2.3ff].

- **Time Redundancy Techniques**: By repeating the action, which has been reported to be faulty, for a specific or infinite time, it is attempted to detect and correct the data. Conventional methods are for example Retransmission or Multi-Sampling of data.

- **Information Redundancy Techniques**: These techniques add additional information to the transferred data, which allows detecting errors and even correct faults. A prominent example is Error Correction Code (ECC), which adds a checksum from the data to the transferred bit vector.

- **Spatial Redundancy Techniques**: These methods try to minimize the fault impact by duplicating parts of the circuit. These additional units are either used as a replacement if a fault is detected or as rulers by evaluating what the majority of these units decide and accepting this as the correct way. Even though the additional area consumption is quite high, Triple Module Redundancy (TMR) is a well-known implementation and often applied. There is a variety of implementation forms. On the one hand the whole circuit can be copied, on the other hand, it can also only focus on critical parts. Furthermore, the amount of duplication can vary.

These counter-measures are not equally suited against different fault appearance types. They either fix the fault directly or try to circumvent it by reconfiguring the network. Short duration faults (transient and short-time intermittent faults) are usually countered by correcting the fault itself in the transferred data, instead of coping with performance loss and power consumption through repeatedly adapting the network, which is only of temporary usefulness. However, for long duration faults (intermittent faults and permanent faults) avoiding the defective area is better. Otherwise, the short duration recovery mechanism to fix data would always be active, requiring power and introducing performance loss, as well as possibly not being able to recover the fault (e.g. Hop By Hop (HBH) retransmission will not fix a faulty wire) [WEH+].
Consequently, one tolerance method on its own will not be able to fix all reliability issues. Instead, using combinations (especially cross-layer integration) of several fault tolerance methods can be better for the reliability [RFZJ13].

However, this let the following question emerge: To which point are these methods feasible? [WHL+16] has shown in his simulations for finding the best ECC locations that reliability does not increase with the number of added ECC-units in the network. Furthermore, adding fault protection methods also introduces the possibility of these methods being erroneously themselves [WHL+16].
They are after all commonly implemented with the same technique as the rest of the IC and being close to their application field makes them vulnerable to the same reliability risk, probably causing additional new issues within the system. This is especially concerning because valuable space and power are invested in these protection techniques, which might be used more efficiently. Therefore, it is mandatory that the NoC and the fault tolerance methods have to be evaluated under the effect of faults to rate the protection strategy truly, else unexpected fault effects and synergies can appear and will pull down the whole system. This crucial topic leads to the goal of this thesis.

1.4 Goals of this Thesis

In this thesis, it is argued that there is a limit to the feasibility of extending systems with fault tolerance methods as well as that some combinations are to prefer over others.

Every added piece of IC consists of elements which can become faulty as well. The more abilities an implementation provides and the more area it uses, so much more critical the effects can be if it fails. So faults will reduce the positive impact of the fault tolerance method and it might become a burden, but how strong will these drawbacks be? So while other works focus on the abilities of their fault tolerance methods in their NoCs, in this work faults are going to be injected into the fault tolerance methods as well, to evaluate their impact on the system.

In order to achieve this, a fault injection method is developed to simulate the effects of multiple faults on a fault tolerant system. The method should be widely applicable but still be very close to the actual implementation of the system into the hardware. As a result, the restrictions of this thesis are that the implementation of the injection method is focused on NoC development, but the solution should be adaptable to other projects designed with common Hardware Description Language (HDL)-languages like *VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL)* or *Verilog*.

For future works, it should be possible to modify the behavior of the faults. It should allow to further modify their behavior and the test system should also provide different types of fault models implemented for increasing the flexibility of the evaluation.
For the whole setup, a NoC is developed in a hardware description language (*Verilog*) to get as close to an actual hardware implementation as possible. For evaluation, a test environment will be provided to test the different protection mechanisms and combinations.

1.5 Summary and Thesis Structure

To conclude, this thesis will focus on the effect of faults on different fault tolerance methods in NoCs. It will present a method to test this ability by injecting faults into *Verilog* gate level code of such a system. Selectable probabilities with which these faults appear as well as different types of faults will be tested on NoC and its tolerance methods, which were both created for this work. These added fault protection schemes will be evaluated during this process and it will be proven that influence and performance of different with errors injected fault tolerance methods are so strong that their implementation must be considered from the start of the designing process of the chip.

The following part of the thesis is the State of Art (Chapter 2), which will provide an overview of different fault models, already existing fault injection methods and the evaluation of faults in NoCs. Afterward, the Target NoC (Chapter 3) will be described by giving an introduction about NoCs and their fault tolerance methods, including the presentation of the NoC which was designed for this thesis. The following chapter discusses Fault Injection (Chapter 4), which describes how faults are injected and how the different fault models are designed. Chapter 5 contains the testbench and simulation setup, and it provides insight on the results and comparison between the different fault tolerance configurations. Last but not least, the Conclusion (Chapter 6) will sum up the evaluation and results of this thesis and gives a small outlook for possible future enhancements.

2 State of the Art

This chapter discusses how to inject and evaluate faults on chips, afterward focusing on the fault models, which shall resemble the issues within the systems considered in this thesis. Therefore several already existing works are compared, in particular the differences to this thesis.

2.1 Fault Injection

As mentioned in Chapter 1 there are many sources for faults in Application Specific Integrated Circuits (ASICs), and it is important to assure that the system can cope with them to a striven level. In this context *Fault Evaluation*, as the name already suggests, can be used to determine the performance of the system under the effect of faults. While in an actual implemented system the physical causes (cf. [RFZJ13]) for these faults are researched, it is beneficial to test the impact of faults on the system beforehand. Actual tests can be done in hardware or by simulating the system in software, whereas the latter one can be further divided into software-implemented fault injection or simulation-based fault injection [SSM08]. It is worth mentioning that there also exist non-fault injecting models, which send non-Bernoulli random number streams into the input of a circuit and evaluate the reliability by decoding the output stream in perspective of the input [HCL+]. While modeling the faults by probabilities of defect logic was an interesting approach, the lack of being able to observe the behavior logic as well as the lack of adding design specific code to it, reduced the usability of this approach for this thesis.

A prominent way to prove a concept for designing a system is to use high-level simulation-frameworks. These are commonly programmed in a non hardware description language like *C*, *C++*(*SystemC*) or *Python*. This not only allows an abstract approach to the implementation (code written can not be directly used on a chip) but provides higher flexibility for code adaptations and great possibilities to add further evaluation elements for benchmarking the performance, latency or power consumption [CMM+15, WEH+]. For Network on Chips (NoCs) popular frameworks are *Noxim* [CMM+15] [12] or *POPNET* [10] (used by [WEH+]), but also further developments in research are made [YA10]. These frameworks allow a wide variety of changes to set up the system to the user's wishes. Configuration options can support different network topologies, a variety of packet injection patterns or customizing the data transfer format. The simulators do not need to be built solely for NoCs, a modified communication network simulator is also a possible way to implement a test suite but will have shortcomings due to its origin, as shown in [AWAN06].

In context with fault evaluation, simulators have the disadvantage of not being an actual design implementation of the chip because they do not consist of gates and wires. However, there are still ways to emulate their faults for example by injecting errors into links, router and network interfaces or into the data packets to test different Error Correction Code (ECC)-variation on a system [YA10]. However, this is a high-level approach and in regards to this thesis and it is not possible to get that close to the hardware implementation. This would also require that already existing Hardware Description Language (HDL) code needs to be reimplemented into the specific simulator, which led to the decision to not use this concept.

The exact opposite approach for evaluation is hardware based. This method impresses by its high speed due to the code being executed on a Field Programmable Gate Array (FPGA) [SSM08, NPCZR15]. However, by being bound to the hardware, there are limitations, but certain solutions have been developed. *Fired* reconfigures only selected Static Random Access Memory (SRAM) cells of the FPGA, in which the Device Under Test (DUT) is located to enable time or event triggered faults [NPCZR15]. The system does not need to halt for this Single Event Upset (SEU) fault injector but the technique only works on supported FPGA-boards (*Virtex-5* board, and there are future plans for the *Zynq* devboard). Another method is called *Fito*, which also benefits from higher simulation speed (approximately 79% more than software simulation) and makes use of synthesizable fault modules [SSM08]. It stands out by its small area overhead and the well-integrated control and evaluation workflow (comparison between *golden run*[1] and *faulty* trace file). For this thesis external physical effects are ruled out as the source for errors on the chip, like forcing faults with electric fields or heat due to the possibility of destroying the chip and the requirement of specialized equipment.
The method *ARROW* is a way for fault injection and implemented via VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) [BH09]. The faults are caused by so-called *saboteur* units which get placed between the modules of the DUT and can change the values and delays of signals. These also contain a controller and a Random Access Memory (RAM) module for correctly triggering the faults but still have a small footprint on the chip to be able to run during the test on FPGAs. It is noteworthy that these units can be modified during simulation via a *C++* application. This ability may be interesting but is of secondary importance. Additionally, *ARROW* is only able to corrupt signals between two modules (VHDL modules or Intellectual Property (IP) cores), which would require an individual fault model for each of these modules. However, this is not desired, a uniform fault module is preferred for this work.
In the research of [Man13], another hardware-based approach, also FPGAs were used for the simulation while focusing on the behavior of chips under the influence of radiation in high atmospheres an automated SEU fault injection method was developed. This automated tool, called *NETFI* (Netlist Fault Injection), injects a fault chance in every memory by modifying the D-flip-flops, the RAM, the Look Up Tables (LUTs), the logic gates, and the multiplexers. This approach is faster due to its hardware execution, but it can reach a more substantial overhead for bigger circuits because of the extra logic and signals for reaching the fault injection units.
Neither of these approaches above was usable for this project, be it due to the missing flexibility for the fault modules or being limited to a specific hardware.

As a result, it was decided to use a software simulation based approach for this project. Most of the reviewed research differ on how they inject the faults, in which language they are implemented

[1] Fault Free Trace File

and which types of faults they support. Commonly they consist of injection tools, a simulation setup and some observation logging function. Functionwise they modify existing HDL code (be it VHDL or Verilog/SystemVerilog), run a so-called golden trace file (fault-free run) and compare it to faulty ones. In the following some software simulation approaches are presented.

MEFISTO-L [BPC98], successor of *MEFISTO*, is a fault injection tool and has overcome the limitation of its predecessor of being stuck to only commands which the VHDL-simulator supported. Its goal was to detect weaknesses for fault tolerance improvement by splitting the signals and putting probes or *saboteurs* (units which create faults) in between the signals. A Graphical User Interface (GUI) is used for placing these Fault Injection Units (FIUs) and the *saboteurs* could be triggered by certain signal-expression from another probe or a special signal-wire, which activates the fault for a certain time. While the method of placing these *saboteurs* seems to be very good, the limitation of only supporting VHDL, as well as the expected effort for placing these and configuring the conditions, are drawbacks to this method.

INJECT [ZME03a] [ZME03b] is able to inject faults in *Verilog* and in VHDL code. This is possible because the used test environment (*Modelsim*) supports mixed-mode simulations, so files, which are programmed in different languages, can still be simulated together. To select *Verilog* as the implementation language for the fault models also opens up the possibilities for injecting faults into more design levels. This tool supports fault models for *switch-level*, *gate-level*, *Register Transfer Level (RTL)-level*, behavioral-level and structural-level. Switch and *gate-level* faults are placed by splitting the wire and inserting a FIU there, which can resemble different types of faults (for additional information see Chapter 2.2). All of these faults are triggered by Fault Injection Signals (FISs), which need to be placed and configured by the user.

In their research [AM10] made use of another form of mixed simulation by converting the DUT into a circuit level netlist and simulating it via Simulation Program with Integrated Circuit Emphasis (SPICE). They focused on testing three typical types of faults (single event transients, electromagnetic interference, and power disturbance faults), which were implemented in *Verilog-A*, which is used for simulating analog circuits. However, as stated in the paper, the amount of *Verilog-A* files should be kept to a minimum for performance reasons, which is not compatible with placing faults in a whole NoC with thousands of gates.

There are also other concepts of software based simulation, but they turned out to be of limited use for this thesis. For example, [AZl12] points out how important it is to evaluate the influence of faults early on and tests this by merely forcing signals to a specific value (*stuck-at-0*, *stuck-at-1*) with his simulation environment (Mentor Graphics Modelsim). This is only feasible for small circuits due to manually selecting the "faulty" signals. Instead of doing an exhausting and slow simulation on the *gate-level*, [TAZ03] aims to find a way to cover their faults via a RTL fault list. For this purpose, he runs the fault injection on *RTL-level*, which benefits the performance. However, only one signal/wire of the DUT will be faulty per each test run, which would be a disadvantage for a much larger NoC circuit.

Therefore, the most promising concept for this thesis was to create its own variation of a *gate-level* fault injection method, which has similarities with [ZME03a]'s work but instead of placing FIUs in selected parts, the whole system is targeted. Furthermore, this would allow using a uniform

fault module by focusing on the gates of a design, but it could also be used for both dominant HDL languages (VHDL can be synthesized to *Verilog gate-level* code).

2.2 Fault Model

As pointed out in the last section there are several ways to inject faults into a system. However, how do they behave, when do they become faulty and what does it mean to become faulty? As predictable they shall resemble faults in an actual Integrated Circuit (IC), without having to set up all circumstances which create it.

As a result fault models for NoCs are developed to resemble physical failures in a system. These models work with functions through abstractions. A simple comparison with a valve inside a machine can be used as an example for further abstraction. This valve has an assumed maximum count of actuation until physical failures can appear due to friction, material fatigue, and wear-out. An abstraction for a software simulation would be that after a particular time of closing and opening, it would be stuck and will stay that way, even if a different control signal appears. This would avoid implementing the wear-out behavior, reducing the complexity of the fault model but still providing its failure. In the perspective of ICs, the abstraction from physical failures is logic errors [WEH+]. Instead of creating a fault model including voltage/current fluctuations, shortages due to wear-out and broken delay constrains for the transistors, the result of the changed logical value is used.
Furthermore not all physical faults create logic errors, some get masked by the system [HW16, KH04]:

- "Electrical masking" summarizes effects where the voltage variation pulse (e.g. due to SEUs) will be attenuated by the properties of the gates, which it passes through.
- "Logical masking" resembles the fact that the inputs of a gate are not all relevant for its output, so for example a correct high signal in one input of an OR-gate will overrule all the faulty inputs which are low.
- "Timing masking" masks faults which appear outside the time frame in which the gate evaluates its inputs and will not store or process the error.

A further abstraction for NoCs is *Network Misbehavior* [WEH+], which resembles the faulty behavior of the elements inside the routers. These faults can corrupt data or mislead packets but cannot be derived directly from logical errors because error behavior also depends on network theory and the NoCs traffic distribution ([PPNS12] presents several of such misbehaviors).

To be able to work with faults they need to be categorized. [BH09] distinguishes physical, logical and functional layers for faults and tries to portray these in his fault model implementation. Regarding the physical layer, the Printed Circuit Board (PCB) is seen as "the base of all evil" and he states that in order to avoid the huge amount of implementation complexity it is necessary to use a high-level abstraction [BH09, Chapter 3.1].

Therefore, he represents the physical faults in a logical fault model, "which makes it possible to inject faults without manipulating the design physically" and leads to the logical layer [BH09, Chapter 3.2]. Faults in the *switch-level*, which are often caused by manufacturing errors, can be modeled by using the *Bridging Fault* (40% to 50% of all faults can be modeled by it), which shows an incorrect connection between two or more wires.

His structural model sees the circuit as a connection of gates, where links can become faulty. On this level *stuck-at* faults cover 80% - 85% of all physical faults. A modified version of the *stuck-at* fault is also used to portray delay faults by keeping the signal for a longer duration before updating it to the changed value [BH09, Chapter 3.2].

To resemble impact is one part of implementing the fault model, the other one is to recreate its appearance pattern. In NoCs for example faults can occur due to crosstalk, thermal issues, cosmic radiation, aging or combinations of these and often cause specific types of faults. Crosstalk commonly triggers transient faults due to creating temporarily electric fields/noise, which disappear after a few cycles. Contrary heat issues result in permanent failures, from which the circuit will not recover due to damaging the structure. *Intermittent* faults are often caused by aging, where parts of the system wear-out and start repeatedly misbehaving, often transforming themselves into permanent faults over a longer duration [WEH+, Chapter 3]. The different pattern should be tested on a system to get valid information about its reliability.

To decide which level of detail shall be used for a fault model the advantages of each layer were evaluated. On *switch-level* additional information (e.g. used transistor technique and its placement) would be needed which is not relevant for this thesis and would add unnecessary granularity, primarily because for this thesis no actual hardware chips will be created.
On *gate-level* it would be possible to have a uniform fault model for the whole NoC because every implemented part could be broken down to gate logic, where faults are injected.
The last abstraction layer, the *functional layer*, has no general model for faults due to the fact that every code module is different and would need its own fault model. There is still ongoing research in order to find an automated method for a general model on the functional layer because it would be a significant improvement for the performance in large system simulations compared to an all *gate-level* simulation. [TAZ03] found a way to approach this goal but states that further work is needed to have a more reliable model.

In summary, resembling logical errors as fault model on *gate-level* allows having a fine enough granularity for the necessary simulations. Its usage as a uniform fault is a significant advantage and its ability to cover a wide amount of faults resulted in using it for this thesis.

2.3 Evaluation of Fault Tolerance in Networks on Chip

A common way to evaluate a faulty system is to use *Golden Trace* files as shown in [AM10, ZME03a, SSM08]'s work. These files are created by a fault-free run of the simulation and display how the system should work if everything runs perfectly. A fault would create a diversion from the fault-free run and compared to a tree a new branch would be followed, which can further

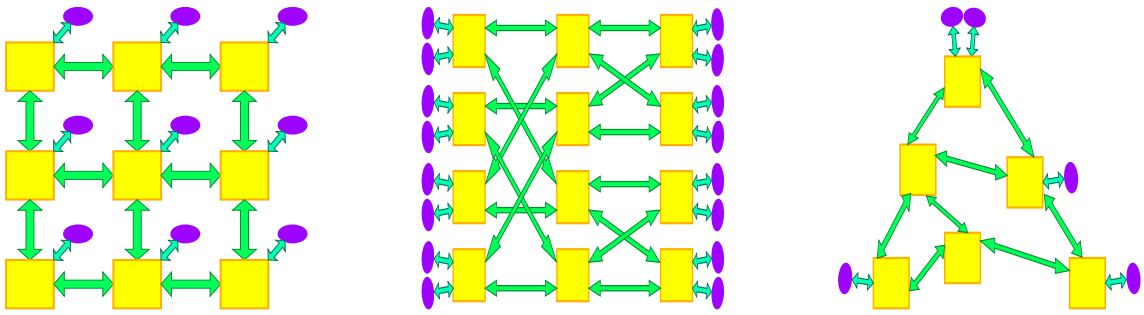
be split up due to additional errors. It was concluded that comparing faulty trace files with the golden ones works better with smaller systems, where the impact of the fault and the divergence from the *Golden Trace* files is somehow limited for the rest of the simulation time. On the other hand, larger systems with a high level of interconnections will quickly differ from the perfect run, thus making the approach for this thesis of questionable use.
[RFZJ13] points out that the fault tolerance research for NoCs becomes more goal-oriented regarding their functions and constraints, but their evaluation lacks comparability, suggesting that standard scenarios with defined performance indicators would be beneficial. In [WEH+]'s paper three metrics are defined. Firstly, the delivery time, which is the duration of the first time the packet gets injected in the NoC until it is successfully ejected. Secondly, the retransmission time, which is the additional duration the packet needs for retransmission. Lastly, the delivery rate, which resembles the percentage of injected and correctly received packets. The first and last parameter will be adopted for this thesis to test the performance.

3 Target Network on Chip

This chapter focuses on the target Network on Chip (NoC) used for the fault tolerance method evaluation in this thesis. The NoC follows the popular features for NoC research and was specifically developed for this thesis. The topology, the routing behavior, the router structure as well as the fault tolerance methods are introduced. In each aspect, the design of target Network-on-Chip is given after reviewing published works[1].

3.1 Topology

Topology defines the structure of the connections between routers and Intellectual Propertys (IPs). The capabilities of the NoC are located in the bottom four layers of the Open Systems Interconnection (OSI) model, and the connected IPs resemble the fifth layer (cf. Figure 1.2). The IP is connected via a Network Interface (NI) (colored violet in Figure 3.1) to the router, this group is also called a *tile*. The routers (colored yellow in Figure 3.1) are connected via links (colored green in Figure 3.1) to exchange data packets. For different purposes the topology can be adapted, e.g. mesh, butterfly or irregular (cf. Figure 3.1). However, the majority of scientific works (e.g. [WHL+16, LYA16, SZBR09, GAP+, BM06]) focus on a regular network and so the most popular design is the mesh topology.



(a) Mesh Topology. **(b)** Butterfly Topology. **(c)** Irregular Topology.

Figure 3.1: Different NoC topologies.

[1]The work of [DT03], *Principles and Practices of Interconnection Network* was the main resources for Chapter 3 except other sources are being cited.

The mesh topology is a direct network. Every router has an exit to a Processing Element (PE). Its homogeneous arrangement is not only beneficial for extending the network more efficiently, but it also allows to keep the wires short, thus supporting a higher communication frequency. The grid-like structure, which is used for bidirectional communication between two routers, provides a good path diversity but adds an increased hop count, which is an accepted trade-of [DT03, Chapter 5].
Examples of its usage can be found in *CONNECT* [ACM/SIGDA12][11] and [LYA16, PNK+06, WEH+, SSM16, Ebr13], which gave convincing reasons to use this mesh topology for this thesis. The implemented design consists of 64 nodes[2] (8 x 8 grid) in a convectional two dimensional mesh topology (cf. Figure 3.2).

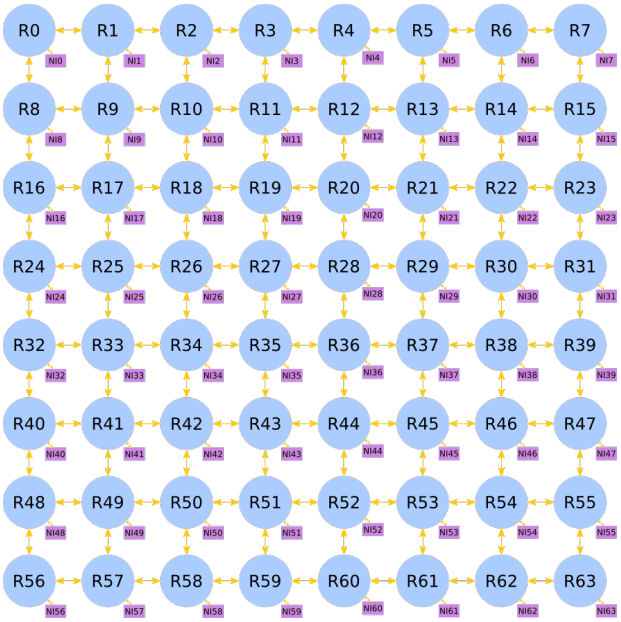


Figure 3.2: Target NoC topology consisting of 64 Routers (colored blue) and their NIs (colored violet).

The routers are connected to each of their four neighbors (symbolically positioned in the hemispherical directions north, east, south, west) as well as to the local interface via which data can be transferred in or out of the network via the NI. This structure is displayed in Figure 3.3, which resembles a router of the target NoC, which has an additional Virtual Channels (VCs) into the north and south direction.

3.2 Routing

The routing assures that the data uses the correct path to reach its destination. This behavior marks the significant feature compared to the common bus system and is accomplished in two different ways within the network: packet-switching or circuit-switching [DT03]. This thesis will concentrate on packet-switching due to its preferred usage in NoCs as the hop-wise transfer of packets reduces the length of wires and allows to operate with higher frequency in the hardware implementation. The routing of these packets is a crucial aspect to balance the traffic load in order to avoid deadlocks (cf. Figure 1.4) and livelocks. Livelocks describe the situation where

[2] Resembling the router and its NI.

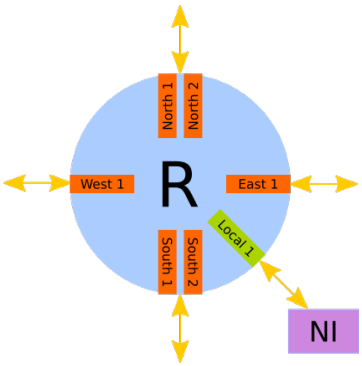


Figure 3.3: An overview of the router of the target NoC.

packets are continuously forwarded but will not reach their destination.

The three main concepts are [DT03, Chapter 8]:

- *Deterministic Routing*, which always selects the same path between source and destination and makes no use of path diversity.
- *Oblivious Routing*, which sends the data packet to a random router which will forward it to the destination.
- *Adaptive Routing*, which minds the state of the network and selects the route accordingly.

The decision for route selection is controlled by either Look Up Tables (LUTs) or algorithms. Both provide the direction depending on the packet's destination but the former can be quickly adapted to new topologies but has to be hardcoded into the system, or else a discovery-phase is necessary during the startup. Algorithms, on the other hand, are defined only for a specific topology but they have the advantage of a small footprint and high execution speed [DT03, Chapter 11].

The routing is also a part of the flow control of the packets, which covers the aspects of allocating the necessary resources (channel, buffer, other control signals) for further propagating the packet towards its destination. These techniques can roughly be split into bufferless and buffered flow-control. The bufferless mechanism will not store the data so if they cannot fulfill the demands they will misroute or drop it. In contrast the buffered flow control stores packets on their path by using buffers. It should be mentioned that a packet is split into several parts, which are called FLow control unITs (FLITs) and these are forwarded in the network. These FLITs are stored in the order of their arrival. This allows a buffer to contain parts of multiple packets and improves the utilization of the storage units. A packet commonly consists of a *headflit*, which contains necessary routing data or system relevant information, zero or several *bodyflits*, which transport the data, and the *tailflit*, which signals the end of the packet [DT03, Chapter 2]. This type of flow control is implemented in the target NoC, avoiding data loss while waiting for the resources to become free [DT03, Chapter 12].

The exact FLIT-structure for the target NoC can be seen in Figure 3.4, showing the name of each FLIT and the content of their 34-bit wide payload. For the simulations, the data of the *bodyflit*

Name	Content																															
Headflit	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	Type		Source						Destination						Reserved				ID												Flags			
Bodyflit	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	Type	Data																																
Tailflit	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	Type	Checksum																																

Figure 3.4: Structure of the 34 bit wide FLITs in the target NoC.

and the checksum of the *tailflit* were further modified for tracking purposes.

For the situations where a packet occupies a channel and blocks the system, VCs are introduced, which are like an "additional channel" inside a router. So if for example a packet in VC A cannot be forwarded, the channel completely blocks the switching process because VC B of the same port can step in and prevent a possible deadlock [DT03, p. 221ff].

Bearing all this information in mind it was decided that the target NoC uses a modified DyXY-Algorithm, which is deadlock-free, with *wormhole flow control* using VCs into the north and south direction. Packets get moved first in x-direction then, when the same x-position as the destination is reached, in y-direction using the neighboring buffer levels for flow control.

3.3 Router Architecture

The structure of the router decides how the packets are stored and forwarded and which elements are in control of these actions. Variations of router designs can be found in [DT03, p.305ff] or [EYPZ09, MO09, KPN+05, PNK+06] but the design presented in this section focuses on the target NoC. In general the elements in a router, see Figure 3.5 (cf. [DT03, Figure 16.1], [PM05, Figure 2], [BM06, Fig. 10.]) for a schematic overview, can be associated with either the *Control Path*, which is responsible for handling the route calculation, allocating the necessary resources and other logical actions, or the *Data Path*, whose purpose is to buffer, switch and transmit the data [WEH+].

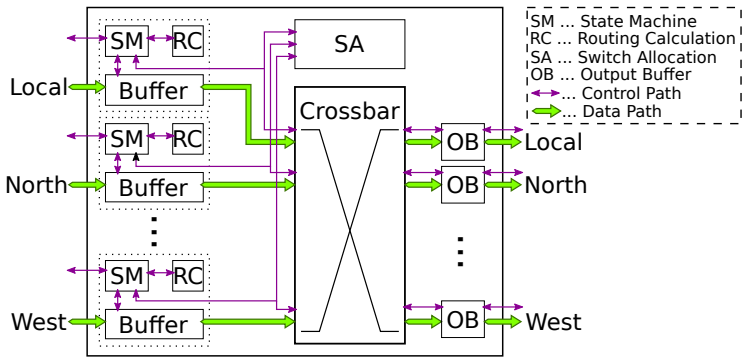


Figure 3.5: Overview over *Data* and *Control Path* of the simplified router of the target NoC.

In more detail, the procedure inside the *Data Path* would start with the packets arriving FLIT-wise at the input buffer and then getting switched over the crossbar to the output registers. The *Control Path* inside the router, which the packet has to pass through, can be represented by a five-stage pipeline as it can be seen in the left side of Figure 3.6 and shows how each FLIT is processed

for transmission if as in an ideal case a stage would take only one cycle. The abbreviations are explained in Table 3.1.

Table 3.1: General states of packet transmission (cf. [DT03, Chapter 16]).

State	Description
Routing Calculation (RC)	Calculates which outgoing port should be selected for the next hop
Virtual Channel Allocation (VA)	Selects one of the pending VAs that is trying to transmit its packet
Switch Allocation (SA)	Is responsible for avoiding conflicts in the switch configuration
Switch Transaction (ST)	Describes the action of forwarding the FLIT to the output buffer
Link Transaction (LT)	The transferring of the FLIT into the next router

In the target system (cf. right side of Figure 3.6) the procedure starts also with the RC but the VA is fused with the SA, due to a different implementation of VCs. These phases are handled by a *State Machine* module in the code, except the LT, which is handled by the output buffers at the exits of the router. The pipelines are finished with the arriving of the *tailflit* and the allocated resources are freed up for further use [DT03, Chapter 2].

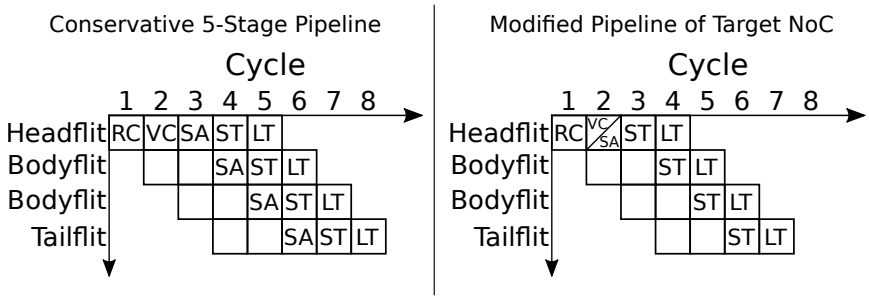


Figure 3.6: Stages of FLIT processing inside a router.

Actual implementations can differ in the number or order of the stages, and these actions can stretch over several clock cycles until they are finished. However, in the optimal case, they would take only as few cycles as possible thus leading to different approaches like being grouped or being so tight designed that a FLIT can be sent out and received in one cycle [EYPZ09].

3.4 Fault Tolerance

Fault tolerance methods are used to improve the reliability of NoCs against faults and their evaluation and the effect of different fault tolerance implementations is the main focus of this thesis. The general approach is that an error needs to be detected, confined and if possible the affected system should be recovered ([DT03, p.414]) for which [WEH+, Fig. 2] gives an example of a general fault tolerance workflow. This thesis tackles the fault tolerance on a smaller scale, providing a fault tolerance method for each OSI layer occupied by the NoC except the physical one.

Detection can generally be achieved by Built-In Self-Test (BIST) (also known as Built-In Self-Diagnosis (BISD)), which only detects faults by values it receives from the system or by providing its own test pattern to check for anomalies [RFZJ13, PPNS12]. The test frequency is an important parameter to have a good fault detection. If it is too high, it may introduce overhead in

power and area consumption as well as a reduced throughput but if it is too low it can let faults slip through. A more lightweight but limited approach is possible with *Assertion Units* [LYA16, Chapter 2][PNK+06], which check for irregular conditions like if the ingoing and outgoing FLITs of a router are in balance. Due to the overhead of BIST, *Assertion Units* in routers were preferred for this thesis' target system and will be implemented in one of the fault protection schemes.

As already mentioned in Section 1.3, these fault tolerance methods can be divided into three groups of redundancy namely spatial, time and information. Another classification, which will be used in this thesis, is to map these solutions to the corresponding layer in the OSI-model, due to the fact that a NoC covers the *Physical Layer*, *Link Layer*, *Network Layer* and *Transport Layer* of the OSI-model (cf. Figure 1.2). Fault tolerance methods can occur in multiple layers but their implementation would be different[3] [RFZJ13].

The first layer of the OSI model, the *Physical Layer*, will not get a fault tolerance method because *Verilog* which is used for the simulations, is good at describing logic circuits and is not good at describing analog features, e.g. different voltage, influence of crosstalk. *Verilog* can make use of external code modules, but this would need additional information, some only available after the layout process, and stress the performance, which in perspective of the size of the simulation would be a critical limit.
The next layer is the *Link Layer*, which has to cope with corrupted or lost packets. The fault detection can be done by *resampling* or the use of data encoding. The tolerance methods on this layer can be realized by *Multisampling*, *Retransmission*, *Split Link Transfer*, *Encoding* (e.g. Hamming Code) or *Spatial Redundancy* (e.g. Triple Module Redundancy (TMR)) [RFZJ13]. For this thesis encoding the data seems to be the most promising fault tolerance method in order to avoid adding redundant elements (for simulation performance purposes). *Hamming(7,4)* linear error correction code is used to create a checksum for every FLIT (27 bit checksum for 34 bit FLIT), which is attached to the top of the FLIT itself [WHL+16]. The encoder is placed at the ingoing NI and a decoder at the outgoing side as a final check. Inside every channel, a so-called "Intercoder" is placed, which corrects errors in the FLITs.

Faults in the *Network Layer* can cause FLIT-loss, FLIT-duplication or misrouting. While time redundancy is covered by the layer above (end-to-end retransmission) and/or below (hop-by-hop retransmission), on this layer the focus is set on information and spatial redundancy. The information technique sends the duplicated information towards its destination, which variations can be summarized by the term *flooding*. The spatial one is already implemented by all the additional routers in a network and provides spare routes between the packet source and destination, but the routing calculation must support alternative routes and needs an ability to detect defect routers [RFZJ13, Chapter 3].
The higher packet count of *flooding* inside a faulty NoC increases the risk of deadlocks, so instead of *flooding* a fault-tolerant routing system is added to the fault tolerance methods in this work.

To detect faulty routers the *Assertion Units* inside the NoC are used and if their counted signals exceed a certain threshold an "abandon signal" warns the neighbor routers. The routing calculation then reacts to these signals and reroutes the packets to avoid such broken links, still using a $DyXY$-algorithm [Ebr13].

[3] E.g. *Retransmission* can be realized on *Link-Layer* as hop-to-hop retransmission and on *Transport-Layer* as an end-to-end retransmission

Lastly, the *Transport Layer* is covered which assures the communication between the PEs and NoCs over NIs. Faults on this layer can cause wrong packet addresses or even packet loss.
The fault localization inside the network is difficult because these endpoints are not able to review every step of the packet but mention that techniques exist, which probe in every direction or presume the fault position from evaluating error pattern. Generally, Error Correction Code (ECC) and retransmission techniques are used to perform fault tolerance on this layer [RFZJ13, Chapter 4].
While ECC was already implemented on the *Link Layer*, it was decided to implement a retransmission technique for the *Transport Layer* for the target system by connecting specialized elements to the NIs.

It should be noted that retransmission alone is not able to recover from permanent faults, so the path needs to be adapted for this problem. Also, redundancy techniques can be performed on this layer by duplicating for example parts of the NI or by calculating alternative routes, which is only possible if source routing is used (the packet path is completely defined at the source router) [RFZJ13, Chapter 4].

Aside from the implemented fault tolerance additions named above, a sturdier version of the state machine was added. This so-called *Secure State Machine* has additional abilities to cope with faults. While the normal state machine can drop a faulty headflit (ECC-module required or an *Assertion Unit*), the secure state machine is capable of waiting to possibly let the fault recover (1 cycle was used in this thesis), giving the system a higher chance to transmit more packets. Additionally, if a new *headflit* appears before the *tailflit* arrives the secure state machine reacts by stopping the transmission of the *tailflit*-missing packet and continuing with the new arrived *headflit* since otherwise this new packet would be lost. Furthermore, it can wait for a cycle if an error in other elements occurs (e.g. routing calculation or switch allocation) and if it will not recover after a predefined amount of cycles, it goes back to the *IDLE+RC* state. This can lead to a loop of the state machine, but the *IDLE+RC* will give all the other state machines in the router a chance to continue their work and prevents that the faulty state machine propagates its faults.
Parts of these abilities depend on the *Assertion Units* of the *Fault Tolerant Routing*, or the error output of the ECC tolerance method and need to be activated to work. These combinations can and will be tested separately during the simulations to evaluate their performance.

Each of these fault protection schemes will be tested in the simulations, but as [DT03, RFZJ13] and [WEH+, p.3] state a mix of fault tolerance methods on different layers is more capable of countering faults than just using a single method and will be considered during the simulations of this thesis.
Such combinations are already done, for example by [PNK+06], who proposed a system using retransmission (Hop By Hop (HBH) with Forward Error Correction (FEC)) and specialized VAs, which are also used to locate and break deadlocks in order to avoid resource hungry deadlock-free algorithms.
Also [LYA16]'s work focuses on the concept of slicing the router into smaller parts, not only for stability but also for power-saving benefits. A channel consists of three narrower channels, which run in parallel mode for peak-throughput, single mode to recover from faults and in a TMR-mode to cope with faulty parts. However, both examples display that one of these methods cannot be easily deactivated without affecting the rest of the system, which makes it challenging to evaluate also each method separately.

Therefore, the target NoC is focused on simpler, but separated fault tolerance mechanisms, which can be switched off without debilitating the rest of the system but also have synergies if activated together (e.g. ECC signals the fault-tolerant routing that the packets from one neighbor are always defect). The first approach to use was an auto-generated NoC from *CONNECT* [ACM/SIGDA12] as a base for the test system which failed because the code was on *gate-level*, which made it practically impossible to gracefully expand the system to the simulation requirements. This resulted in creating a NoC specifically for this thesis, but it has not been focused on low power consumption or high latency performance and will be only used as a test system.

4 Fault Injection Implementation

This chapter will describe the procedure of the fault injection, the implemented fault model design and the fault injection process, which which includes a short description of the used framework *pyverilog*.

4.1 Fault Injection Flow

The Figure 4.1 displays the fault injection flow designed for this thesis. First the *Verilog* [Ver06] code of the finished target Network on Chip (NoC) is synthesized file by file to gate-level, which provides a uniform structure of these files for the fault injection process in the next step.

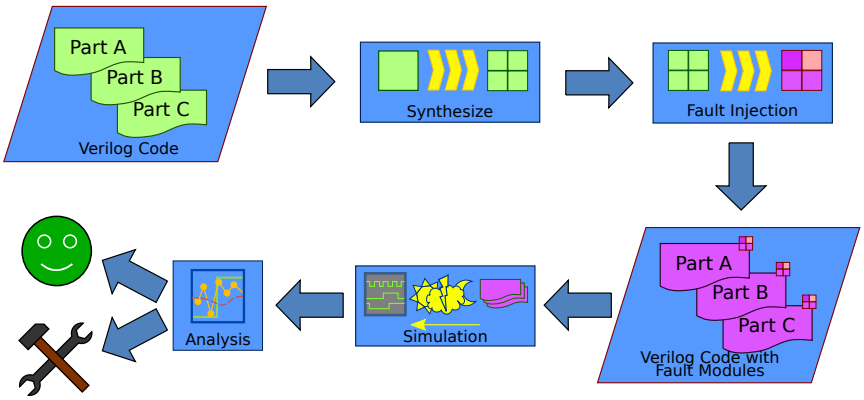


Figure 4.1: The workflow of injecting the code with faults for simulation and analysis.

During the fault injection step, fault modules are placed by a script in each module of the target system, which also includes its fault tolerance methods. After all *Verilog* files are ready the simulations process starts, for which *QuestaSim* [3] by Mentor Graphics was used to execute the testbench written in *Systemverilog* [Sys13]. In the final step, the simulation results are analyzed, which marks the end of the fault injection flow for this thesis but the fault injection process is not bound to this design. Other *Verilog* systems could be tested as well and afterwards decided to be good enough ("smiley face") or that rework is needed ("hammer and wrench"). The simulation results will be shown in Chapter 5, while the following sections will focus on the fault module followed by the fault injection script.

4.2 Fault Module

The fault modules are coded elements in the NoC and will be used to resemble the fault model inside the system. Letting the fault model resemble logical errors on *Gate Level* is seen as optimal. Logical errors provide an abstraction from the physical faults without needing knowledge about the structure, placing and technology used for the Integrated Circuit (IC), while they can still be used in a uniform way for the whole system. A further abstraction would require an adapted fault model for every logic module inside the NoC, which would be not only challenging to retrieve but also not transferable for possible further NoCs.
For its development *Verilog* was chosen as the programming language. It is widely used as a Hardware Description Language (HDL) to model chip designs and allows to describe designs from *Behavioral level* over *Gate Level* down to *Switch Level*.

Common logical errors, which appear in ICs, are *stuck-at-0* or *stuck-at-1*, which lets the output value of the gate stay 0 or respectively 1, and in some cases also *bit-flip* error, practically inverting the result of the gate. Simplified, it can be seen that every gate, to which circuits can be broken down, has a chance to get faulty, which affects its outgoing signal. At the beginning of the development, it was the idea to use the resolution function[1] of *Systemverilog* [Sys13] to manipulate the signals but not all of its necessary features were supported by simulation environments like *ISE* from Xilinx Integrated Development Environment (IDE), *Icarus Verilog* or *QuestaSim*. The thought behind this fault injection is now to place a fault module at the output of every gate to modify with a certain probability its behavior in the system. Aside from the logical errors named above, it should also be possible to resemble different appearance patterns, to be precise transient, intermittent and permanent ones.

The implemented model has two input ports, named `in` and `faultyClock`, and one output port, named `out`. The output signal of the selected gate will go into the `in` port and will exit, possibly modified, via the `out` port as it is displayed as the red module in the right side of Figure 4.2.

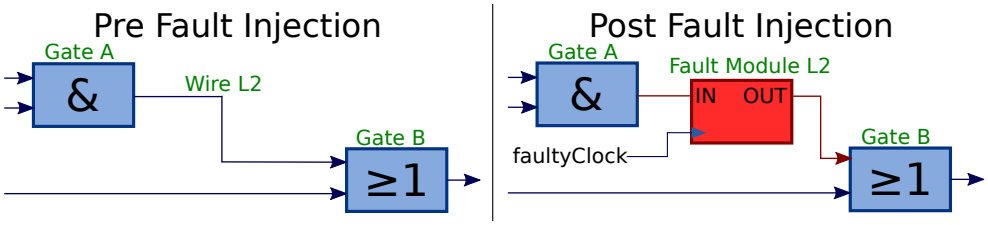


Figure 4.2: Fault Module placed between two gates.

Inside the module is a state machine witch a *CORRECT* and *FAULTY* state, which can also be represented as Markov Chain (cf. Figure 4.3). These states decide if a specific bit is set, which decides if the input `in` will be modified before it is passed to the output `out` or not. To change between these states, a random number is generated every cycle, for which the `faultyClock` is used as a trigger and compared with predefined probability values, which are stored in a parameter file and are valid for all fault modules in the NoC. If the number is below the value, the module will change its state else it stays in its current one.

[1] This function resolves which signal level has a wire connected to low and high signals.

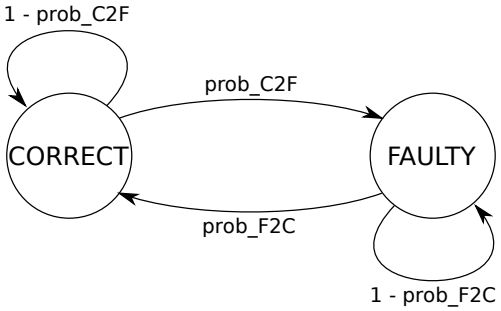


Figure 4.3: State machine of the Transient/Permanent Fault Model.

This model allows to represent transient and permanent error, which can also be displayed in Markov matrices, to present the probabilities of the state transfer.
In such matrices, the rows and columns are matched to individual states, in example 4.1 marked as A, B, and C. Each column represents transfer probabilities from a state. For example column A has a 0.3 chance to stay in state A, a 0.7 chance to switch state B and no chance to transfer to state C. Therefore, the sum of a column must equal 1, containing all transfer possibilities.

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 0.3 & 0.2 & 0.1 \\ 0.7 & 0.2 & 0.2 \\ 0 & 0.6 & 0.7 \end{bmatrix} \end{array} \tag{4.1}$$

The matrix 4.2 represents the transient error and has aside from the probability to appear, named *prob_C2F*, a high chance to return to the *CORRECT* state,named *prob_F2C*.

$$P_{transient} = \begin{bmatrix} 1 - prob\_C2F & prob\_F2C \\ prob\_C2F & 1 - prob\_F2C \end{bmatrix} \tag{4.2}$$

In the matrix 4.3 the permanent error is displayed, which looks quite similar to the matrix $P_{transient}$ but has a zero probability to recover.

$$P_{permanent} = \begin{bmatrix} 1 - prob\_C2F & 0 \\ prob\_C2F & 1 \end{bmatrix} \tag{4.3}$$

For intermittent errors the state machine is extended by a third state (cf. Figure 4.4), called *GROGGY*, which is positioned after the *CORRECT* state (leaving probability named prob_C2G) but before the *FAULTY* state (leaving probability named prob_F2G).

In the *GROGGY* state, the error is not active (leaving probabilities named prob_G2C and prob_G2F), but there is a certain chance that it switches to *FAULTY*. It can also be displayed as Markov matrix, represented as the matrix $P_{intermittent}$ in 4.4, which is extended by the state *GROGGY*. As it can be seen there, there is no direct connection between *CORRECT* and *FAULTY* state (probability 0).

$$P_{intermittent} = \begin{bmatrix} 1 - prob\_C2G & prob\_G2C & 0 \\ prob\_C2G & 1 - prob\_G2C - prob\_G2F & prob\_F2G \\ 0 & prob\_G2F & 1 - prob\_F2G \end{bmatrix} \tag{4.4}$$

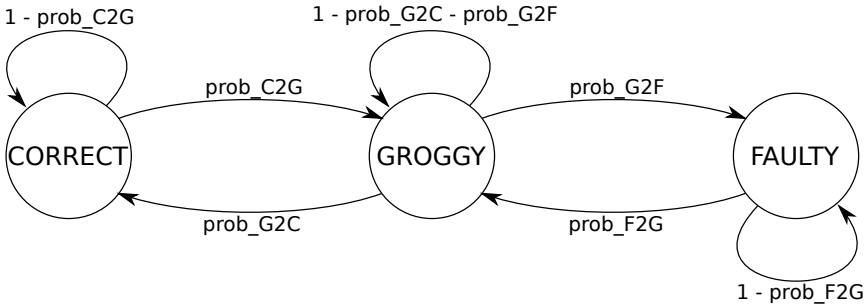


Figure 4.4: State machine of the Intermittent Fault Model.

While creating this state machine and the signal modification was straightforward, the implementation of the random number generation and passing the probability value contained certain challenges. The used IDE (QuestaSim) supports mixed simulations, so that the random number of the function `$urandom()` of *Systemverilog*, a successor of *Verilog*, was used because it returns an unsigned 32-bit number, which seemed to be a good approach to map the probabilities on it. However, this function, which will be triggered in the same cycle in all fault modules of the NoC, returns during an evaluation an identical value in all instances. In other words, it is unfit for such a simulation. In contrast the standard *Verilog* `$random()` function returns different random 32-bit signed values. This led to updating the design using this function. To cope with signed values, the probabilities were transformed into the positive range of it (0–2147483647), and the negative part was discarded for the state-change-evaluation. A transformation function is used to still be able to represent numbers after the decimal point (e.g. 0.002) for the fault probability, up to a given degree of precision, as integer values. First, the whole number is multiplied by 10^x, where x represents the number which is needed for the value to be free of digits after the decimal point. The maximal available positive integer value (2147483647) is then divided by 10^x and then multiplied by the transformed number, e.g. to represent a probability of 0.002 % the transform function would be $MaximalValue/Divider * Factor = 2147483647/1000 * 2 = 4294966$.

To improve the situation the technique to fuse multiple `$random()` values was evaluated. However, the text macros (`‘define MACRONAME VALUE`) which are used to pass the probabilities to the fault modules threw a warning in the IDE, that only 32-bit values are supported. This resulted in retaining the 32 bit signed random values and the fact that these values will be rounded is seen as an acceptable trade-off.

4.3 Fault Injection Script

The fault module of the previous section needs to be placed after every gate in the code of the NoC to run the fault simulations as displayed in Figure 4.2. However, the modules, which are used for the NoC[2] are described on the behavioral level for a more convenient development. To get a homogeneous file format from which the injection script can work, the *Design Compiler* [4] from *Synopsis* was used to synthesize these files to a *Gate Level* netlist. This required that the code style has tighter constrains[3] because in this step a netlist for actual hardware implementation

[2]Each module of the NoC corresponds to one *Verilog* text file.
[3]e.g. no open wires or infinite for-loops.

is created[4]. This step required using a proprietary gate library named *"tcbn65gplus.v"*, which contains all used gates and also defines its inputs and outputs as well as their logic behavior as can be seen for example in Listing 4.1. A further advantage is that this library also contains information about the direction of the ports of the gates, making it possible to attach the fault module in the correct way.

Listing 4.1: Example for gate library notation

```
module AND (IN1, IN2, OUT);
    input IN1, IN2;
    output OUT;
    and (OUT, IN1, IN2);
endmodule
```

This script itself is written in *Python* [6] (Version 3.5) and is executed from the command line. A schematic flow diagram can be seen in Figure 4.5.

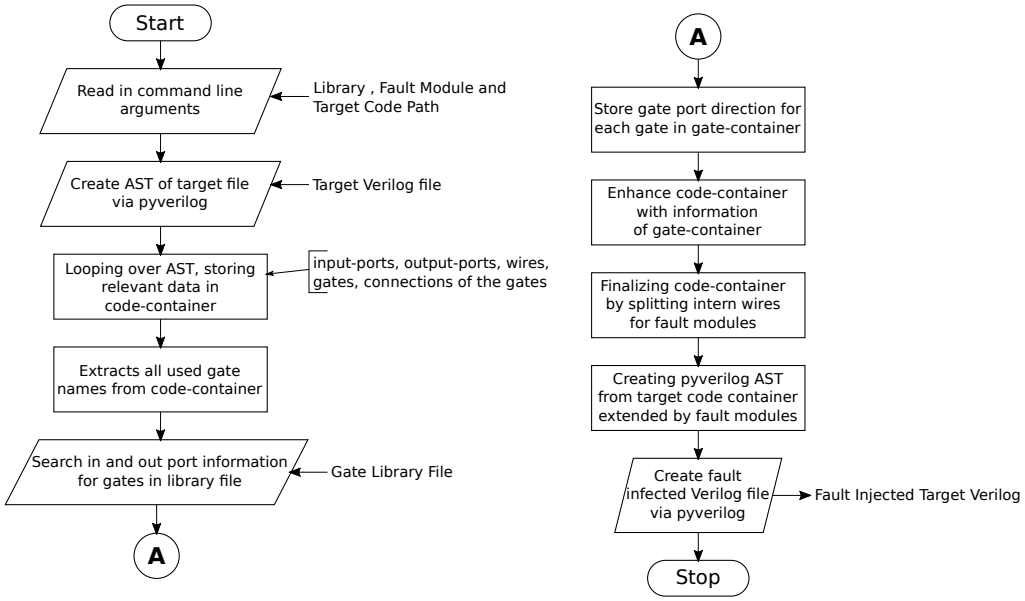


Figure 4.5: Simplified flow chart of the injection script.

The script starts by reading in the three mandatory arguments containing the path to the gate library, the fault injection module, and the target code module. In the next step, the target module gets transformed into an Abstract Syntax Tree (AST) by the parser from the framework *pyverilog* [7], which was used in version 1.0.6, and this script depends on it. *Pyverilog* itself depends on the open source HDL-compiler *Icarus Verilog* [8] for *Verilog* and VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL), which has to be installed as well.

The next part of the script reads out the relevant data from the AST by looping over it and storing the data (number and direction of ports, wires, gates, gate inputs/outputs) in an extra "container"-class. To increase the performance, a list of all used gates is looked up and extracted from the gate library instead of working with the whole library file. The extracted minimal information is again parsed by *pyverilog* to evaluate which port of the gate is an input and which

[4]Additionally, tests with tools from Synopsis *VCS* [5] were applied on the code for further timing and delay tests but this tool was not used any further than that.

an output. This information regarding the direction is then added to the stored gate-data in the container from the first parsing process. Afterwards the input port for the `faultyClock` is added to the topmodule and all the output signals of the gates are split into two parts, distinguishable by the postfix "_a" and "_b". This requires additional parsing steps for vectors (e.g. `wire [3:0] credit` as an example declaration in *Verilog*) to select the correct bit of it (e.g. `credit[2]` refers to a certain bit in *Verilog*). Furthermore, the input ports of the topmodules need to be excluded from the fault injection process because there is no gate, where a fault module could be added, before them.

In the followed action an AST is created by iterating over the container adding the appending data and also the fault modules in a specific format. In the final step the code generator of *pyverilog* is used to create the output file, named after the target file with the postfix "_faulty" appended to the name.
In the Listing 4.2 an example of the injection script command is shown. The `python3` command is necessary to execute the `faultInjectionScript`.

Listing 4.2: Example for commandline usage

```
$python3 faultInjectionScript.py tcbn65gplus.v m_FaultInjectionUnit.sv m_Crossbar.v
```

To present the result of the script a *DEMUX* module of the NoC is transformed as an example. In Listing 4.3 the code of the *DEMUX* after the transformation with *Synopsis Design Compiler* can be seen. Aside from the inputs and outputs, it has two wires (n3, n1) and four modules (U4, U5, U6, and U7) to execute its task.

Listing 4.3: Gate-Level source code **before** the fault injection

```
module m_Demux_1in2out_validIn ( select, data_in, data_out_0, data_out_1 );
  input [0:0] data_in;
  output [0:0] data_out_0;
  output [0:0] data_out_1;
  input select;
  wire   n3, n1;

  AN2D8 U4 ( .A1(select), .A2(data_in[0]), .Z(data_out_1[0]) );
  INVD0 U5 ( .I(n3), .ZN(n1) );
  CKND8 U6 ( .I(n1), .ZN(data_out_0[0]) );
  INR2D0 U7 ( .A1(data_in[0]), .B1(select), .ZN(n3) );
endmodule
```

After the transformation, presented in Listing 4.4, the input `faultClock` was added and both wires were split into an "_a" and "_b" part. Additionally, the two output ports each received an "_a"-post-fixed wire, which is connected to their fault module. The fault module is placed after the gates `U4` and `U6`, else they would be directly connected to the output without any chance to become faulty. The four modules have their signals adapted inside, and the four corresponding fault modules were added. The fault injected module can now be further used.

Listing 4.4: Gate-Level source code (formatted) **after** the fault injection

```
module m_Demux_1in2out_validIn_faulty ( data_out_0, data_out_1, data_in, select,
                                        faultClock );
  input [0:0] data_in;
  input select;
  output [0:0] data_out_0;
  output [0:0] data_out_1;
  input faultClock;
```

```
  wire [0:0] data_out_0_a;
  wire [0:0] data_out_1_a;
  wire n3_a;
  wire n3_b;
  wire n1_a;
  wire n1_b;

  AN2D8 U4 ( .A1(select), .A2(data_in[0]), .Z(data_out_1_a[0]) );
  INVD0 U5 ( .I(n3_b), .ZN(n1_a) );
  CKND8 U6 ( .I(n1_b), .ZN(data_out_0_a[0]) );
  INR2D0 U7 ( .A1(data_in[0]), .B1(select), .ZN(n3_a) );

  m_FaultInjectionUnit #( .MODULEID("data_out_0_0") )
  data_out_0_0_faulty ( .CLK(faultClock), .in(data_out_0_a[0]), .out(data_out_0[0]) );

  m_FaultInjectionUnit #( .MODULEID("data_out_1_0") )
  data_out_1_0_faulty ( .CLK(faultClock), .in(data_out_1_a[0]), .out(data_out_1[0]) );

  m_FaultInjectionUnit #( .MODULEID("n3") )
  n3_faulty ( .CLK(faultClock), .in(n3_a), .out(n3_b) );

  m_FaultInjectionUnit #( .MODULEID("n1") )
  n1_faulty ( .CLK(faultClock), .in(n1_a), .out(n1_b) );
endmodule
```

Due to its relevance to this thesis, it was decided to provide additional information about the *pyverilog* framework. It is developed by Shinya Takamaeda-Yamazaki and was first published in [TY15]. The goal of his work was to provide a toolkit to ease the development of new Computer Aided Design (CAD) tools for testing and improving *Verilog* code, which is in a broad sense also the intention of this thesis.
Pyverilog consists of five parts: A code parser, a data flow analyzer, a control-flow analyzer, a visualizer and a code generator. Both of the analyzers and the visualizer were not used.

The code parser has an integral part in the fault injection script to read the *Verilog* files (cf. Listing 4.5) and uses *Icarus Verilog* as preprocessor before its *pyverilog* code transfers it into an AST.

Listing 4.5: Envoking parsing process in Python script

```
fileList = []
fileList.append(VerilogFile)

ast, directives = parse(fileList, preprocess_include=include, preprocess_define=define)
```

The returning AST can be displayed with the command `ast.show`, of which an extraction can be seen in Listing 4.6.

Listing 4.6: Shortened output of ast.show

```
Source:  (at 8)
Description:  (at 8)
ModuleDef: m_Demux_1in2out_validIn (at 8)
Paramlist:  (at 0)
Portlist:  (at 8)
Port: select, None (at 8)
Port: data_in, None (at 8)
Port: data_out_0, None (at 8)
Port: data_out_1, None (at 8)
Decl:  (at 9)
Input: data_in, False (at 9)
Width:  (at 9)
```

```
IntConst: 0 (at 9)
IntConst: 0 (at 9)
Decl:  (at 10)
...
```

However, for evaluation, it is preferred to get into this tree structure itself, which differs from Listing 4.6 as can be seen in Figure 4.6 but has a more defined structure to iterate over it.

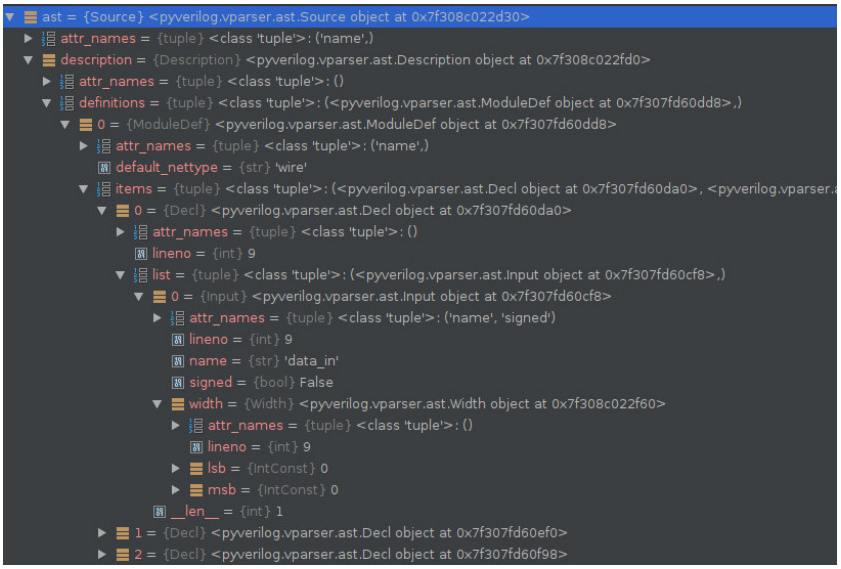


Figure 4.6: Overview of actual AST structure.

The code generator needs a self-created version of AST to write the *Verilog* files. This process demands that the values and names are appended in predefined format[5] to a list, which then will be collected in a final module definition, which is passed to the code generator. A simple working example can be seen in Listing 4.7, in which module "m_Simple" is created. The code shall show how to use *pyverilog*, but the created file has no actual function.
After passing its name to a variable, the ports are defined. The lists `ModulePortInput/ModulePortOut` represent the definition of the port inside the file, how wide it is and which direction it goes, while the list `portMembers` assures that the port is listed between the brackets of the topmodule. A similar procedure is executed for the wires. At this point, it should be noted that the example only covers singular ports, not vectors. Next, a module ("m_Counter") is initialized, followed by connecting its ports and concluding by adding its module to the "InstanceList". After the data it wrapped, it is passed to the `ASTCodeGenerator()` function which then needs to write and create a final file. This describes the code generation process which differs from the script, in which several nested loops, which iterated over the in- and outputs, wires, gates and fault modules with *if-conditions* were used to create the output.

Listing 4.7: Example script of how to create an AST and then pass it to the *Verilog* code generator

```
# Necessary variable initiation
topModuleName = "m_Simple"
emptyList = []
moduleList = []
itemMembers = []
```

[5]The preview of the function parameters of the IDE (Pycharm Community Edition [9]) was quite useful for this task.

```
portMembers = []
ModulePortInput = []
ModulePortOutput = []

# Module Input Ports
portMembers.append(vast.Port(name="CLK", type=None, width=None))
ModulePortInput.append(vast.Input(name="CLK", width=None, signed=False))

# Module Output Ports
portMembers.append(vast.Port(name="even", type=None, width=None))
ModulePortOutput.append(vast.Output(name="even", width=None, signed=False))

# Adding a module
moduleName = "m_Counter"
moduleType = "counter_logic"
modulePortList = []

argName = vast.Identifier(name="CLK")
portArg = vast.PortArg(argname=argName, portname="clk")
modulePortList.append(portArg) # Resulting in the Verilog Code -> .clk(CLK)

argName = vast.Identifier(name="even")
portArg = vast.PortArg(argname=argName, portname="output")
modulePortList.append(portArg)

# Instance Handling
moduleInstance = vast.Instance(module=moduleType, name=moduleName,
                portlist=modulePortList, parameterlist=emptyList,
                array=None)
moduleInstanceList = vast.InstanceList(module=moduleType, instances=[moduleInstance],
                     parameterlist=emptyList)
moduleList.append(moduleInstanceList)

# Preparing the code generation
# Ports
itemMembers.append(vast.Decl(list=ModulePortInput))
itemMembers.append(vast.Decl(list=ModulePortOutput))
# Module Declaration
itemMembers.extend(moduleList)

portList = vast.Portlist(portMembers)
ast = vast.ModuleDef(name=topModuleName, paramlist=None, portlist=portList, items=itemMembers)

# Executing the code generation
codegen = ASTCodeGenerator()
result = codegen.visit(ast)

# Writing code to file
content.outputFile = open((topModuleName + ".v"), 'w')
outputFile.write(result)
outputFile.close()
```

5 Simulation & Analysis

In this section, the simulations on the fault injected target Network on Chip (NoC) will be evaluated and explained. First, the evaluation of the target NoC followed by a mathematical approach of calculating the fault probability of the system are shown. Afterwards, the results of the simulations will be presented, the different configurations compared and their behavior analyzed.

5.1 Evaluation of the Target Network on Chip

The evaluation of the target system starts after the transformation of the code modules into `gate-level` and the fault injection process on these modules is finished. With all its different configurations the NoC code includes 27 modules, which ware representing the behavioral elements of the system. Within the 27 modules the needed width and port adaptions are also counted. In Figure 5.1 their connections inside one of the 64 routers of the target NoC are shown. Each router consists of seven *Channels*, a *Crossbar*, a *Switch Allocation*, five *Output Buffers*, two data *Demuxs* and two regular *Muxs* for the Virtual Channel Allocation (VA) as well as two *Round Robin Units* for their allocations. The *Channels* consist of a *Buffer*, a *State Machine* and a *Routing Calculation*. Table 5.1 lists their transformation results. The first column contains the name of the module, the second one the number of gates and the third one the number of added fault modules. In most cases, the number of gates is equal to the number of added fault modules, but it differs for some because certain gates have multiple output ports (normal and inverted output ports). Following its logic, the script adds a fault module to them, thus creating this increased count.

For the simulations, different combinations of active fault tolerance methods are used, which also affect the NoC. They can be divided into five basic configurations, which are described in the following lines. For the bare unextended structure, therefore called *blank* (I), no additional elements are activated. Activating the ECC fault tolerance method (II) adds an *Encoder* and a *Decoder* at the Network Interface (NI) of the NoC as well as an *Intercoder* into every *Channel*. Inside the network the logic will not be modified but the whole data path needs to be widened for the enlarged FLow control unIT (FLIT) with its ECC checksum. The retransmissions (III) do not modify the router but instead the testbench receives additional logic to enable the retransmission and Acknowledgement (ACK)-logic. The Fault Tolerant Routing (IV) extends the system by several assertion units and error counters, which are placed in the *channels* and the router itself.

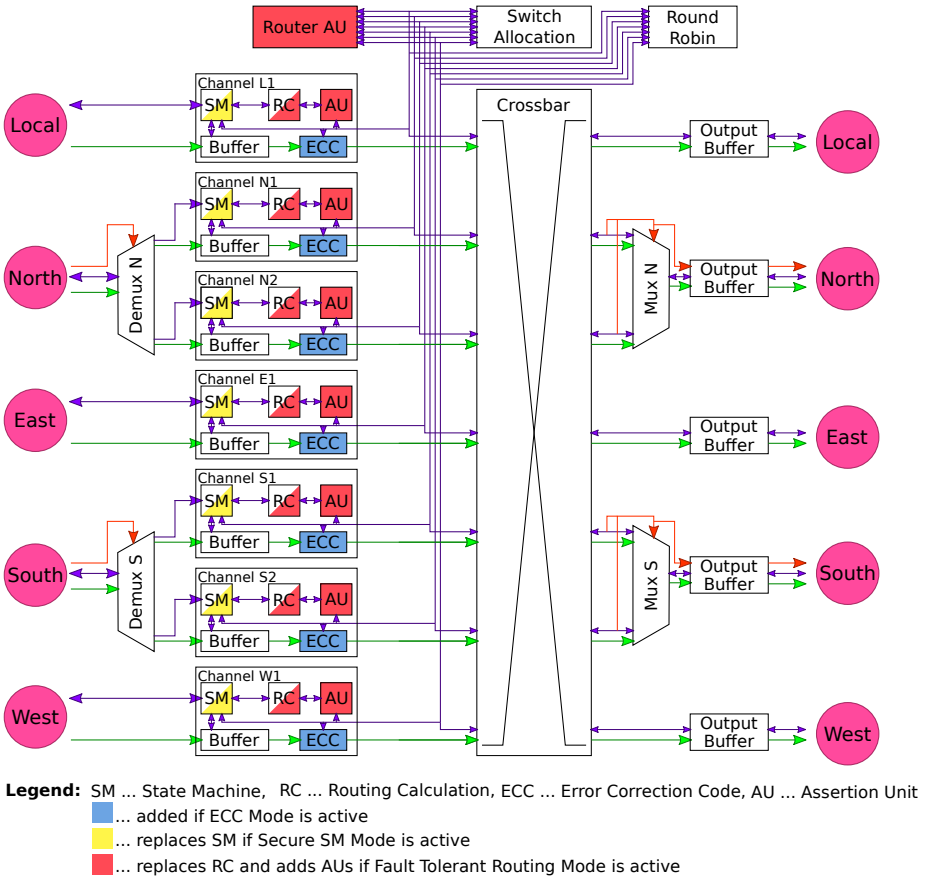


Figure 5.1: Overview of the target NoC with color coding for fault tolerance methods.

It also replaces the *Routing Calculation* with the fault tolerant version. Last but not least, for the *Secure State Machine* tolerance method (V) the *State Machine* in every *Channel* is updated. These five configurations were selected for a fault probability calculation of the network (see Section 5.2).

Table 5.2 gives an overview over the used modules in one router for each of these five basic configurations described above. The information of the number of gates and fault modules is retrieved from the source code of the target NoC. Due to the design of the router the configurations have a similar usage of the modules (e.g. four of five configuration use the normal buffer), but differ in detail. If combinations of fault tolerance methods would be selected, their module usage will be combined.

Table 5.3 presents for each of the basic five configurations the different amounts of gates for only one router (Gates/Router) and the amount of fault modules for each configuration for one router (Fault Modules/Router). For example if you take the number of modules from the column of ECC from Table 5.2 and multiply them by their according number of gates in Table 5.1 your result equals 21000 Gates/Router, which is the corresponding entry in Table 5.3. Furthermore, for the amount of gates for the whole NoC (Gates/NoC) the values of Gates/Router of Table 5.3 are multiplied by 64. The amount of fault modules for the whole NoC (Fault Modules/NoC) is calculated in the same way, except that the column of the fault modules of Table 5.1 is used. As already explained in Section 3.4 the changes for the retransmission fault tolerance are external to the NoC, therefore resulting in the same gate/fault module count as the *blank* configuration.

Table 5.1: Number of gates and fault modules in the used NoC modules.

Module	Gates	Fault Modules
AssertionUnit Crossbar	508	508
AssertionUnit Packet	64	64
AssertionUnit Routing Calculation (RC)	13	13
AssertionUnit Round Robin	21	21
AssertionUnit Switch Allocation	269	283
AssertionUnit State Machine	37	37
Buffer	913	919
Buffer wide[1]	1577	1583
Crossbar	1737	1737
Crossbar wide	2837	2837
Demux 1in2out dataIn	230	230
Demux 1in2out dataIn wide	413	413
Demux 1in2out validIn	4	4
D FF sync channel	4	4
D FF sync data	109	109
D FF sync data wide	193	193
ECC Unit Inter	305	305
Error Counter Channel	133	164
Error Counter Router	145	176
Mux 2in1out	73	73
Mux 2in1out wide	130	130
Routing Calculation	111	111
Routing Calculation fault tolerant	245	248
Round Robin EvaluationUnit	15	15
State Machine	134	136
State Machine secure	203	203
Switch Allocation	1165	1165
Switch Allocation extended[2]	1196	1196

[1] wide ... represents modules which have a wider data port for the additional checksum for Error Correction Code (ECC).
[2] extended ... exposes additional internal signals for the assertion unit used by the Fault Tolerant Routing.

Table 5.2: Number of used modules in a router for the five basic fault tolerance configurations.

Module	blank	ECC	Retransm.	FT Routing	Secure SM
AssertionUnit Crossbar	0	0	0	1	0
AssertionUnit Packet	0	0	0	7	0
AssertionUnit RC[1]	0	0	0	0	0
AssertionUnit Round Robin	0	0	0	2	0
AssertionUnit Switch Allocation	0	0	0	1	0
AssertionUnit State Machine	0	0	0	7	0
Buffer	7	0	7	7	7
Buffer wide	0	7	0	0	0
Crossbar	1	0	1	1	1
Crossbar wide	0	1	0	0	0
Demux 1in2out dataIn	2	0	2	2	2
Demux 1in2out dataIn wide	0	2	0	0	0
Demux 1in2out validIn	2	2	2	2	2
D FF sync channel	5	5	5	5	5
D FF sync data	5	0	5	5	5
D FF sync data wide	0	5	0	0	0
ECC Unit Inter	0	7	0	0	0
Error Counter Channel	0	0	0	7	0
Error Counter Router	0	0	0	1	0
Mux 2in1out	2	0	2	2	2
Mux 2in1out wide	0	2	0	0	0
RC	7	7	7	0	7
RC Fault Tolerant	0	0	0	7	0
Round Robin EvaluationUnit	2	2	2	2	2
State Machine	7	7	7	7	0
State Machine Secure	0	0	0	0	7
Switch Allocation	1	1	1	0	1
Switch Allocation extended	0	0	0	1	0

Table 5.3: Summed up number of gates and fault modules for different configurations of the whole NoC.

	blank	ECC	Retransmission	FT Routing	Secure SM
Gates/Router	12217	21000	12217	15788	12700
Gates/NoC	781888	1344000	781888	1010432	812800
Fault Modules/Router	12273	21056	12273	16127	12742
Fault Modules/NoC	785472	1347584	785472	1032128	815488

5.2 Mathematical Approach

The information of the previous chapter allows calculating the fault probability of the whole NoC for its next cycle under some assumptions. First, the NoC starts in a fault-free state and every fault module has the same probability to switch from the $CORRECT$ to the $FAULTY$ state. In this thesis faults are considered to be faults without any regard if the fault would be masked or not.
For the calculation, a simple stochastic approach for independent events is used. As an example two fault modules, A and B, with the same fault probability $probC2F$ (i.e. probability to switch from $CORRECT$ to $FAULTY$) have three possible combinations of resulting in an error. Case one and two are that either only A or only B change their state, which can be mathematically described by $Case1 = Case2 = probC2F * (1 - probC2F)$. Or case three where both get faulty, mathematically described by form $Case3 = probC2F * probC2F$. Summarizing these three equations results in the fault probability of this example. However, for this large number of fault modules of the target system it is easier to calculate the chance that the system stays correct and subtract the result from 1 to get the fault probability. Due to the fact that all probabilities to become faulty are the same the calculation can be reduced to $1 - (1 - probC2F)^n$, where n is the number of fault modules in the system. Applied on the target NoC the results are displayed in Table 5.4 (truncated to two decimal places). The result is under the assumption that the fault probability value ($probC2F$) is $1.0e - 7$. This value was chosen because several test runs with transient faults on the target NoC showed that values above $1.0e - 7$ had an impact which was too strong on the performance. $1.0e - 7$ was the first value that did not break the system and allowed valid simulations.

Table 5.4: Fault probability of different configurations of the whole NoC.

	blank	**ECC**	**Retransmission**	**FT Routing**	**Secure SM**
Fault Modules/NoC	785472	1347584	785472	1032128	815488
Fault Probability [%]/NoC	7.55	12.61	7.55	9.81	7.83

The range of the fault probability for each of the five basic configurations spans from 7.55 % up to 12.61 %, which states that it takes stochastically about ten cycles to transfer the system into a faulty scenario. Furthermore, it can be implied that a NoC with more elements has a higher chance to get faulty, reducing the reliability of the whole system. Under this assumption the ECC in Table 5.4 would have the highest probability to become faulty, concluding that additional combinations of ECC with other configurations would add to this negative effect as the overall number of elements would be increased. However, the impact of the additional fault modules appearing in the ECC would mostly affect the data path by corrupting a bit in a FLIT, which possibly will not have any further negative effect on the packet. On the other hand faults in the fault tolerant routing calculation or the secure state machine introduce more possibilities to extend their erroneous behavior to the rest of the router, which can possibly have a stronger influence on the system. However, this theoretical approach will not spare evaluating the practical efficiency of the fault tolerance methods. Therefore, it is necessary to get closer to the actual implementation simulations.

5.3 Simulation Setup

For the simulations, the fault infected NoC is inserted in a testbench, which is equipped with several evaluation elements to calculate the reliability and latency of the packets. A highlighted feature is that every packet receives an unique ID during its transmission, which is used for the tracking if the packet is injected or ejected from the network and it is reset after successfully reaching its destination.
For the reliability the number of returned IDs, which equals the successfully ejected packets, is divided by the number of injected packets. If the retransmission technique is active this tracking is modified by counting a packet as injected at the moment its ID is occupied by the sender and as ejected when the ID is returned from the same sender. This is to counteract against the distortion of the additional number of retransmitted packets. This result is named *relative* reliability (cf. Equation 5.1) in this thesis.

$$Reliability_{relative} = \frac{Packets_{ejected}}{Packets_{injected}} \tag{5.1}$$

The *absolute* reliability is another metric which is the result of the ejected packets divided by the absolute number of to-be-injected packets, which for this simulation will be 2000 packets (cf. Equation 5.2). This calculation includes under specific circumstances information about the congestion in the network by including the not injected packets, which were prevented from entering the network due to the formerly mentioned congestion. The limitation of this measurement is that it focuses only on heavy congestion, if it is not heavy it will not indicate significant changes. In other words if the injected packets equal the 2000 limit, this graph will not provide further information. But it allows a comparison with other fault tolerance methods, which have not reached this limit of 2000 packets.

$$Reliability_{absolute} = \frac{Packets_{ejected}}{Packets_{toBeInjected}} = \frac{Packets_{ejected}}{2000} \tag{5.2}$$

For the latency calculation, a counter variable for every occupied ID is incremented, which is reset when the packet (with this ID) is injected into the network. When it is ejected the number of passed cycles is added to a summation variable, which is divided by the number of ejected packets (cf. Equation 5.3).

$$Latency = \frac{TraveldurationVariable_{Packets}}{Packets_{ejected}} \tag{5.3}$$

An active retransmission technique changes the setup, while the increment and reset of the cycle counter processes are the same. Also, the ACK-packets latency is added to the summation variable, which is then divided by the sum of the normal and ACK-packets (cf. Equation 5.4).

$$Latency_{Retransmission} = \frac{TraveldurationVariable_{Packets+ACKs}}{Packets_{ejected} + ACK_{ejected}} \tag{5.4}$$

Pretests showed that a simulation runtime of 20000 cycles while injecting up to 2000 packets consisting out of two to seven FLITs was seen as the most promising test scenario for relevant results. The packet injection chance per NI and cycle was set to 1%, which provided a usable duration for the simulation runs.
The packet injection and the fault trigger functions were both based on random number generation, which will produce the exact same simulation result for one seed in *Verilog*. To achieve an averaged result, simulations for each combination of fault pattern (transient, intermittent, permanent) and fault tolerance methods are done with different seeds (1, 1000, 10000). The fault probability during these simulation runs is swept from 0 up to 0.0000010 in incremental steps of 0.0000001.
Therefore, for each simulation configuration (which consists of active fault tolerance methods + fault pattern + seed) a block of 11 simulation runs is performed. One simulation run took from four up to eleven hours (some even more), while a complete simulation block consisting of 11 runs was finished in about five days. Each block needs to run three times due to the three different seeds. To increase the throughput such a block of simulations[2] would be executed on one of 11 provided computers, eight of them were Virtual Machines (VMs) and three real hardware systems.
For switching between the fault tolerance methods, *macros* are used in the parameter file, which also allow configuring the fault probability as well as the fault pattern, but the failure type (i.e. *Stuck-At-0/1*, *Bit-Flip*) needs to be configured in the fault module itself. Incrementing the fault probability, starting and stopping the simulation and backing up the data is automated by a simulation script written in Python using *Questasim* in command line mode.

As noted above all fault patterns for the fault modules have the same probabilities of leaving the *CORRECT* state, but all the other probabilities differ to resemble their appearance pattern. The transient fault (see Markov matrix in Equation 5.5), has a high chance to recover (90%) to resemble its short appearance, whereas the permanent fault, see Equation 5.6, has a 0 % chance to leave the *FAULTY* state, slowly aggregating faults in the system.

$$P_{transient} = \begin{bmatrix} 0.9999999 & 0.90 \\ 0.0000001 & 0.10 \end{bmatrix} \tag{5.5}$$

$$P_{permanent} = \begin{bmatrix} 0.9999999 & 0 \\ 0.0000001 & 1 \end{bmatrix} \tag{5.6}$$

However, the intermittent fault (see Equation 5.7) is more refined. It has a fixated chance of 6.25% to recover from its *GROGGY* state in which it has a 50% chance to become *FAULTY* but also recover quickly back to the *GROGGY* state (50%) to resemble its fluctuating behavior[3].

$$P_{intermittent} = \begin{bmatrix} 0.9999999 & 0.0625 & 0 \\ 0.0000001 & 0.4375 & 0.5 \\ 0 & 0.5 & 0.5 \end{bmatrix} \tag{5.7}$$

[2] In total 1089 simulations were executed, without counting re-runs and test-runs.
[3] Pre-Tests and [WEH+] were considered for these values.

Next, the different fault tolerance methods configuration will be discussed. As already mentioned in Chapter 3.4 the fault tolerance methods implemented are ECC, End to end (E2E) retransmission, fault tolerant routing and an improved version of the state machine.

In Table 5.5 the different configurations are presented, displaying which tolerance methods are active in it. Additionally, every combination has an abbreviation based on its configuration as a codename for a faster recognition, which will be used during the simulation and analysis. The only abstractly named one is *blank*, which resembles a NoC without any active fault tolerance methods, while the others are self-explanatory. After the *blank* configuration each fault tolerance method is evaluated on its own before being activated all together to test the combined fault tolerance performance. The last three entries cover configurations, which are supposed to have synergies between each other. First, *ssm-ecc* enables the secure state machine to check the packet (especially the *headflit*) for faults. Second, the added assertion units of the fault tolerant routing calculation enable the secure state machine to wait if a fault appears in *ssm-rcTol*, which might possibly recover on its own. Third, *ssm-ecc-rcTol* combines both of these features.

Table 5.5: Overview of the simulated fault tolerance methods configurations.

Abbreviation	ECC	Retransmission	FT Routing Calculation	Secure SM
blank				
ecc	x			
retr		x		
rcTol			x	
ssm				x
retr-ecc-ssm-rcTol	x	x	x	x
ssm-ecc	x			x
ssm-rcTol			x	x
ssm-ecc-rcTol	x		x	x

As the failure type, the *Bit-Flip* was selected because it has the most hostile effect on the system. It will be discussed further in the Subsection 5.4.5 because it should be to be considered in context of the simulation results. This selection is not as common as *Stuck-At* faults are for actual implementations, but the goal is to push the NoC to its limit. By doing so, the importance of evaluating fault tolerance should be emphasized aside from performance and power consumption.

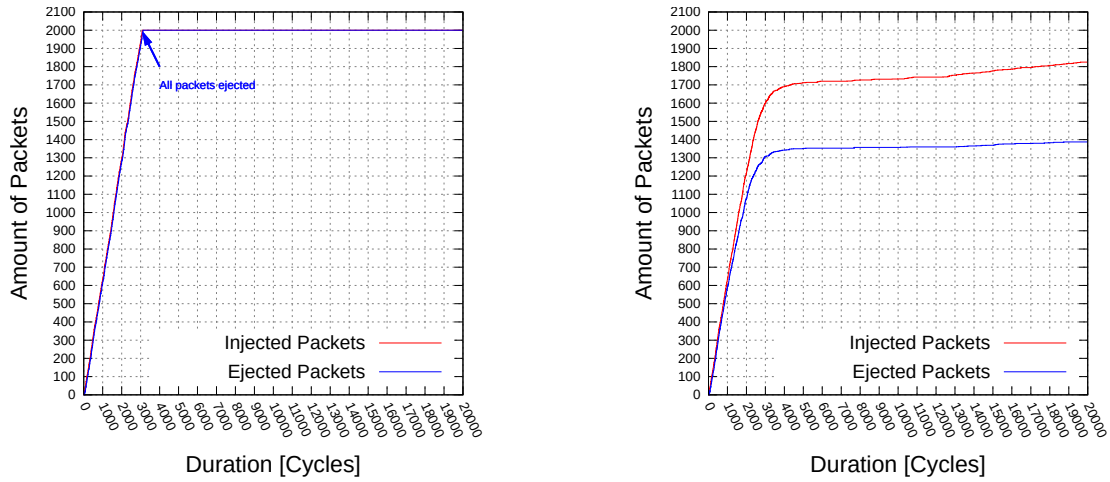
5.4 Simulation Results

In this section, the simulation results are evaluated starting with a NoC configuration without any active fault tolerance methods. Afterwards, the performance results of the different fault appearance patterns with several fault tolerance configurations are evaluated.

5.4.1 NoC with no active Fault Tolerance Methods

As mentioned in the previous section, up to 2000 packets will be injected into the system to measure the reliability and latency of the NoC. At first, the ideal case will be covered, presenting the results when no fault modules are active (i.e. fault probability is 0). In this scenario it takes the *blank* NoC around 3000 cycles to pass the packets through the network. As expected the

reliability is at 100 %, which can be seen in Figure 5.2a. Even when the simulation runs for 20000 cycles (x-axis) all injected packets leave the network shortly after the 3000 cycles mark and not one packet was lost. It may seem that the lines overlap, but this illusion appears due to the narrow gap between the injected and ejected packet line and the in comparison long range of the x-axis. The average duration a packet stays in the network is 32 cycles as displayed in Figure 5.3a.



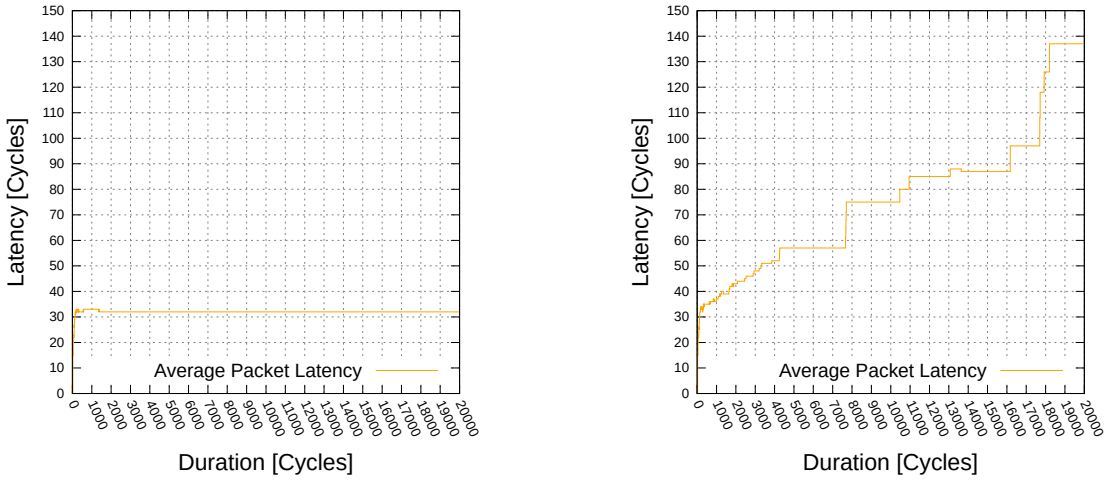
(a) fault free NoC. **(b)** NoC with active fault modules.

Figure 5.2: Example of a fault free vs. faulty NoC's simulation run.

However, with active fault modules (i.e. fault probability > 0) and the associated faults the reliability shrinks and the latency increases. This happens due to modified signals, which are caused by the fault modules. These modified signals lead to errors in the control signals and the transferred data. But the data corruption in *bodyflits* or *tailflits* will not be issued in this thesis because it is assumed to be handled by the Open Systems Interconnection (OSI)-layers above. Still, corruption in the *headflits* will have an impact on the system.

In the following faulty example (cf. Figure 5.2b) the network is affected by transient faults with a fault probability of 0.0000004 and the seed *10000* is used. No fault tolerance methods are active in this example. This is a selected run of the simulation results to explain the process of the evaluation. Within the first 1000 cycles the simulation is similar to the fault-free run from before, but then the curves of the injected and ejected packets start to part ways. The simulation environment is not able to inject all 2000 packets into the network during the preset duration, but the number of injected packets is still increasing so the possibility exists that if the simulation runs further, it could have reached the limit (i.e. 2000 packets). Nevertheless, the behavior compared to the fault-free run can be explained by a clogged NoC. Multiple reasons exist for this behavior, the transfer signal or the credit feedback could be faulty as well as one or multiple deadlocks which created themselves during the simulation by stuck or misrouted packets. If this is the case, the network would not be able to recover, explaining why the amount of ejected packets is so low aside from the possibility that single packets could have been lost.

For the analysis of this particular simulation run the wave data (cf. Figure 5.4) was inspected and it turned out that there was partly any interaction after an intensive phase at the beginning of the simulation until the 5000 cycles mark (1 cycle equals 2 us in the simulation). The *credits_feedback*



(a) Average latency of fault free NoC. **(b)** Average latency with active fault models.

Figure 5.3: Fault free vs. faulty NoC latency.

signals have a low count (maximum value is 8), which suggests that their connected buffers are not empty. Still, there is no further activity on the *out_data* lines which points towards a deadlock situation, where no further data transfer would be possible. This confirms the assumption above.

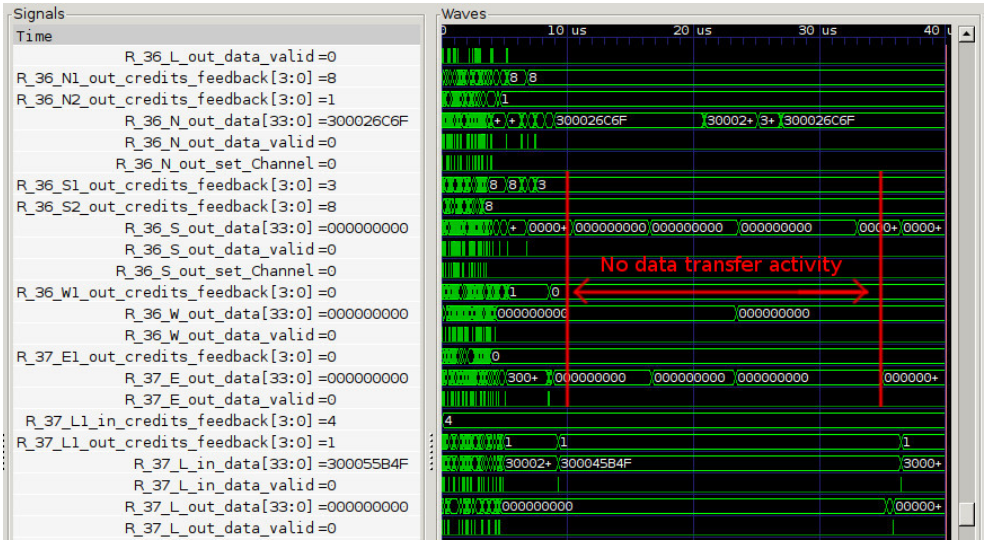


Figure 5.4: Snippet of wave data results of the faulty NoC simulation.

In this particular faulty run[4], the number of the injected packets was 1825, and 1387 packets were ejected. Furthermore, in this thesis, a not injected packet is not viewed as a lost packet, so the *relative* reliability will be calculated by the percentage of successfully transferred packets, which in this case would be $1386/1825 = 0.7595 = 75.95\%$. Still, for the evaluation of this thesis the total amount of "to be injected packets" will also be considered because they are counted for the calculation of the *absolute* reliability.

[4]As a reminder no fault tolerance methods are active, the simulation seed is *10000*, the fault probability is 0.0000004 for each fault module, the error pattern resembles transient faults and logic errors are displayed as *Bit-Flip* error.

Further test runs with the same configuration but different seeds (*1* and *1000*) showed that these simulations can have a high level fluctuation in their results. As it is presented in Table 5.6, the simulation with the seed *10000* has worse reliability than the seeds *1* and *1000*, which resulted into using the average of the three seeds, which is 86.94%. For a better interpretation of the reliability results, the average will be used in the remaining analysis.

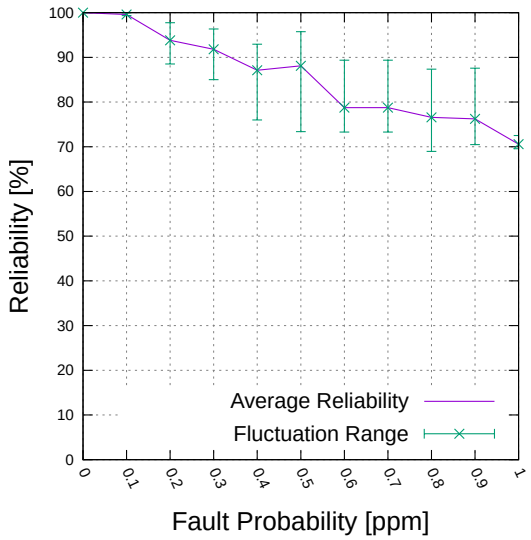
Table 5.6: Simulation results of different seeds presenting fluctuation.

Seed	Relative Reliability [%]	Latency [cycles]
1	92.48	87
1000	92.38	86
10000	75.95	133
Average	86.94	102

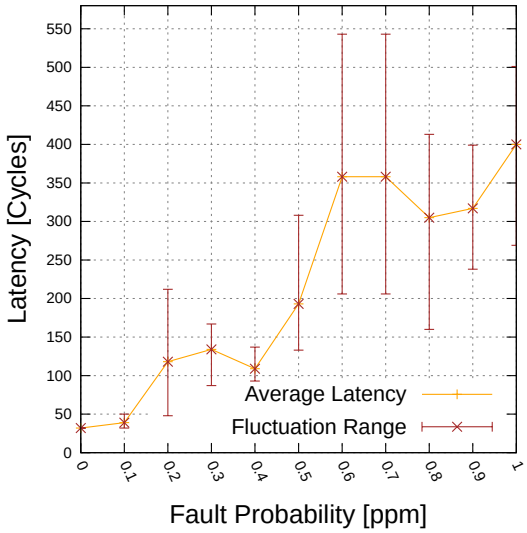
Further analyzing these test simulation runs, it becomes clear that (compared to the fault free run) also the latency, the other evaluation parameter, is affected by the increased fault probability (of 0.0000004) (cf. Figure 5.3b). The results for the latency of all three different seeds can be viewed in Table 5.6. For the latency the mean transmission duration is 102. Thus, comparing all the seeds, the seed of *10000* is again the worst performer.

For a better comparison of this NoC configuration, overview graphs were created, which describe the behavior of the network during the fault probability sweep. First, in Figure 5.5a the average relative reliability is presented, notably decreasing during the increasing fault probability because packets are lost, corrupted or malfunctions appear in the control logic. The second Figure 5.5b resembles the latency of this situation, which is growing due to congestion in the system or the router not being able to allocate the necessary resources because of faults in the system. Both figures make use of error bars, which display the minimum and maximum results of the simulations of the different seeds.

Unfortunately, the graphs display a high level of fluctuation. Factors like the traffic level, different fault positions in the modules and deadlocks create a complex system, which influences the performance of the NoC. For example in Figure 5.5a the reliability of the different seeds fluctuates at the fault probability of 0.9 ppm by about 10 %, emphasizing the sensitivity of such systems. This variation in the results leads to the consideration that it is necessary to calculate the average of many simulation runs to draw firm conclusions from it because each simulation could have a different effect on the averaged results. It can be argued that the majority of the simulation results will concentrate on a certain fluctuation-region, which becomes more defined (i.e. narrower) with the increasing number of simulation runs. Values far outside this region can be seen as a stochastic singular event and could be excluded from the results. For simulations with a limited number of runs, “drops” and “peaks” can affect the average calculation much stronger, therefore giving the impression that certain fault tolerance methods are more beneficial than others by displaying different tendencies.
However, the number of simulations for this work was greatly influenced by the target NoC’s performance. The target NoC is focused on easily switching between different fault tolerance methods and was designed by a novice of NoCs. Several pre-tests showed that the amount of packets and the duration of simulations ended in an acceptable result and so the performance was seen as sufficient in the beginning. But faulty simulations consume much more time, which



(a) Averaged Reliability - Configuration *blank*.



(b) Averaged Latency - Configuration *blank*.

Figure 5.5: Transient fault simulation - Configuration *blank*.

is further increased by runs using different seeds. Therefore, it was necessary to make a compromise. It was decided to limit the simulation runs to three different seeds for each configuration. Additional simulation machines were used to help with the time consuming runs. So the length of simulations of this work is shorter compared to other works which also inject more packets during their simulation runs, which subsequently reduces the significance of each lost or delayed packet and should improve the accuracy of the fluctuation regions.

Taking this into consideration, for this thesis the averaged line for the results in the following graphs must be seen as a reference value, which cannot be used to argue with 100% confidence which fault tolerance method is the best but it allows to recognize that specific fault tolerance methods tend to perform better than others and that some fault tolerance methods have a negative impact on the system. Simulations with three different seeds can therefore indicate some tendencies in the performance of fault tolerance methods.
The graphs in the next sections of the thesis are displaying the mean values without error-bars to avoid visual clutter.

5.4.2 Transient Fault

First, the behavior of the transient fault will be evaluated. Compared to the other chosen faults it has the distinction of a short appearance duration. The tested configurations and their codenames can be viewed in Table 5.5. The results of the 297 simulation runs were used to calculate the mean values for each configuration block (33 for each of the nine configurations e.g. *blank* or *ecc*).

For the relative reliability, Figure 5.6 shows how the increasing fault probability reduces the ability to safely pass packets through the network whereas in Figure 5.7 the absolute reliability presents that in perspective of the goal of injecting 2000 packets, the performance difference between each configuration further increases and the overall performance tends to be worse. The mean values of the absolute reliability (Figure 5.7) show a worse performance compared to the relative reliability for transient faults (Figure 5.6). Especially the configurations *ssm-rcTol*, *retr-ecc-ssm-rcTol*, *ssm* and *retr* are affected and show less reliability.

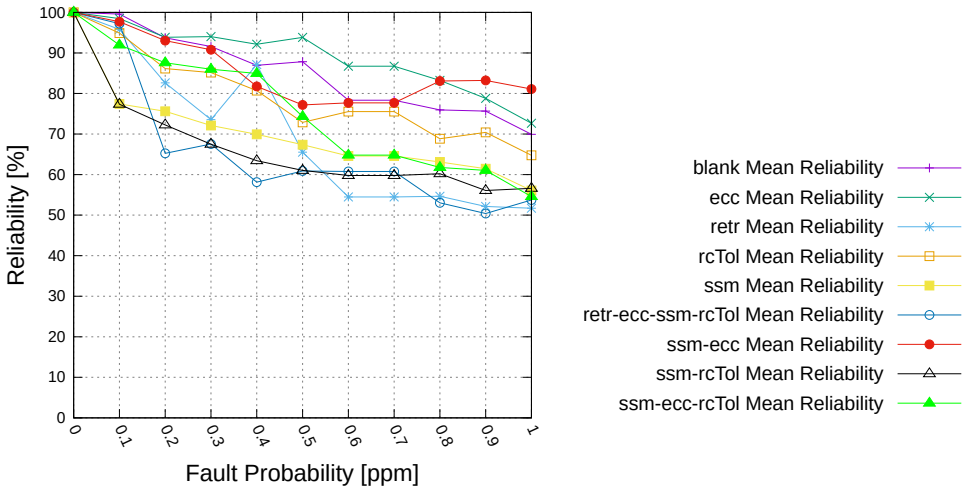


Figure 5.6: Relative reliability for transient faults.

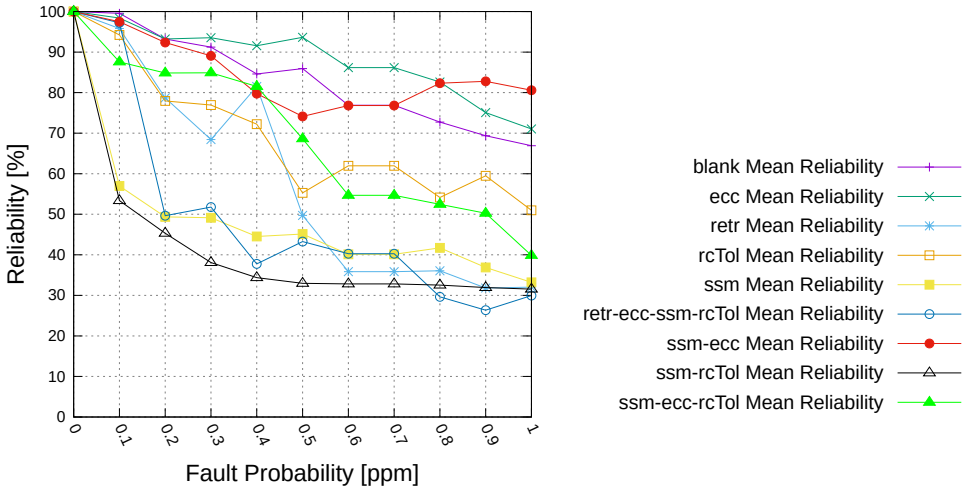


Figure 5.7: Absolute reliability for transient faults.

The three best performing fault tolerance configurations are *blank*, *ecc* and *ssm-ecc* but due to the

overlapping graphs and the fluctuation (simulations could change their order) it is not possible to definitely say which one is leading. *Ssm-ecc-rcTol* is also good at the beginning but has a stronger downward trend later in the graph.
For the *blank* setup it can be assumed that with no additional elements, there is a lower overall chance that an additional critical fault appears (cf. Table 5.4). With increasing fault probability, tolerance methods become necessary, represented by the increasing decline of the graph.
Ecc, a proven technology for encoding, presents its good capabilities against transient faults, even better than *blank* in the relative and absolute reliability graph. This technique is part of the data path of the network, so faults which affect the packets can be fixed thus allowing a further transmission, without adding additional traffic to the system by resending packet data. However, the dropping of faulty packets by the state machine helps the system to run more fluently in the beginning but if the fault probability increases, possibly due to more dropped packets, the reliability goes down.
In combination with the secure state machine (*ssm-ecc*) the results show a different trend where the reliability appears better towards the end of the graph. This suggests that the secure state machine's ability to wait for a predefined duration for the recovery of packets, which have been marked as faulty by the ECC instead of dropping them, is a crucial feature to keep a NoC running. This synergy surprises in regard to the weak performance of *ssm* on its own. While *ssm* alone is also able to wait for the recovery of faults caused by the routing calculation or able to react to an incorrect appearing *headflit*, the critical function of taking the packets health into account is missing. Evaluating the interaction between the routers inside the network showed that there is still activity between the routers suggesting that no deadlock situation happened. Therefore, the poor reliability of the *ssm* configuration can be explained by the lack of evaluating the packets and the possibility of dropping them. This is an excellent example of the importance of this thesis' fault tolerance evaluation. It also points out what the method needs to additionally cope with transient faults.
The fault tolerant routing calculation (*rcTol*) presents itself in the middle of the graphs of relative and absolute reliability for transient faults (Figure 5.6 and 5.7), but in combinations with other methods (*ssm-rcTol*, *ssm-ecc-rcTol*), it appears to have a negative impact on the reliability. One explanation could be that the faulty routers and channels will be abandoned by this method because it does not see a possibility for them to recover. This behavior is optimal for long-lasting faults, where a packet should strictly avoid these elements but for short duration faults it makes resources useless, which could be used again after the fault disappeared.
The retransmission method (*retr*) has the lowest average reliability of the single active fault tolerant methods. In other combinations it also displays a negative effect on the reliability, for example when *ssm-ecc-rcTol* is compared with *retr-ecc-ssm-rcTol*, which can be seen in Figure 5.7. Considering the sensitivity of NoCs against congestions, the additional retransmitted ACK-packets and the resending of packets cause problems for the network, resulting in the drop of reliability. This activity between the routers has been detected as quite high. This indicates that a longer retransmission window could be of advantage for future works and shows how this thesis' evaluation can be used for optimization.
From the results above it can be expected that a combination of all available fault tolerance methods (*retr-ecc-ssm-rcTol*) will not result in the best performance and this is also seen as this configuration ranks under the lowest reliability performers in Figures 5.6 and 5.7. This solidifies that simply adding methods, with the thought that more protection is better, will not necessarily improve the reliability.
The combinations *ssm-ecc-rcTol* and *ssm-rcTol* should further enhance the secure state machine through assertion units of the fault tolerant routing calculation to provide further protection

against faults, aside from the improved routing technique. However, the results lead to the conclusion that *ecc* is the key feature for reliability and the assertion units are actually counter-productive. Without it, the reliability can be increased by 10% (see relative graph Figure 5.6).

After evaluating the reliability, it is important to analyze the latency as it is another essential characteristic of the performance of a NoC. The increasing fault probability causes congestions or temporarily blocks a path, which the packet needs to reach its destination. Thus, it is affecting the transfer duration of packets. Due to the system's dependency on the congestion level inside the router and due to the potential appearance of a deadlock, the results can have a high level of fluctuation as it can be seen in Figure 5.8. The different configurations are crossing each other often, making it difficult to attach a specific behavior to them, so the decision was made to make use of linear regression as it can be seen in Figure 5.9. This is no replacement for the original graph but more of a support for recognizing the tendency of each fault tolerance method.

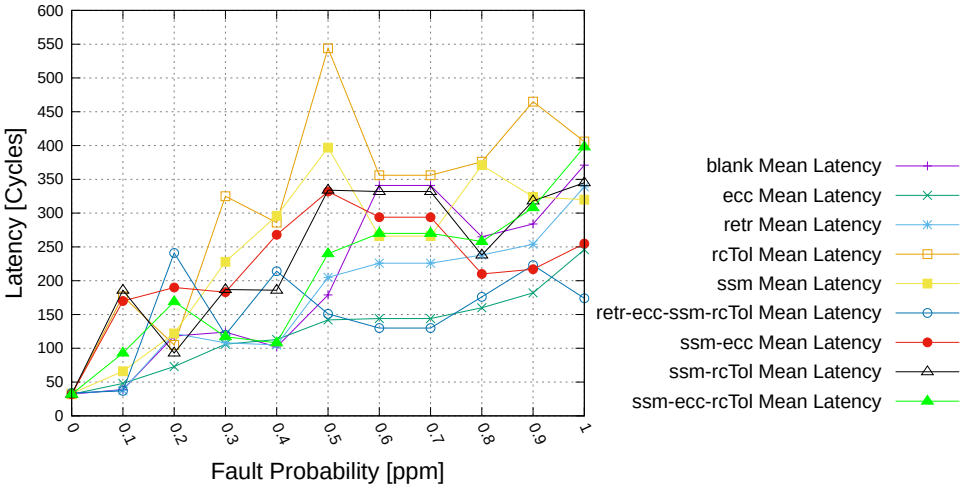


Figure 5.8: Latency result for transient faults.

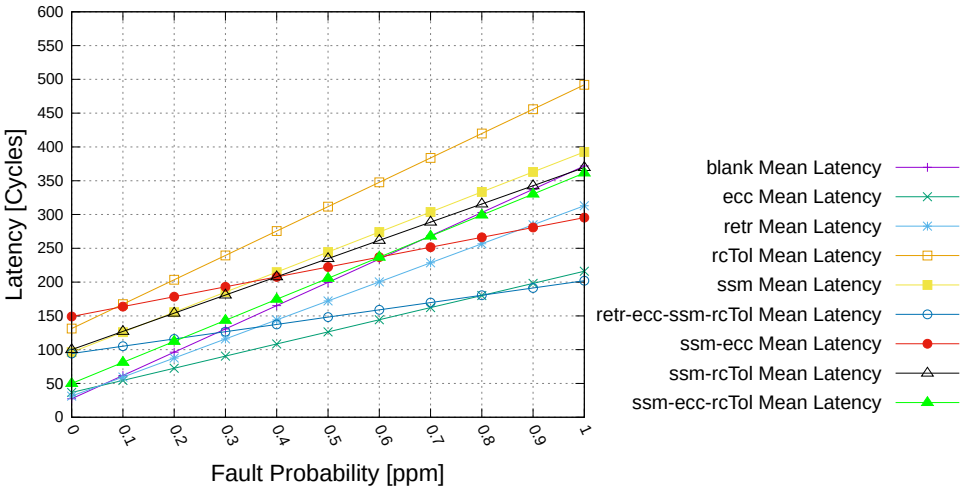


Figure 5.9: Latency results for transient faults with linear fitting.

Furthermore, these results need to be viewed in relation to the reliability graphs (Figure 5.6 and

5.7). As a reminder, the latency for a packet is calculated from the sum of the travel durations of successfully ejected packets divided by their number. However, this means that if only a small amount of packets are ejected early, before the network is so clogged that fewer packets reach their destination, their average latency would be lower than in a network, which continually ejects packets until the end of the simulation with a long travel duration. Counter-measures against clogged networks are for example the alternative paths of *rcTol*, which are not optimal because a longer travel path adds to the latency. Other reasons for a higher latency are the waiting cycle of the *ssm* configuration, additional traffic of the *retr* due to the ACK-Signal or transmission delays due to an increasing fault probability.
This explains why configurations with a low absolute reliability suddenly perform well in regard to their latency, meaning a low latency. Nevertheless, each combination has its own characteristics, which symbolizes how well it performs with congestions and how much overhead it adds to the travel duration.
For the analysis, the best reliability and latency techniques against the transient fault are evaluated. As it can be seen in Figure 5.9 *ecc* has one of the lowest latencies. In Figure 5.6 and 5.7 *ecc* also performs very well regarding reliability. The Hamming(7,4) based error correction of *ecc* achieves a good work to keep up with the transient faults as well as adding no latency overhead to the travel duration, making it a very good counter-measure to transient faults.
In combination with the secure state machine as *ssm-ecc* it also presents a good performance. Regarding reliability it is under the top three and in the latency results it appears in the middle field. Looking on *ssm* on its own it performs clearly below *ssm-ecc*.
Blank has presented a good reliability (cf. Figure 5.6 and 5.7). Looking at its performance regarding the latency it ranks in the middle field.
It is true that *retr-ecc-ssm-rcTol* also shows a lower latency but as stated above the number of packets successfully transmitted by this technique is much lower (low reliability), reducing the significance of this result.
The worst latency performer is *RcTol*. It has the highest latency but regarding reliability it ranks in the middle field. This means that packets are still delivered successfully but they take their time. The behavior can be explained because *rcTol* is based on rerouting packets.

To conclude, *ecc* shows the highest reliability and lowest latency results for transient faults and therefore appears to be the recommended option.

5.4.3 Intermittent Fault

This section analyzes the intermittent fault, which can be characterized by a longer recovery phase compared to transient faults. The Figures 5.10 and 5.11 present the relative and absolute reliability of the selected configurations (cf. Table 5.4).

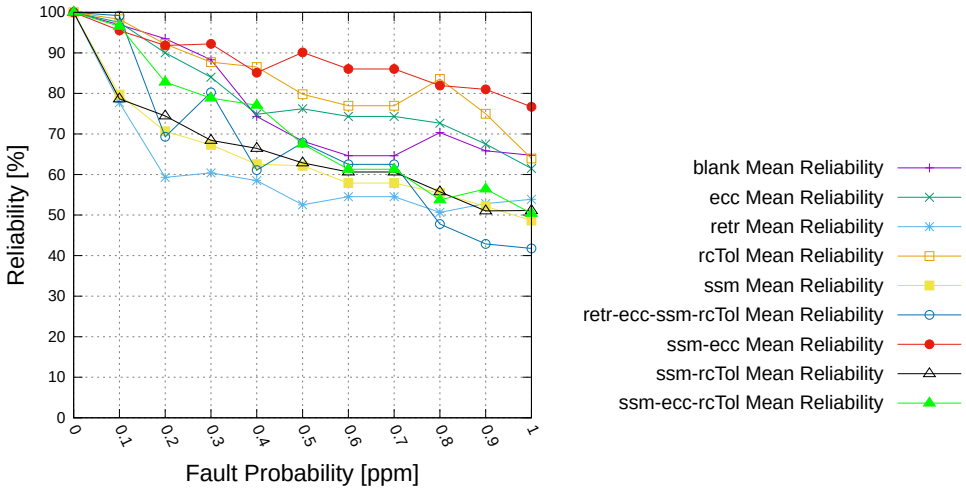


Figure 5.10: Relative reliability for intermittent faults.

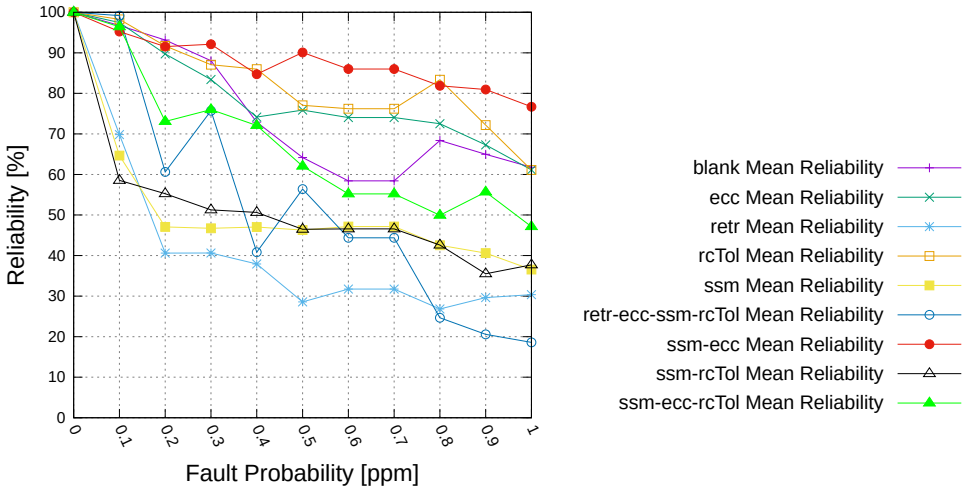


Figure 5.11: Absolute reliability for intermittent faults.

While *ecc* is in the top field, *rcTol* and *ssm-ecc* present an even better reliability (cf. Figures 5.10 and 5.11). Especially *rcTol* displays a different behavior compared to its performance in the transient fault simulations. Therefore, it shows that the reaction of fault tolerance method depends on the encountered fault appearance pattern. This is valuable information and proves the necessity of testing fault tolerance methods with different fault appearance patterns. Else a system will only be equipped with limited protection.
For this scenario of intermittent faults *ssm-ecc* proves to be the best tolerance method combination as it has a high reliability. *Ssm* on its own does not score very well but paired with *ecc* it performs exceedingly well. This points out the synergy between these techniques. However,

ssm-rcTol, another combination with *ssm*, presents a much lower reliability. This is interesting because *rcTol* on its own could be considered the second best method in regard to reliability. Therefore *ssm* has a negative impact in this combination with *rcTol*.
Furthermore, it can be noticed that the *retr* is not performing well again. While congestion is an essential factor for this result, the difficulties of this technique to counter longer active faults is a further reason. If a path suffers from often repeated errors, it will not help to resend the packet over it again and again.

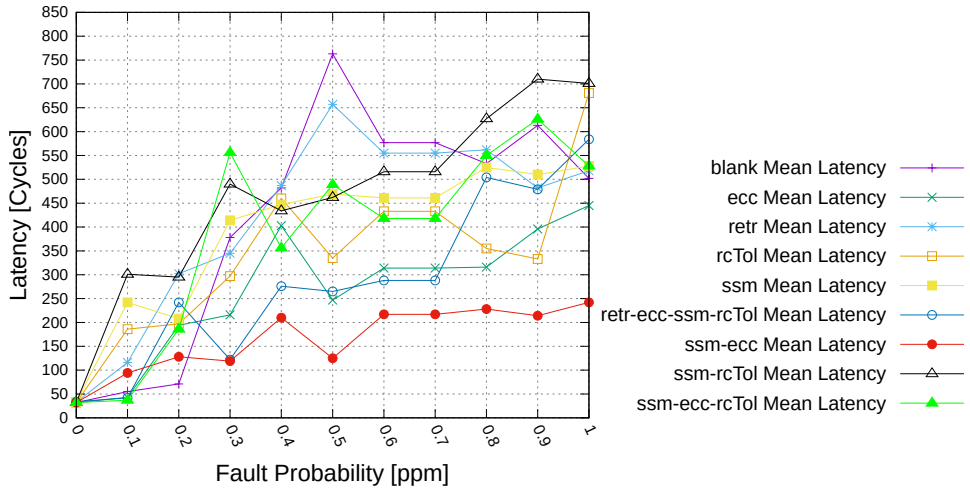


Figure 5.12: Latency results for intermittent faults.

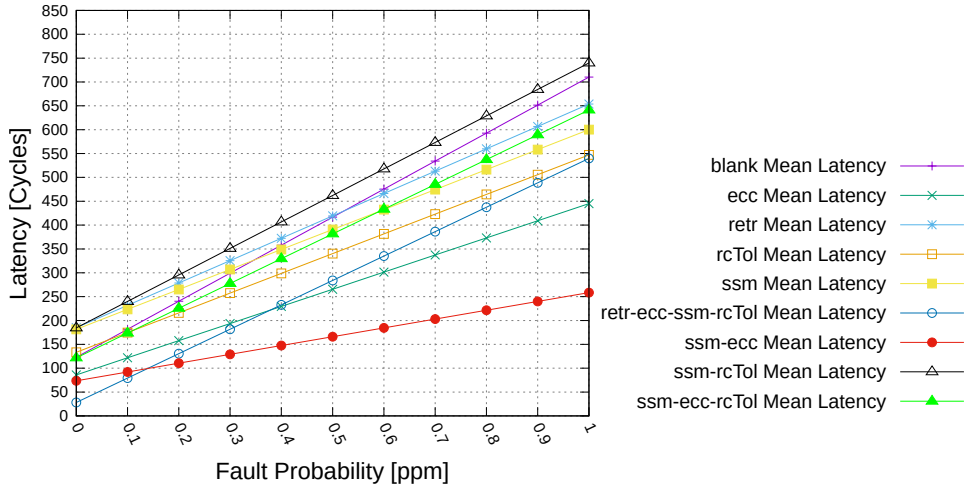


Figure 5.13: Latency results for intermittent faults with linear fitting.

The latency result can be seen in Figures 5.12 and 5.13. The overall latency is increased compared to transient faults, where it was in the range of 50 up 550 cycles. The intermittent faults cause a latency up to 800 cycles. However, taking into account that intermittent faults have a higher toll on the system's performance, this is an expected behavior. The latency results also highlight the different performance of the fault tolerance methods when encountering another fault appearance pattern.
While for the transient fault *ssm-ecc* was in the middle field of fault tolerant methods, it is pre-

senting the lowest latency for intermittent faults. It also has a very good reliability, therefore marking it as a recommendable fault tolerant method against intermittent faults.
ECC on its own (configuration *ecc*) has one of the top positions of the fault tolerant methods for transient faults, meaning a high reliability and a low latency. However, for intermittent faults it cannot maintain its leading position. Still, it is one of the better performing fault tolerant methods against intermittent faults.
Retr-ecc-ssm-rcTol has one of the lowest latencies (cf. Figures 5.12 and 5.13) but it shows one of the lowest reliabilities (cf. Figures 5.10 and 5.11). Looking at *rcTol*, which scored very well regarding reliability, it displays a poor latency behavior.

To conclude, *ssm-ecc* shows the highest reliability and the lowest latency results. Therefore, it seems to be the best choice when encountering intermittent faults.

5.4.4 Permanent Fault

Lastly, the permanent fault is evaluated. This fault pattern is characterized by its disability to recover. As expected, the reliability is the lowest of the three tested faults. The number of successfully transferred packets under the impact of permanent faults is low (for fault probability 1 ppm it is averaged below 200 packets) and only early injected packets are successfully delivered. Still, some packets could be passed through as it can be seen in Figure 5.14 for the relative reliability results and Figure 5.15 for the absolute reliability results. This appears reasonable because the network is fault free at the beginning. But over time the faults will accumulate, therefore stopping the further propagation of the packets. For the packet transfer this means that packets which are early injected and have a short travel distance (e.g., to a near neighbor router) can be delivered but the longer they stay in the network, due to travel distance or congested paths, the lower are their chances of a successful delivery. With the increasing fault probability, the accumulation of faults is speeding up, further reducing the reliability.

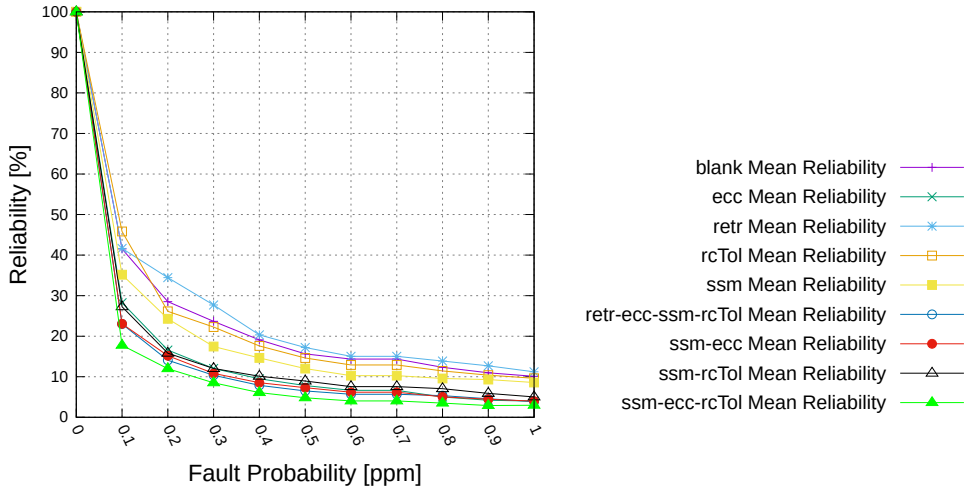


Figure 5.14: Relative reliability for permanent faults.

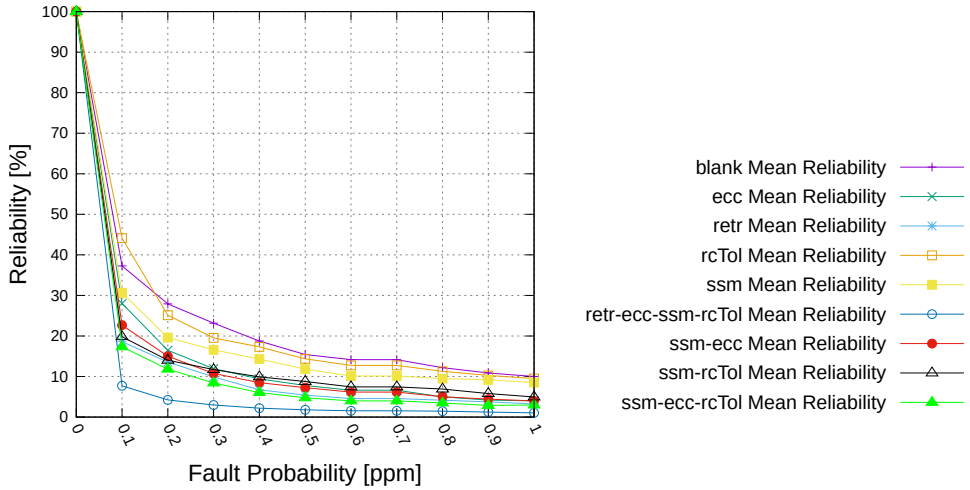


Figure 5.15: Absolute reliability for permanent faults.

Still, regarding reliability, there are some fault tolerance techniques that are better performing than others, in particular *blank*, *rcTol* and *ssm*. While the relative reliability for *retr* is one of the best, the absolute reliability shows that the amount of packets which it was able to inject is less compared to the other techniques. This leads to a reduced interest in this technique.
For *blank*, one of the better performing techniques, the lack of additional elements is an advantage because these elements cannot become "permanently faulty". They will not affect the system at the beginning of the simulation but also will not provide any fault tolerance during the rest of the simulation.
Even though *rcTol* has a lot of additional elements (assertions units, error counter) which have a chance of becoming "permanently faulty", it performs very well regarding relative and absolute reliability. It can be assumed that even if the additional elements become faulty, they do not have such a negative impact on the data transfer.
The last of the top three for the absolute reliability is *ssm*. While its wait-function will not help if a part is broken permanently, it can be seen as a small improvement to *blank* because it still checks the routing calculation result and is therefore prolonging the chance for other channels to succeed if their resource allocations are correct. However, this wait-function does not seem to have improved the *ssm*'s ability to counter these faults. Taking a closer look at *ssm-ecc*, one of the strong performing combinations against transient and intermittent faults, it cannot excel in this scenario. *Ssm-ecc* is behaving similar to *ecc* on its own. The graphs (cf. Figure 5.14 and 5.15) suggest that the added intercoder modules signal that the packet is of poor health, which lets the state machine drop it. In connection with permanent faults, this results in a too high toll on the data path of the system, leading to low reliability.

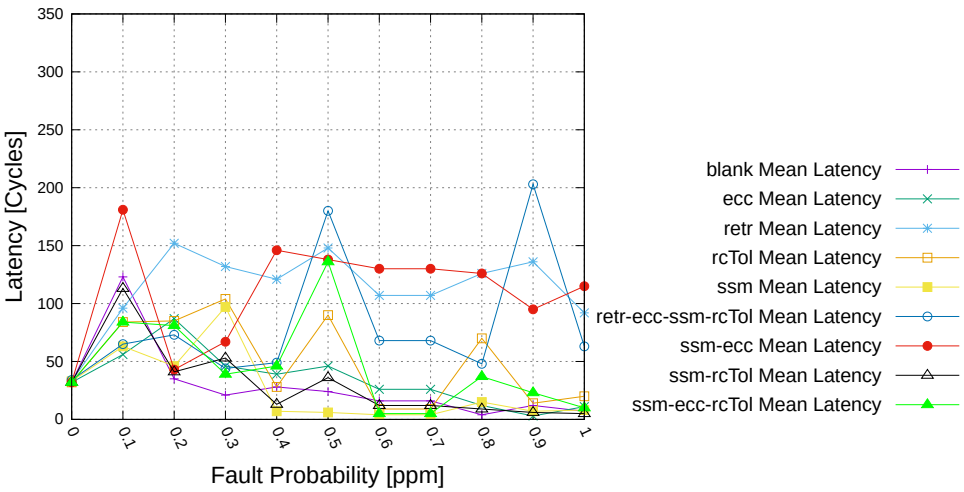


Figure 5.16: Latency results for permanent faults prior to the modification.

The first graph of the latency (cf. Figure 5.16) shows lots of peaks and drops. This seemed rather unusual and led to further analyzing. For this analysis the wave data of some simulations are evaluated. A first look was on the retransmission tolerance method because it had the highest peaks in Figure 5.16. The wave data (cf. Figure 5.17) displayed that the retransmission injection works under permanent faults and regularly injects packets inside the NoC except if the network input is blocked (input buffer full, which equals credit level one or lower). However, evaluating the wave data further for wrongly transmitted packets shows that a part of them were ACK-packets which can be seen as an example in Listing 5.1. The ACK-packets contain a wrong destination

because the acknowledgment should be sent back to the source of the acknowledged packet (i.e. the originator of the packet).

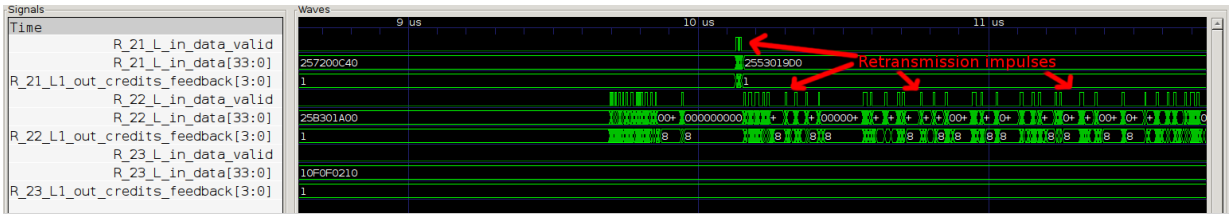


Figure 5.17: Retransmission pulses on router 21 and 22, router 23 is blocked.

Listing 5.1: Shortened and summarized log of wrongly ejected packets

```
$ cat sim_Server_G_SEED1000_pF_retr_Prob_2_vsim_log.txt | grep -i "Router.*56"
# Cycle: 216 - Packet (ID: 49, SRC: 38, DST: 48) got ejected at the WRONG NoC Interface (Router: 56)
# Cycle: 310 - Packet (ID: 163, SRC: 32, DST: 48) got ejected at the WRONG NoC Interface (Router: 56)
# Cycle: 365 - Packet (ID: 211, SRC: 18, DST: 48) got ejected at the WRONG NoC Interface (Router: 56)
# Cycle: 408 - Packet (ID: 231, SRC: 51, DST: 48) got ejected at the WRONG NoC Interface (Router: 56)
# Cycle: 415 - Packet (ID: 256, SRC: 8, DST: 48) got ejected at the WRONG NoC Interface (Router: 56)
```

But the destination for sending the acknowledgment is not correct, so the originator does not receive the ACK-packet. The returning ACKs never arrive at their source but their travel duration is added to the latency calculation. This issue progresses until the fault is fixed or in case of permanent faults the paths are broken. This led to a modification of the simulation environment to counter the effect of wrongly delivered packets on the latency calculation. The modification was installed and the simulation runs for all three fault patterns were executed again. The behavior of transient and intermittent faults did not show a significant change. However, the graphs used in their sections are the updated versions. But, for the permanent fault it displays some critical changes (cf. Figure 5.18).

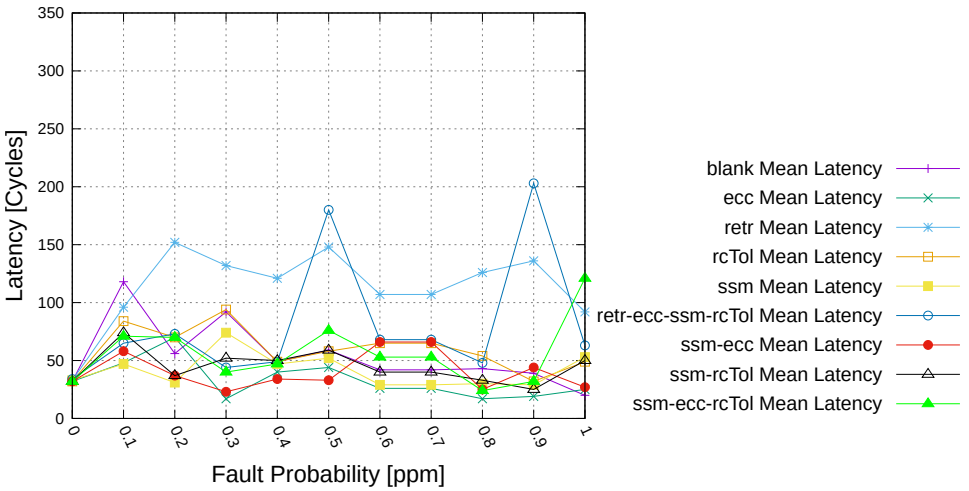


Figure 5.18: New latency results for permanent faults.

This is because the system now compares if the *headflit* destination address matches with the node which receives it. If it matches, it is perceived as successfully transmitted and will be used in the latency calculation. If it does not match the packet will be dropped and will not be included in the latency calculation. This can be seen as a compromise as some transmitted packets will arrive at a destination, which matches their *headflit* but was not their original destination. This

modification changed the latency graph (cf. Figure 5.18). The latency of some fault tolerance methods decreased (e.g. *ssm-ecc*) and also the amount of peaks and drops of some methods were reduced (e.g. *rcTol*). Overall many configurations (e.g. *blank*, *rcTol*, *ecc*) have a low latency, which goes down even further with the growing fault probability (cf. Figure 5.18). However, some peaks are still visible. This can be explained by the fast degrading performance of the NoC under the effect of permanent faults. It can be argued that early injected packets, which travel only a short duration, are able to leave the network before the congestion becomes too strong. Thus, leading to a low latency because most longer traveling packets were not able to reach their destination but for calculating the average latency it is necessary that the packet reaches its destination. Otherwise it will not be included in the calculation. A reason for the peaks could be that some packets were able to arrive at their destination but it took a large amount of cycles due to the congestion, which adds to the average latency. The configurations containing the retransmission technique (*retr*, *retr-ecc-ssm-rcTol*) display the highest peaks. For the increased latency of *retr-ecc-ssm-rcTol* it can be argued that after a blockage has been solved by one of the fault tolerant methods, the retransmitted packets (including the ACKs) are able to reach their destination, thus adding up their longer travel duration to the average latency. While this would indicate that *retr-ecc-ssm-rcTol* is able to cope with permanent faults to a certain extent, the actual low amount of transmitted packets weakens their performance.

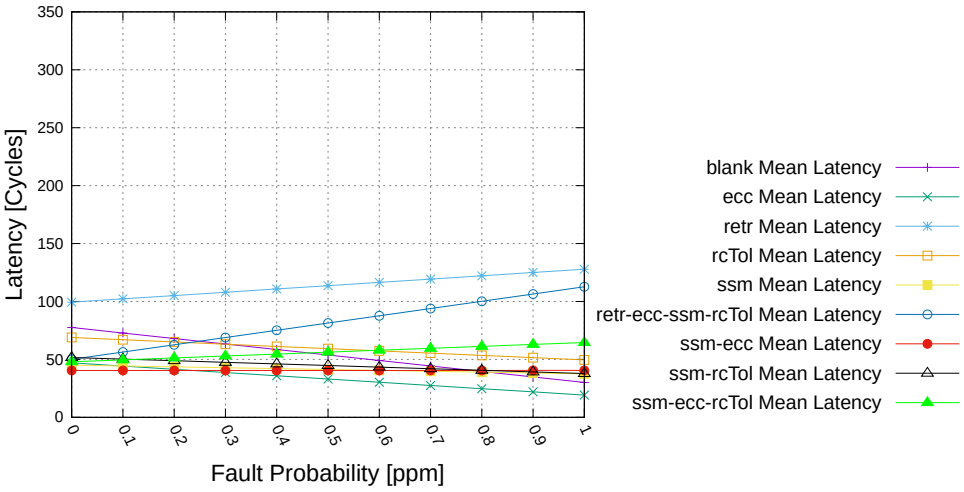


Figure 5.19: Latency results for permanent faults with linear fitting.

To gain a better visual for the latency graph, it was decided to use linear regression again (cf. Figure 5.19). The configurations containing the retransmission technique show a growing behavior. This distortion is caused by the fault-free first simulation and its ideal latency value. Removing the fault free results from the linear fitting of Figure 5.19 results in the latency graph in Figure 5.20.
There it shows that *retr* is actually slightly falling but the *retr-ecc-ssm-rcTol* is still growing, which is caused by the peaks discussed in the paragraph before. This could be due to negative synergies between the fault tolerant methods. *Retr*, *ecc*, *ssm* and *rcTol* show on their own a falling behavior but together they indicate a growing latency. Considering Figure 5.18, where *retr* has a general high latency, it can be argued that *retr* is having a negative impact on their performance on *retr-ecc-ssm-rcTol*. To conclude, the rising behavior of the *retr-ecc-ssm-rcTol* can be explained by the ability of this fault tolerant method to deliver some packets after a solved blockage due to retransmission.

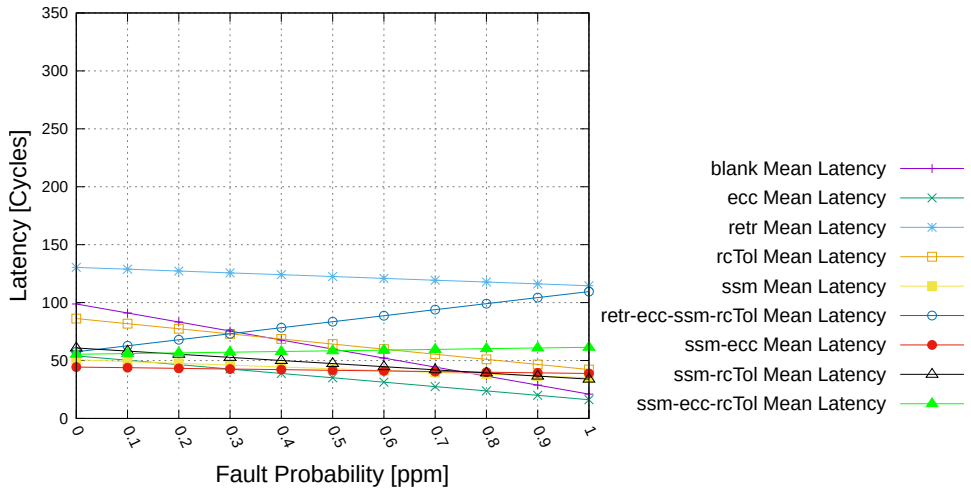


Figure 5.20: Latency results for permanent faults with linear fitting with no fault free runs.

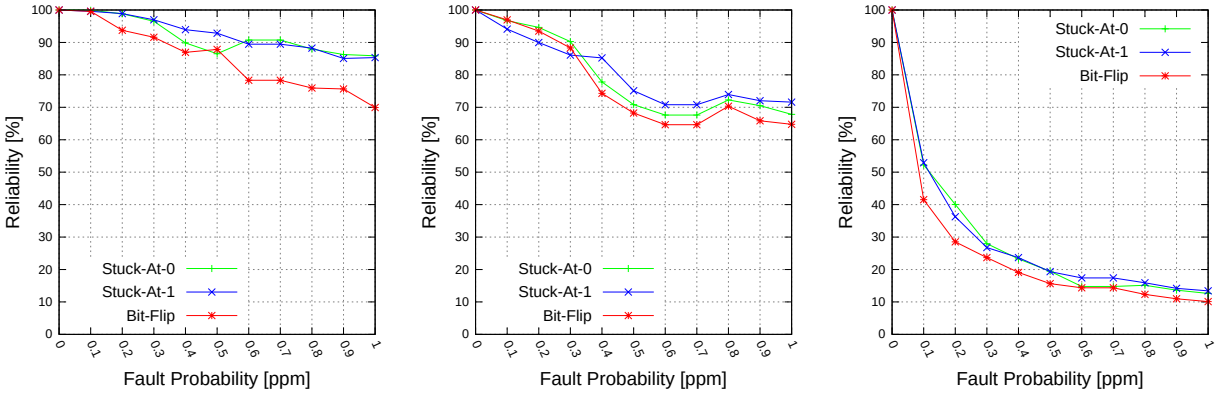
Overall, none of the fault tolerance methods presented a significant performance against permanent faults. However, it should be noted that the most constant behaving latency result is shown by the *ssm-ecc* technique. While its reliability is in the average field, the result of the latency graph suggests that it is impacted less strongly by the faults (cf. Figure 5.20). The ECC is capable of detecting faults in the packets, letting the secure state machine drop the particular packet, which still costs cycles but allows the next packet to reach its destination as well as it keeps other resources free for further allocation.
Blank appears to cope better regarding its reliability for permanent faults but its latency is shown to be in the middle field compared to the other fault tolerant methods. The result in reliability can be explained as *blank* is lacking any additional elements which can become permanently faulty.
While the reliability of the fault tolerant methods under the effect of permanent faults has a distinct result, the latency behaves differently than expected. On closer inspection, this behavior is understandable and points out how crucial this kind of faults are for a NoC. The critical nature of this fault pattern may create the idea that it was used too carelessly in these simulations and the probabilities should have been reduced. However, to be able to compare the different error types, it was necessary to give them all the same settings, which of course resulted in a more drastic output in this situation with permanent faults.

5.4.5 Bit Flip and Stuck-At Faults

As already mentioned, the *Bit-Flip* failure type was selected as it arguably has the most hostile effect on the system. However, to assure that this failure type has in fact the biggest impact, additional simulations (198 simulation runs) were executed. Each appearance pattern (transient, intermittent and permanent) was tested with the fault module modified to represent *Stuck-At-0* and *Stuck-At-1* errors. The simulation setup is the same as for *Bit-Flip* (2000 packets, 20000 cycles).

In Figure 5.21 the impact on the relative reliability is presented. It can be noted that as expected the *Bit-Flip* fault has the worst reliability of the three faults but the distance between these faults was expected to be larger. This foots on the reasoning that a *Stuck-At* error will not cause an

actual error in all of its appearances. For example if the gate output should be 1 and the *Stuck-At-0* was selected as failure type and active, it would be a logical error. However, if the gate output is 0 and therefore equals the error output, it would be masked and thus not be seen as an error by the system. The *Bit-Flip* on the other hand would cause an error in both cases because it flips the signal both situations. So it has more possibilities to create errors in the system, which manifests as anticipated in a lower reliability as it can be seen in Figure 5.21. It can also be seen that while for transient faults the distance between *Stuck-At* and *Bit-Flip* errors grows with increasing fault probability, for intermittent and especially permanent faults their lines are more similar and the gap between them is smaller. This led to the conclusion that while *Bit-Flip* has a worse effect than the other failure types, the appearance pattern has a significant influence on the impact of the error. It showcases that how the error is created is not as important as the duration it stays. It is also interesting that the different *Stuck-At* errors have a very similar reliability. It could have been assumed that the network would suffer under one *Stuck-At* fault much more than the other due to the *Stuck-At* characteristic of being masked if the gate output equals the failure type output and a tendency of a system's logic circuit to have either 0 or 1 as a more common logic output. However, that is not the case, both *Stuck-At* lines in Figure 5.21 are close together over the different appearance patterns, which leads to the realization that the difference between these failure types is small but the appearance patterns have a strong influence on the failure behavior of the system. Further focusing on the comparison between the *Stuck-At-1* and *Bit-Flip* error for the transient and intermittent fault appearance patterns, it can be recognized that their paths are partly quite similar. This leads to the assumption that the impact of these errors is similar or that many *Bit-Flip* errors inside the system are caused by gates resembling the *Stuck-At-1* error.

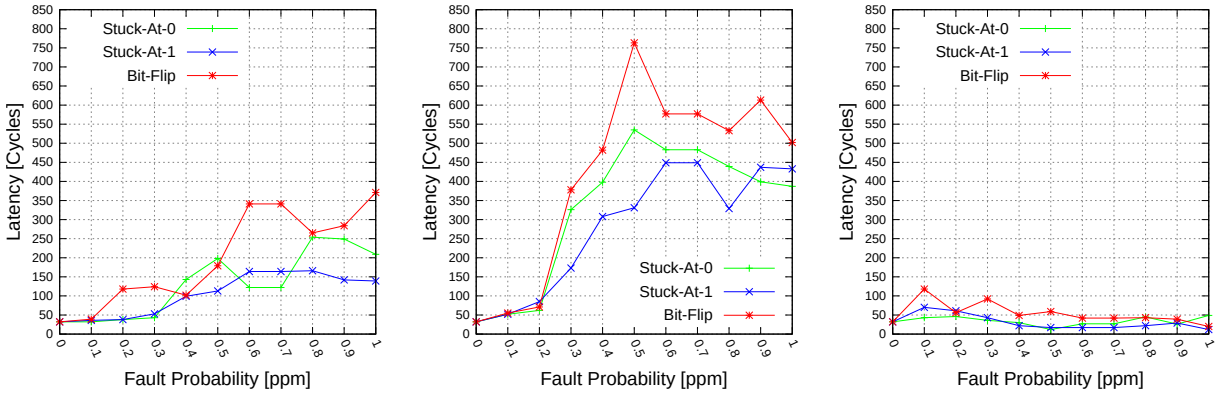


(a) Fault Pattern: Transient. **(b)** Fault Pattern: Intermittent. **(c)** Fault Pattern: Permanent.

Figure 5.21: Average relative reliability for different logic errors under different fault patterns.

The latency graphs (cf. Figure 5.22) show a similar result. While the *Bit-Flip* error has overall a higher latency than the others, the *Stuck-At* errors keep close together. The low latency graph of the permanent faults was already analyzed in Subsection 5.4.4 and can be explained by a low number of successfully delivered packets, which only needed to pass a short distance until the faults made the network impossible to pass through. So seeing the high latency of the *Bit-Flip* error in combination with its low reliability suggests that this error has the most severe effect on the NoC.

To conclude, the *Bit-Flip* error causes the lowest reliability in the NoC, which emphasizes the decision to use it for all the simulations.



(a) Fault Pattern: Transient. **(b)** Fault Pattern: Intermittent. **(c)** Fault Pattern: Permanent.

Figure 5.22: Average latency for different logic errors under different fault patterns.

5.5 Summary

In conclusion it can be noted that the fault injection process on gate level has proven to be a convincing method to evaluate fault tolerance methods. The behavior of different fault appearance patterns was represented and different logic errors were also successfully implemented, both affecting the NoC negatively with an increasing fault probability.
The evaluation results show several findings.

1. The number of gates in a system is not directly connected to the reliability. Otherwise, the configuration with the ECC modules (cf. Table 5.4) would have performed worse than the other fault tolerance methods. It is more important where the additional logic circuit is located. In the case of *ecc* it is placed in the data path, where faults are able to corrupt the data of the FLITs. This becomes an issue if it affects the routing logic (*headflit*) or if the faults corrupt the identification of the FLITs. However, if the additional logic is located in the control path, it has a more severe effect. This is the case for *rcTol*, which does not perform that well. So it can be resumed that while large circuits are more likely to have more faults, the system's reliability depends more strongly on where the logic is integrated into the NoC and therefore where the faults can affect the system.
   In Section 5.2 the configuration *ecc* showed the highest fault probability due to the number of elements. However, it was assumed that it would perform well because of the position of the fault modules, which are inside the data path. Generally, the ECC technique was a guarantor for good reliability in the simulations without influencing the latency that much and leading in the reliability results for transient faults. Therefore, the results solidify this previous assumption.

2. The simulations of combined fault tolerance methods (*ssm-ecc*, *ssm-rcTol*, *ssm-ecc-rcTol* and *retr-ecc-ssm-rcTol*) show that positive and negative synergies between mechanisms exist but the combinations must be chosen carefully and evaluated in detail. Of the four presented combinations only *ssm-ecc* had an above-average reliability. Furthermore, it showed that the general idea of adding more fault tolerance methods (e.g. *ssm-ecc-rcTol*, *retr-ecc-ssm-rcTol*) for better performance is not correct. The number of gates increases due to the additional circuits, which also increases the chances that a fault can appear but the ability

to tolerate faults does not seem to add up. It is also important where the additional logic is placed (as it is argued in point 1) and what the specific fault tolerant method's characteristic is for countering faults. Furthermore, the methods can become faulty themselves therefore creating an unbearable amount of faults. However, if certain fault tolerance are chosen wisely they can show positive synergies as they work together.

3. *Ssm-ecc* displayed the most promising fault tolerance performance in these simulations, but is has to be considered that the fluctuation in the simulation runs could cause further changes to this results. Under the effect of transient faults *ssm-ecc* maintains a good position especially against higher fault probabilities, and during the intermittent fault simulations it was constantly in the top position. The general performance of all methods against the permanent faults was weak, which is understandable because of the strong impact on the system and how these faults will be accumulated over time. A lower fault probability would probably lead to a more realistic appearance of such errors, but the comparison with the same values was essential and necessary for this thesis. Furthermore, the strong impact of congestion on the NoC points out its vulnerability against any form of a stuck packet.

4. Lastly, the simulation results further demonstrate that certain methods have different reliabilities against different fault appearance patterns, which can be seen for *rcTol* in the transient and intermittent fault simulations. This emphasizes the advantage of the evaluation environment of this thesis, which is flexible enough to include different simulation scenarios.

As a result, fault tolerance methods need to be chosen wisely and evaluated often early on in the development process to secure a good performance. Considering to achieve this manually is an unwieldy task for such large systems, therefore contending that the simulation approach taken in this thesis is a right approach.

6 Conclusion

In this chapter, an overview of the thesis and its results is given. In addition, an outlook for possible future enhancements is provided.

6.1 Results

The technology development allows to produce Integrated Circuit (IC) with an unprecedented density of transistors so that more elements than ever before can be placed on one die. Especially, the generation and development of System on Chips (SoCs) benefit from this development but require adapted communication systems for best usage. Network on Chips (NoCs) are the most promising solution but, as all ICs, they are prone to suffer from faults caused by environmental issues like heat, radiation or aging. Due to that fault tolerance becomes more and more a necessary component when developing a modern multi-core system.
Evaluating its importance, integrating it into the design process early and showing how the different fault tolerance methods can affect the NoC's performance is a crucial aspect to keep future developments running stable. In this thesis, the behavior of fault tolerance of a NoC is evaluated and faults are not only injected into the NoC but also into its fault tolerance methods. The simulation results proved that injecting fault modules in all elements of a NoC is a working implementation for fault tolerance evaluation and pointed out how sensitive the test NoC reacts to different configurations.
Furthermore, the evaluation of the implemented fault-tolerant routing displayed that a fault tolerant system needs to be tested against different error types because each technique has their weak and strong points against them. Also, the assumption that adding more fault tolerance techniques to a system makes it more reliable is disproven. These parts can become faulty as well and therefore put additional stress on the system, so an optimum needs to be found. On the other hand, not using any fault tolerance methods is not ideal either. For instance the *blank* configuration performed well for transient faults but not for intermittent faults, indicating that fault tolerance methods are needed. Analyzing the results, methods like the secure state machine and the Error Correction Code (ECC) need to be highlighted as a great example of fault tolerance methods. By adding the improved state machine to the overall well-working ECC it achieves higher reliability against intermittent faults.
An overall weak performance is displayed by the re-transmission technique, making it a not recommendable protection method in this implementation. The cause for its weak performance can be explained by the additional traffic load. Clogging seems to be the major problem of this implementation due to tight calculated retransmission windows and possibly due to the sensitivity of the test NoC.

Despite this thesis' focus on the fault injection and fault tolerance methods, it needs to be mentioned that the development of the test NoC was very time consuming and needed several iterations to work well. At first, the fault-free runs presented a satisfying system to start the fault injection process, but under faults a better behavior than the shown results were expected. On the other hand, the fault injection script and the fault modules were working well and allowed an easy configuration of the system.
To conclude it can be said that the developed system is capable of injecting faults into *Verilog* based Hardware Description Language (HDL) code and by doing so, the weaknesses of a system against faults can be detected. This thesis emphasizes the importance to test systems for this behavior. It can be seen that this research area has lots of possibilities so that this thesis presents a successful evaluation and implementation for future projects.

6.2 Outlook

One shortcoming of this approach is the high amount of memory needed and the long duration of the simulations. This is especially a drawback for NoCs, whose key feature is to combine lots and lots of elements. Finding a concept to infect only certain parts/elements while keeping close to real fault models could ease this problem. However, fault injection on this level will always demand more resources than models on higher levels.
Also, the fault modules could be modified to resemble the fault in even more detail. While it is not possible to know the placement of wires to get additional information to resemble crosstalk, which is a main source for transient faults, a counter could be used to increase the fault probability in relation to the amount of value changes of a gate, going even further and changing it to a permanent fault after a certain limit is reached. Otherwise, it could be useful to attach individual fault modules to specific gates, therefore allowing different fault behaviors to be evaluated within one simulation run.
Another point which appeared during the development is that the `$random()` function provided by Questasim, while it fulfills all needs for a first test, is seen as an open issue left for improvement. Even if different seeds were used, specific patterns seem to repeat. However, modern simulation environments allow adding own *C* functions to the code, which are a possible solution for this issue by adding an own random number generator.
Furthermore, regarding the simulation aspect of this thesis, it needs to be noted that the tests were all started on a "cold" system, so there was no warm-up phase by pre-injecting packets. Developing and adding the necessary technical elements for this could add additional usage to the system, resembling real systems even more. It can be expected that the results would differ, but it would require severe modifications of packet tracking and enhancing the fault module.
Generally, the tracking of packets, would be an exciting enhancement to such a system as it would allow following the action inside the network better without depending on wave signal diagrams. While a tool for this purpose was tested, the issue was that it expected the network to run free of faults. However, the simulations of this thesis included faults, which caused logic errors in the system and led to the tool's failure. Still, such a tracking program could provide further insight of the causes for reliability or latency issues within a network.
Lastly, maybe the most interesting point is how other NoCs perform under the influence of faults. With small modifications it should be possible to use this thesis' fault injection process for injecting fault modules in other NoCs, which are synthesizable by Synopsis *Design Compiler*. This would allow to compare different systems further and find the best-suited one against a specific fault pattern.

Literature

[ACM/SIGDA12] ACM/SIGDA (Veranst.): *CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs*. 2012

[AM10] Ahmadian, S. N. ; Miremadi, S. G.: Fault injection in mixed-signal environment using behavioral fault modeling in Verilog-A. In: *Proc. IEEE Int. Behavioral Modeling and Simulation Conf. (BMAS)*, 2010. – ISSN 2160–3804, S. 69–74

[AWAN06] Ali, M. ; Welzl, M. ; Adnan, A. ; Nadeem, F.: Using the NS-2 Network Simulator for Evaluating Network on Chips (NoC). In: *Proc. Int. Conf. Emerging Technologies*, 2006, S. 506–512

[AZl12] Abdelmalek, G. A. ; Ziani, R. ; laghrouche, M.: Fault injection for verifying testability of fault tolerant structures at the Verilog level. In: *Proc. 24th Int. Conf. Microelectronics (ICM)*, 2012. – ISSN 2159–1660, S. 1–4

[BH09] Birner, M. ; Handl, T.: ARROW - A Generic Hardware Fault Injection Tool for NoCs. In: *Proc. 12th Euromicro Conf. Digital System Design, Architectures, Methods and Tools DSD '09*, 2009, S. 465–472

[BM02] Benini, Luca ; Micheli, Giovanni D.: Networks on chip: a new paradigm for systems on chip design. In: *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, IEEE, März 2002

[BM06] Bjerregaard, Tobias ; Mahadevan, Shankar: A Survey of Research and Practices of Network-on-Chip / Technical University of Denmark. 2006. – resreport

[BPC98] Boué, Jérome ; Pétillon, Philippe ; Crouzet, Yves: MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance. (1998), S. 168–173

[CMM+15] Catania, Vincenzo ; Mineo, Andrea ; Monteleone, Salvatore ; Palesi, Maurizio ; Patti, Davide: Noxim: An open, extensible and cycle-accurate network on chip simulator. In: *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on* IEEE, 2015, S. 162–163

[DT03] Dally, William J. ; Towles, Brian ; Elsevier (Hrsg.): *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003

[Ebr13] Ebrahimi, Masoumeh: Reliable and Adaptive Routing Algorithms for 2D and 3D Networks-on-Chip. In: *Routing Algorithms in Networks-on-Chip*. Springer, 2013. – ISBN 978–1–4614–8273–4, Kapitel 9

[EYPZ09] Eghbal, A. ; Yaghini, P. M. ; Pedram, H. ; Zarandi, H. R.: Fault injection-

based evaluation of a synchronous NoC router. In: *Proc. 15th IEEE Int. On-Line Testing Symp*, 2009. – ISSN 1942–9398, S. 212–214

[FL10] Fu, Zhizhou ; Ling, Xiang: The design and implementation of arbiters for Network-on-chips. In: *Industrial and Information Systems (IIS), 2010 2nd International Conference on* Bd. 1 IEEE, 2010, S. 292–295

[GAP+] Grecu, C. ; Anghel, L. ; Pande, P. P. ; Ivanov, A. ; Saleh, R.: Essential Fault-Tolerance Metrics for NoC Infrastructures. In: *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, S. 37–42

[HCL+] Han, J. ; Chen, H. ; Liang, J. ; Zhu, P. ; Yang, Z. ; Lombardi, F.: A Stochastic Computational Approach for Accurate and Efficient Reliability Evaluation. 63, Nr. 6, S. 1336–1350. – ISSN 0018–9340

[HW16] Huang, H. ; Wen, C. H. ..: Layout-Based Soft Error Rate Estimation Framework Considering Multiple Transient Faults—From Device to Circuit Level. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35 (2016), April, Nr. 4, S. 586–597. – ISSN 0278–0070

[KH04] Karnik, T. ; Hazucha, P.: Characterization of soft errors caused by single event upsets in CMOS processes. In: *IEEE Transactions on Dependable and Secure Computing* 1 (2004), April, Nr. 2, S. 128–143. – ISSN 1545–5971

[KPN+05] Kim, J. ; Park, D. ; Nicopoulos, C. ; Vijaykrishnan, N. ; Das, C. R.: Design and analysis of an NoC architecture from performance, reliability and energy perspective. In: *Proc. Symp. Architecture for networking and communications systems ANCS 2005*, 2005, S. 173–182

[LYA16] Li, C. ; Yang, M. ; Ampadu, P.: An Energy-Efficient NoC Router with Adaptive Fault-Tolerance Using Channel Slicing and On-Demand TMR. In: *IEEE Transactions on Emerging Topics in Computing* PP (2016), Nr. 99, S. 1. – ISSN 2168–6750

[Man13] Mansour, Wassim: An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs. In: *IEEE Transactions on Nuclear Science* 60 (2013), Nr. 4, S. 2728 – 2733

[MO09] Marculescu, Radu ; Ogras, Umit Y.: Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives / IEEE. 2009. – Forschungsbericht

[NND09] Nicopoulos, Chrysostomos ; Narayanan, Vijaykrishnan ; Das, Chita R.: *Network-on-Chip Architectures: A Holistic Design Exploration (Lecture Notes in Electrical Engineering)*. Springer, 2009. – ISBN 978–90–481–3030–6

[NPCZR15] Nunes, J. L. ; Pecserke, T. ; Cunha, J. C. ; Zenha-Rela, M.: FIRED – Fault Injector for Reconfigurable Embedded Devices. In: *Proc. IEEE 21st Pacific Rim Int Dependable Computing (PRDC) Symp*, 2015, S. 1–10

[PM05] Pande, Partha P. ; Micheli, Giovanni D.: Design, Synthesis, and Test of Networks on Chips / Washington State University. 2005. – Forschungsbericht

[PNK+06] Park, Dongkook ; Nicopoulos, C. ; Kim, Jongman ; Vijaykrishnan, N. ; Das, C. R.: Exploring Fault-Tolerant Network-on-Chip Architectures. In: *Proc. Int. Conf. Dependable Systems and Networks (DSN'06)*, 2006. – ISSN 1530–0889, S. 93–104

[PPNS12] Prodromou, A. ; Panteli, A. ; Nicopoulos, C. ; Sazeides, Y.: NoCAlert: An On-Line and Real-Time Fault Detection Mechanism for Network-on-Chip Architectures. In: *2012 45th Annual IEEE/ACM*, 2012 45th Annual IEEE/ACM, 2012, S. 60–71

[RFZJ13] Radetzki, Martin ; Feng, Chaochao ; Zhao, Xueqian ; Jantsch, Axel: Methods for Fault Tolerance in Networks-on-Chip / University of Stuttgart. 2013. – resreport

[SSM08] Shokrolah-Shirazi, Mohammad ; Miremadi, Seyed G.: FPGA-based fault injection into synthesizable verilog HDL models. In: *Secure System Integration and Reliability Improvement, 2008. SSIRI'08. Second International Conference on* IEEE, 2008, S. 143–149

[SSM16] Suvorova, E. ; Sheynin, Y. ; Matveeva, N.: Reconfigurable NoC development with fault mitigation. In: *Proc. 18th Conf. of Open Innovations Association and Seminar Information Security and Protection of Information Technology (FRUCT-ISPIT)* IEEE, 2016, S. 335–344

[Sys13] SystemVerilog: IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. In: *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)* (2013), S. 1–1315

[SZBR09] Schonwald, T. ; Zimmermann, J. ; Bringmann, O. ; Rosenstiel, W.: Network-on-Chip Architecture Exploration Framework. In: *Proc. 12th Euromicro Conf. Digital System Design, Architectures, Methods and Tools DSD '09*, 2009, S. 375–382

[TAZ03] Thaker, P. A. ; Agrawal, V. D. ; Zaghloul, M. E.: A test evaluation technique for VLSI circuits using register-transfer level fault modeling. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22 (2003), Aug, Nr. 8, S. 1104–1113. – ISSN 0278–0070

[TY15] Takamaeda-Yamazaki, Shinya: Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In: *Applied Reconfigurable Computing* Bd. 9040, Springer International Publishing, Apr 2015, S. 451–460

[Ver06] Verilog: IEEE Standard for Verilog Hardware Description Language. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), S. 1–560

[WEH+] Wang, J. ; Ebrahimi, M. ; Huang, L. ; Jantsch, A. ; Li, G.: Design of Fault-Tolerant and Reliable Networks-on-Chip. In: *2015 IEEE Computer Society Annual Symposium on VLSI*, S. 545–550

[Whi10] White, Mark: Scaled cmos technology reliability users guide / Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2010. 2010. – Forschungsbericht

[WHL+16] Wang, Junshi ; Huang, Letian ; Li, Qiang ; Li, Guangjun ; Jantsch, Axel: Optimizing the Location of ECC Protection in Network-on-chip. In: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA : ACM, 2016 (CODES '16). – ISBN 978–1–4503–4483–8, S. 19:1–19:10

[WJ08] Wolf, Wayne ; Jerraya, Ahmed A.: Multiprocessor System-on-Chip (MPSoC) Technology / IEEE. 2008. – Forschungsbericht

[YA10] Yu, Q. ; Ampadu, P.: A Flexible Parallel Simulator for Networks-on-Chip With Error Control. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29 (2010), Jan, Nr. 1, S. 103–116. – ISSN 0278–0070

[ZME03a] Zarandi, H. R. ; Miremadi, S. G. ; Ejlali, A.: Dependability analysis using a fault injection tool based on synthesizability of HDL models. In: *Proc. 18th IEEE Int Defect and Fault Tolerance in VLSI Systems Symp*, 2003. – ISSN 1550–5774, S. 485–492

[ZME03b] Zarandi, H. R. ; Miremadi, S. G. ; Ejlali, A.: Fault injection into verilog models for dependability evaluation of digital systems. In: *Proc. Second Int Parallel and Distributed Computing Symp* IEEE Computer Society, 2003, S. 281–287

Internet References

[1] News: Intel Retains Its Crown For Highest Number Of CPUs Sold This Quarter - https://wccftech.com/intel-sells-most-cpus-this-quarter/, Last visited: 2018-07-15.

[2] Source ITRS - Shrinking transistor size from 1971 till 2018 - https://en.wikipedia.org/wiki/International_Technology_Roadmap_for_Semiconductors, Last visited: 2018-10-21.

[3] Mentor Graphic's Questasim Website: https://www.mentor.com/products/fv/questa/, Last visited: 2018-10-20.

[4] Synopsis Design Compiler Homepage: https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html, Last visited: 2018-10-20.

[5] Synopsis VCS Homepage: https://www.synopsys.com/verification/simulation/vcs.html, Last visited: 2018-10-20.

[6] Python Homepage: https://www.python.org/, Last visited: 2018-10-20.

[7] pyverilog Repository: https://github.com/PyHDI/Pyverilog, Last visited: 2018-10-20.

[8] Icarus Verilog Homepage: http://iverilog.icarus.com/, Last visited: 2018-10-20.

[9] Jetbrain's Pycharm IDE Homepage: https://www.jetbrains.com/pycharm/, Last visited: 2018-10-20.

[10] August 2010. Popnet Website: http://www.pudn.com/Download/item/id/1274390.html, Last visited: 2018-07-13.

[11] 2012. CONNECT Homepage: https://users.ece.cmu.edu/~mpapamic/connect/, Last visited: 2018-07-07.

[12] June 2015. Noxim Repository: https://github.com/davidepatti/noxim, Last visited: 2018-07-13.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 10.02.2019 _____

Lukas Temmel